LEER JEZELF

PROGRAMMEREN

EN

ONTWERPEN

EEN INLEIDING VOOR DE (BEGINNENDE) PROGRAMMEUR
VOOR HET VERWERVEN VAN INZICHT TEN BEHOEVE VAN
HET BOUWEN VAN COMPUTERPROGRAMMA'S
BIJ BEDRIJVEN EN INSTELLINGEN

MARC H.E. VOS

Auteurs- en Copieerrecht notitie

Het is alleen aan de auteur voorbehouden de inhoud van dit boek op enige wijze te veranderen en, eventueel opnieuw, te publiceren, op wat voor wijze dan ook.

Copiëren en uitdelen, ook van onderdelen, van het boek is toegestaan onder de voorwaarde dat ten allen tijde het voorblad en deze shareware- en copyright-pagina met de copie worden meegeleverd zodat duidelijk is dat ook van de ontvanger van de copie verwacht wordt de shareware-fee te betalen.

Afstandverklaring

Het materiaal in deze uitgave is uitsluitend bestemd ter informatie waaraan geen verdere rechten kunnen worden ontleend.

Het materiaal kan zonder voorafgaande waarschuwing door de auteur worden veranderd, ingekort of uitgebreid.

Ondanks alle aan de samenstelling van dit boek bestede zorg kan de auteur geen aansprakelijkheid aanvaarden voor schade die het gevolg is van enige fout in of wat er mist aan deze uitgave of van de handelswijze van wie dan ook op basis van informatie uit deze uitgave.

Indien u enig commentaar hebt op de inhoud van dit boek of juist over inhoud die er niet is maar toegegvoegd zou moeten worden, dan kunt dat e-mailen naar marc@vos.net.

WAAROM DIT BOEK

Het werken met computers is één van de dingen die een mens erg leuk vindt om te doen. Op de een of andere manier wordt het apparaat een onmisbaar onderdeel van onze samenleving. Tekstverwerkers en spelletjes en andere computerprogramma's ontstaan echter niet vanzelf; daar zijn programmeurs voor nodig. En wat zijn dan eigenlijk allemaal computerprogramma's? Eigenlijk alles wat er voor zorgt dat de computer kan worden gestart, dat het apparaat en de aangesloten randapparatuur blijft werken, dat er met een muis geklikt kan worden, dat men tekst kan tikken, dat men kan scannen, foto's bewerken, enzovoort. Om er een paar te noemen: MacOS, MS-DOS, MS-Windows, OS/400, VMS, MS-Word, WordPerfect, MacWrite, Photoshop, Freehand, Dreamweaver, GoLive, RagTime, PowerPoint, Wordpad, Kladblok, Simpeltekst, en nog veel, veel meer.

Besturingssystemen zijn programma's die de computer moeten besturen en toepassingsprogramma's en stuurprogramma's moeten kunnen verwerken. MacOS, Unix en MS-Windows zijn wel de drie bekendste onder de vele andere. Toepassingsprogramma's zijn programma's waarmee iets nuttigs kan worden gedaan. Voorbeelden zijn tekenprogramma's, tekstverwerkers, handelspakketten, boekhoudsoftware, noem maar op. Weer een andere categorie zijn de stuurprogramma's (*drivers*). Dit type software stuurt specifieke hardware aan zoals een CD-speler die aan je computer is gekoppeld. Of een scanner. Of een printer. En de categorie die voor ons van belang is zijn de ontwikkelomgevingen (*Development Environment*), welke zijn geprogrammeerd met als doel het andere programmeurs weer makkelijker te maken programma's te schrijven. Een Visual Java of Visual Basic ontwikkel omgeving is heel waarschijnlijk met Visual C++ en machinetaal gebouwd. En Visual C++ is weer met gewoon C en machintaal ontwikkeld.

Om de beginnende programmeur een goede, snelle start te geven in de wereld van het programmeren, met de nadruk op het besturingssysteem- en taalonafhankelijke programmeren, is dit boek ontstaan. Geen enkele taal is geheel gelijk voor de diverse besturingssystemen en hardware en daarom is het van groot belang dat men de overeenkomende structuren leert kennen waardoor het overzetten van een computerprogramma van de ene naar de andere programmeertaal veel en veel makkelijker wordt. Ik programmeer nu al 20 jaar vanaf een Burroughs tot de laatste AS/400 en van COBOL tot FORTH tot ASP. De laatste 10 jaar heb ik regelmatig aan nieuwe programmeurs hetzelfde leerzame verhaal verteld zodat ik het tijd vond worden deze verhalen aan het papier toe te vertrouwen en ze zo aan een breder publiek ter beschikking te stellen.

Uiteraard heeft dit boek een bepaalde uitstraling en zit er een bepaalde verwachting in. Ik ga er vanuit dat elke beginnende programmeur zichzelf als doel heeft gesteld het beroemdste programma van de wereld te schrijven. Helaas, dat lukt er maar een paar. Wil je tóch een hele goede baan krijgen met een goede verdienste, dan zijn er wel enige doelen die je jezelf kan stellen. Een programmeur moet iemand zijn die begrijpt wat hij of zij met een computer aan het doen is en kan doen, en vooral wat die ánder, waarvoor een programma gemaakt wordt, er mee wil gaan doen. Een programmeur moet ook zijn of haar verantwoordelijkheden durven nemen. Een programma kan niet zomaar half af opgeleverd worden omdat de vacantie is aangebroken.

Een programmeur moet ook de rode draad door een verzameling van programma's kunnen

blijven volgen of juist kunnen omleggen. Hij of zij hoeft echt niet elk stuk hardware van binnen te kennen, echter wèl de basisbegrippen die aan de werking ervan ten grondslag liggen. Een programmeur lost geen fouten op door ergens anders nieuwe te creëren: 'het ene gat dichten door een ander gat te graven' wordt vaak gehoord. Een programmeur documenteert van te voren de geplande wijzigingen en corrigeert deze documentie achteraf op de een of andere manier, zodat die ook voor derden toegankelijk èn begrijpelijk blijft. Ook nog na een half jaar.

Sinds 1970 is door velen getracht allerlei soorten ‘standaarden’ door te drukken zodat communicatie tussen verschillende soorten mensen en mensen in verschillende disciplines makkelijker zou kunnen verlopen. Er zijn standaarden die meer gericht zijn op het intikken van code dan op het oplossen van problemen, er zijn standaarden die gericht zijn op terminals die slechts 80 goed leesbare tekens in de breedte weer kunnen geven, en gelukkig zijn er ook nog een paar standaarden waarmee goed ontworpen en geprogrammeerd kan worden. Alle hebben ze hun voordelen en beperkingen. Ik ben er echter vanuit gegaan dat we ons tegenwoordig niet zoveel zorgen hoeven te maken over wat wel en niet op het beeldscherm past en dat we met zo min mogelijk tikwerk een, ook voor anderen, overzichtelijke broncode krijgen, waarbij de vrijheid van de programmeur niet in het geding is.

Ik wil met dit boek daarom een bijdrage leveren aan het versoepelen van de communicatie tussen de programmeur(s) en de uiteindelijke gebruiker(s) door te trachten de lezer een handvat te bieden bij het leesbaar en overzichtelijk kunnen opbouwen van functionele ontwerpen en broncode van programmeertalen zoals COBOL, SL, C, BASIC, RPG, DCL, CL, HTML, Java, etc.. zonder een standaard op te willen dringen. De achterliggende gedachte bij dit boek is dat je je in je denken niet moet beperken tot één programmeertaal of computersysteem.

Elke programmeertaal kent zo zijn eigen specialiteiten waarmee optimaal gebruik kan worden gemaakt van het besturingssysteem. Nadeel is dat deze talen vaak producent gebonden, dus niet algemeen gangbaar, zijn. Wordt er gebruik gemaakt van deze specifieke eigenschappen, dan wordt de eventuele overdraagbaarheid van het programma naar een andere programmeertaal er uit gehaald. In de voorbeelden in dit boek is daarom ook regelmatig te zien hoe een bepaald probleem in verschillende talen opgelost moet of kan worden. Op die wijze worden de overeenkomsten tussen diverse talen ook duidelijk. Een programmeur die gewend is op verschillende computersystemen, *cross-platform*, te werken, zal vanzelf zich een manier van programmeren aan te leren die op alle verschillende systemen en in alle talen die hij kent van toegepast kan worden. Dat betekent wel dat er dus nooit voor de volle 100% van een taal gebruik gemaakt kan worden, omdat altijd in het achterhoofd rekening gehouden wordt met de mogelijke conversie van het programma naar een ander systeem, bij een andere klant, in een andere taal.

Soms is het een zwaar beroep, maar ik heb me werkelijk nog nooit verveeld. Ik wens een ieder die dit boek leest heel veel programmeer plezier. En pas er voor op dat je partner geen 'computerweduwe' wordt.

2002,
Marc H.E. Vos

1. PROGRAMMEERTALEN

Waarom zijn er eigenlijk programmeertalen? En waarom zijn er zoveel? Om de eerste vraag te beantwoorden: deze zijn er om de mens op een begrijpelijke manier een computer iets te laten doen. In den beginne moest je een computer één en één bij elkaar op laten tellen door allerlei mechanische schakelaars handmatig in de juiste stand te zetten. Aan of uit, dus. Nul of één. En zo ontstond het binaire cijfer (*Binary Digit = bit*)! Terwijl de technologie met rasse schreden voorwaarts ging, en nog steeds gaat, werden de mechanische schakelaars electronische schakelaars en kon je door het invoeren via een toetsenbord, met de uitvoer naar een printer want beeldschermen had je nog niet in die tijd, de schakelaars aan (1) en uit (0) zetten. Deze computers werden zeer warm van dat gereken en hadden een koelinstallatie nodig die samen met de computer een ruime gymzaal in beslag nam. Tegenwoordig zijn deze aan/uit (*flip/flop*) schakelaars zo klein en electronisch gemaakt, dat je ze met het blote oog alleen via electronenmicroscopen kan waarnemen. Zo klein passen er tenminste een heleboel op een paar vierkante centimeter zand.

Eerste Generatie Talen

Deze enen en nullen noemen we maar de Eerste Generatie Talen, 1GL's (*First Generation Language*). Een hele serie van enen en nullen achter elkaar heet een instructie. Een reeks instructies achter elkaar vormt een programma. En verderop in het boek zul je opmerken dat elke computer nog steeds volgens deze methode werkt, omdat het aan en uitzetten van schakelaars direct gerelateerd is aan de processoren. De taal wordt niet meer zo direct ingevoerd, op een paar uitzonderingen na. Nu zul je wel zeggen dat die enen en nullen geen taal vormen, maar bedenk wel: het spijkerschrift zag er ook niet uit maar was toch ook een taal. Het is de logische opvolger van het handmatig schakelen van de computer. Aangezien enen en nullen intikken een zeer complexe manier van programmeren is, het visuele onderscheid tussen de woorden marginaal is waardoor de foutkans erg groot wordt en er ook geen handleiding voor te maken is, werden de assembleertalen (*assembler*) uitgevonden. Nederlandse terminologie gebruiken we het liefst zo weinig mogelijk, dat werkt op den duur toch maar tegen je. We zijn tenslotte wereldburgers en de Nederlandse taal is niet echt gangbaar binnen de automatiseringswereld.

Tweede Generatie Talen

Het mooie van assembler was dat je door het intikken van, weliswaar cryptische, engelse tekst de computer meerdere 1GL instructies kon laten uitvoeren! Dit werden de tweede generatie talen, 2GL (*Second Generation Language*), genoemd. Een programmeur kon deze instructies ook iets makkelijker onthouden. Hier onder een voorbeeld van assembler voor de Z80 processor (een antieke):

```
PUSH AF
INSTAT: IN A,(0DFH)
BIT 0,A
JR Z,INSTAT
POP AF
```

```
OUT (0DEH),A
RET
```

Voorbeeld 1: Z80 assembleert taal

Wat dit doet is niet interessant. Zoals je ziet is deze taal toch ook nog niet ècht makkelijk. En dat er nog steeds heel veel zo wordt geprogrammeerd kan je bedenken als je je computer open maakt een eens stil staat bij wat je allemaal ziet en dat dat allemaal werkt en doet wat het moet doen.

Wat met deze talen al wel was bereikt was dat op het niveau van de taal een redelijke mate van overeenstemming was bereikt over de in de taal op te nemen instructies! Waarom was dat nodig?

Een assembleertaal is in ruime mate geschreven voor de processor. Er zijn overeenkomende instructies tussen de talen voor verschillende processoren, maar een programma in assembleertaal was en blijft computer gebonden. Als de instructies qua schrijfwijze en werking overeenkomen, is een taal makkelijker te onthouden. Omdat een computer intern nog altijd werkte en werkt met enen en nullen moesten deze instructies eerst daar naar worden. Dat gebeurt door middel van een vertaler (*compiler*).

Een programmeur was in die tijd veel meer hardware gericht bezig en moest meerdere assembleertalen uit zijn hoofd leren. En dan kan je wel eens in de war raken. Uit die tijd stamt ook het eeuwige communicatieprobleem tussen programmeurs en gebruikers. Een programmeur was met zijn gedachten veel meer bezig te bedenken hoe hij de wensen van de gebruiker door de hardware kon laten uitvoeren. Die dacht helemaal niet aan lay-out en gebruikersgemak omdat hij dat zelf niet had. Intussen ging het onderzoek naar het programmeren van computers via taal gestaag door.

Derde Generatie Talen

En zo ontstonden vanzelf de derde generatie talen, 3GL (*Third Generation Language*), die ook wel Hogere Programmeertalen (*High Level Languages = HLL*) worden genoemd. Om er een paar te noemen:

COBOL (Common Business Oriented Language)

- Voor zakelijke, administratieve applicaties.
- Voor bijna elke computer is er een COBOL compiler te koop.

Synergy Language (formerly DEC DIBOL)

- Voor zakelijke, administratieve applicaties.
- Synergy Language heette voorheen DIBOL en was van DECen is nu eigendom van Synergex.
- Synergy Language is beschikbaar voor VAX/VMS, UNIX en Microsoft Windows.

RPG and ILE/RPG (Report Program Generator)

- Voor zakelijke, administratieve applicaties.
- RPG/400 en ILE/RPG compilers zijn alleen op een IBM AS/400 te krijgen.
- De visuele ontwikkelomgeving 'VisualAge for RPG' van IBM is alleen beschikbaar voor de Microsoft Windows omgeving.

BASIC (Beginners All-purpose Symbolic Instruction Code)

- Een taal voor ‘iedereen’.
- Voor ELKE computer is er wel een BASIC compiler te koop.
- Bij sommige versies kun je ook in 2GL programmeren, zodat je dingen kunt doen waarin de taal niet voorziet.
- Er zijn nu visuele ontwikkelomgevingen te krijgen voor Apple MacOS (Xojo) en Microsoft Windows (VisualBASIC en Xojo).

VBScript (Een BASIC variant van Microsoft)

- Speciaal voor het internet ontworpen ter uitbreiding van HTML.
- De VBScript interpreter is ingebakken in het blader-programma Microsoft Internet Explorer.
- De VBScript interpreter is ook ingebakken in Microsoft Windows zelf en is aanweizg als de Scripting Host.

C, C++

- Een veelgebruikte taal voor technische toepassingen zoals besturingssystemen en stuurprogramma's.
- Voor ELKE computer is er een C compiler te vinden.
- Voor bijna elk besturingssysteem is er een **C++** (objectgeoriënteerd C) compiler te koop.

Pascal
- Een veelgebruikte taal voor allerlei toepassingen.
- Voor bijna elk besturingssysteem is er een Pascal compiler te koop.

Java
- Een soort C++
- Speciaal voor het internet ter vervanging van het trage interpreteren van HTML en het mogelijk maken van besturingssysteem-onafhankelijke programma's.
- Voor bijna elk besturingssysteem is er een Java compiler beschikbaar. Je kan deze op het Internet vinden.

JavaScript

- Speciaal voor het internet ontworpen om meer te kunnen doen met webpagina's.
- De Javascript interpreter is ingebakken in alle webbrowsers.
- De JavaScript interpreter is ook ingebakken in Microsoft Windows zelf en is aanwezig als de Scripting Host (**JScript**).
- Een webpagina die met JavaScript (of een andere script-taal) de inhoud van die webpagina regelt, noemt men het zogenaamde Dynamic HTML, ofwel DHTML.

AppleScript (Automator)

- Een programmeertaal om Mac OS en Mac OS applicaties te besturen.
- Beschikbaar vanaf Mac OS 7.

- De taal wordt ook gebruikt om programma's te besturen. Dit zijn 'scriptable' programma's die als 'dictionary' fungeren voor AppleScript.

HTML (HyperText Markup Language)

- Een opmaaktaal voor web pagina's.
- De HTML interpreter bevindt zich in alle webbrowsers.

PostScript
- Een opmaaktaal voor printers.
- Wordt veel gebruikt voor drukwerk en PDF (Portable Document Format) bestanden.
- De interpreters bevinden zich in de printer, de belichter of in aparte afdruk- en/of weergave programma's.

Zo zijn er nog honderden andere talen te noemen. Het grote verschil met de tweede generatie talen is dat het niveau van de taal is opgekrikt en dat de hardware verschillen naar de compiler zijn verplaatst. En de derde generatie talen zijn een goede poging om een programmeertaal dichter bij een spreektaal te brengen. We spreken nu ook niet meer over instructies, maar over *statements*. Een 3GL statement bestaat uit meerdere 2GL instructies welke weer opgebouwd zijn uit meerdere 1GL instructies. Echter, de stap naar 2GL wordt overgeslagen: de compiler zet 3GL direct om in 1GL.

De belangrijkste reden voor de ontwikkeling om de programmeertalen meer en meer op spreektalen te laten lijken is het cryptische en hardware afhankelijke 2GL en 1GL. Spreektalen (Engels, Spaans, Duits, Frans, Nederlands, etc.) bijvoorbeeld, kunnen door meerdere mensen worden gebezigd en toch vereist elk van die mensen een andere manier van benaderen als je ze iets wilt laten doen. Dat geldt ook voor computers. Met een 3GL hoeft een programmeur zich niet meer te bekommeren over de in- en uitvoer van en naar apparaten zoals beeldschermen, harde schijven, printers en scanners die met de computer zijn verbonden. Dat gaat allemaal via softwarematige interfaces (*Application Program Interface = API*) die door andere programmeurs zijn gemaakt en vindt onder andere plaats op het niveau van het besturingssysteem. En met de hoeveelheid mensen die met computers bezig zijn, ontstaan er bijna diezelfde hoeveelheid programmeertalen die allen voor dezelfde of juist specifieke doelen zijn ontworpen. Er zijn nu bijna geen 3GL talen meer die aan één computer gebonden zijn, en dat is fijn; ze zijn nu besturingssysteem-gebonden. Dat betekent dat je op papier een heel software pakket zou kunnen programmeren in een hogere programmeertaal en dat je de klant er het besturingssysteem en de hardware bij laat kiezen. Om een voorbeeld te geven: C is beschikbaar op Compac/VMS, IBM's OS/400, Microsoft Windows, Apple's MacOS, etc.. Maar Visual Basic is er alleen voor Microsoft Windows en kan ook alleen programma's genereren voor dat besturingssysteem, terwijl REALBasic er alleen is voor de Apple Macintosh maar de gebouwde applicatie ook kan genereren voor Microsoft Windows.

Omdat derde generatie talen oorspronkelijk in Engelstalige landen zijn ontwikkeld, lijken ze dan ook erg op een engelse taal. Ze lezen als een bijna gewone, voor ons Nederlanders buitenlandse, taal. Er werd er ook al veel gesproken over 4GL. 4GL moet een niveau hoger zijn dan 3GL en houdt daardoor in dat dát type talen gelijk zou moeten zijn aan de door ons gebezigde schrijftalen. Er zijn 3GL’s die al erg ver gaan zoals COBOL en AppleScript. Daarmee kan je hele zinnen in tikken die al aardig op een natuurlijke taal gaan lijken. Maar je tikt je suf als programmeur en dat wil je nou net niet; daarom is 3GL het nog steeds helemaal,

gewoonweg omdat dat het dichtst bij ons voorstellingsvermogen ligt, snel te onthouden, snel te leren en te begrijpen is. De discussie over 4GL, en voor sommigen die geen onderscheidingsvermogen hadden zelfs 5GL, is nu gelukkig voorbij. Want in diezelfde tijd dat de 3GL tot ontwikkeling kwam, ontstond de discussie over hoe een source er van binnen uit zou moeten zien zodat die door willekeurig welke programmeur begrepen zou kunnen worden. Een illusie, maar ja, je hebt ze erbij. Om daarvoor te zorgen en het tikken van statements voor een groot deel weg te nemen, ontstonden ontwikkelomgevingen, 4GL-Tools en CASE-tools (*Computer Aided Software Engineering*) genoemd, waarbij op een conceptueel niveau 3GL programma's werden gemaakt. Deze gereedschappen genereren 3GL broncode (*sourcecode*) die altijd dezelfde opbouw heeft en waar een programmeur eigenlijk niet meer naar hoeft te kijken. Deze gereedschappen waren bij uitstek geschikt om kaartenbak programma's te maken zoals een ledenadministratie voor een tennisclubje. Voor de, meestal niet eens zo, ingewikkelde zaken, moest je toch nog met de hand programmeren. De doelstellingen waren echter juist. Minder bezig zijn met statements tikken. Uiteindelijk zijn er een paar uitgegroeid tot zware ontwikkelomgevingen, waarmee grote administratieve systemen te bouwen zijn.

Object Georiënteerd programmeren

Tegelijkertijd met de ontwikkeling van deze 4GL- en CASE-tools ontstond het begrip 'object'. Een object was opeens een 'iets' dat iets doet wat je programma nodig heeft en waar de programmeur zich niet hoefde te bemoeien met het ontstaan ervan, alleen met de werking ervan. De term 'object georiënteerd programmeren' was ontstaan. Nu deden en doen zichzelf respecterende programmeurs dat natuurlijk jaren, alleen toen pas wisten ze wat ze eigenlijk aan het doen waren. Hele subroutinebibliotheken zijn en worden er nog steeds gebouwd waarvan de subroutines nu onder de term object kunnen vallen. Je programmeerde een subroutine die een knop ergens op een scherm plaatst en je noemde deze routine KNOP. De inhoud van de parameters bepaalde waar en hoe. Tegenwoordig krijg je het object KNOP, plaats je het ergens op het scherm en vul de eigenschappen van KNOP in. Subroutines hebben parameters en objecten hebben eigenschappen. Op de keper beschouwd zijn dat ook gewoon parameters. Objecten zijn dingen als vensters, knoppen, menubalken, paden naar gegevensobjecten, het afdrukken van een pakbon, etc..

Visuele ontwikkelomgevingen

Uit het OO-concept en de CASE- en 4GL-tools zijn de ‘visuele’ ontwikkelomgevingen ontstaan. Onder een visuele ontwikkelomgeving wordt verstaan dat je direct ziet wat je maakt en hoe het zich gedraagt, met nog steeds de mogelijkheid tot het intikken van sourcecode, zonder dat er merkbaar sourcecode wordt gegenereerd. De compiler maakt een integraal onderdeel uit van de ontwikkelomgeving. Het bulkt er van de objecten, van tekstvelden tot hele internet bladeraars (*browser*).

Eigenlijk bestaan er al lang visuele ontwikkelomgevingen. De terminologie was anders, ze gingen namelijk over de toonbank als gereedschappen waarmee je een prototype van je scherm kon maken (*prototyping*). Alle oude en nieuwe programma's die gemaakt zijn om bijvoorbeeld in- en uitvoerschermen te ontwerpen kunnen nu ook onder de categorie 'visueel'

vallen, bijvoorbeeld SDA (Screen Design Aid) op IBM's AS/400. Je zou kunnen zeggen dat de zoektocht naar een ideale ontwikkelomgeving nu eindelijk ten einde is. Visueel ontwikkelen en gewoon 3GL tikken, dát is het helemaal. Nu zijn we alleen als visueel-programmeurs overgeleverd aan de weidse blik of de oogkleppen van de ontwikkelaar van de visuele ontwikkelomgeving.

Dit type ontwikkelomgevingen is vaak makkelijk te herkennen, want de namen van de meeste van deze 'visuele OO' omgevingen of gereedschappen beginnen met Visual.... (VisualBasic, VisualC++, VisualJavascript, VisualAge, etc.), hoewel er ook andere zijn met een gewone naam zoals RealBasic, Delphi en Synon. Voor bijna alle van de eerder genoemde talen zijn er van deze visuele omgevingen te krijgen.

Compilers en interpreters

Om de statements in een source om te zetten naar iets waar een processor mee overweg kan heb je een compiler en een koppelaar (*linker*) nodig. De compiler zet die ene 3GL regel om in die honderden 01001001 instructies en de linker knoopt deze omgezette sourcecode (*objectcode*) met specifieke objecten voor communicatie met het besturingssysteem (*operating system = OS*) aan elkaar tot een uitvoerbaar programma voor een bepaald besturingssysteem. Als Microsoft het programma Word compileert en linkt voor het besturingssysteem Windows, dan werkt dat programma niet onder het besturingssysteem MacOS. Daarvoor moet het gelinkt worden met objecten die het MacOS aansturen.

Een gecompileerd en gelinkt programma dat je zó kunt uitdelen, met de enige beperking dat het besturingssysteem gelijk of van een hogere versie moet zijn, noemen we ‘op zich zelf staande’ (*standalone*) programma’s. Er zijn ook compilers en linkers die geen standalone programma’s produceren. Deze compileren en linken de sourcecode tot een object welke door een ánder programma, en dat is niet het besturingssysteem, uitgevoerd moeten worden. Dat ándere programma noemen ze een ‘*runtime*’. In een runtime zijn alle functies aanwezig die normaal gesproken door de linker direct aan je programma geknoopt zouden worden. Als je een programma dat voor een runtime is gecompileerd uit wilt delen, moet je de betreffende runtime ook meeleveren. En in de documentatie die bij de programmeertaal wordt geleverd staat of dat mag en of het gratis mag. Een voordeel van het gebruik van een runtime-programma is dat het minder schijfruimte inneemt omdat de rest van de code in de runtime zit geprogrammeerd. En schijfruimte was in de beginjaren verschrikkelijk duur. Voor een 10 megabyte harde schijf betaalde je rond 1970 ongeveer tienduizend gulden (duizend gulden per Mb), rond 1990 was dat nog maar duizend gulden (honderd gulden per Mb) en nu liggen ze bij bosjes in de vuilcontainer en bij de kringloopwinkels.

Dan is er nog een categorie programma’s die sourcecode helemaal niet compileert maar interpreteert en controleert op het moment dat de sourcecode wordt aangeboden. De bekendste interpreter uit de oudheid is de BASIC interpreter. De modernste interpreters zijn de internet browsers zoals Microsoft Internet Explorer, Netscape Navigator, iCab, Opera, Mozilla, etc.. Deze interpreteren HTML en Javascript en sommige ook VBScript. Ook in printers en belichters zitten interpreters die printertalen zoals Postscript verwerken. Omdat interpreters traag zijn, zijn de compilers uitgevonden. En de nieuwste compilers die opgang maken zijn JIT (*Just In Time*) compilers. Deze bufferen de te interpreteren sourcecode totdat de hele source is ontvangen en compileren die source dan tot een uitvoerbare routine die door

de oorspronkelijke ontvanger kan worden uitgevoerd. Dit gebeurt op grote schaal bij de Javascripts die in internet pagina's zijn geprogrammeerd. Het kost een internetbrowser veel te veel capaciteit om naast HTML ook nog eens een volwaardige Javascript interpreter te zijn. Wat nu gebeurt is dat de volledige Javascript sourcecode aan de JIT-compiler wordt aangeboden die het razendsnel compileert en linkt naar voor het besturingssysteem uitvoerbare code, welke code vervolgens door de browser geactiveerd wordt.

Maar niet alles is te compileren. HTML bijvoorbeeld is eigenlijk een opmaaktaal die volledig dynamisch te genereren is vanuit Javascript of VBScript. Een standalone HTML-programma kan daarom niet bestaan en zou ook zinloos zijn vanwege de structuur van de taal: bovenin de source beginnen en onderaan eindigen met interpreteren en dat is het dan. Meer niet. Geen lussen en geen condities.

Een compiler is ook maar gewoon een programma, maar wel een hele slimme, geschreven in een 2GL of 3GL. En met de aanschaf van een computer, groot of klein, heb je alleen de taal waarmee je het besturingssysteem kunt besturen en marginale programma's mee kunt maken tot je beschikking:

MacOS AppleScript
Windows Scripting Host / DOS
OS/400 CL (Command Language)
VMS DCL (Digital Command Language)
UNIX sh (shell)

Net zoals je je tekstverwerkingssoftware los van de computer koopt, koop je ook een ontwikkelomgeving apart van de computer. Wanneer je dus zelf programma's wilt maken, kan je zelf kiezen in welke taal je dat wilt doen. Tenzij de klant of het bedrijf waarvoor je gaat programmeren anders voorschrijft omdat er daar al voor een bepaalde omgeving is gekozen. Je koopt bijvoorbeeld een BASIC-, C- of COBOL-omgeving voor je computer. Of misschien wel allemaal, als je dat beter uitkomt.

Voorbeelden

Hieronder en op de volgende bladzijden volgen wat voorbeelden in diverse derde generatie talen die allen hetzelfde doen, namelijk de schrikkeljaar test. Als je een Apple Macintosh hebt, met Systeem 7 of hoger, heb je de beschikking over AppleScript en kan je het AppleScript voorbeeld intoetsen. Met Windows98 en hoger heb je de Scripting Host. Toets het VBScript programmaatje in, voeg er een 'msgBox' aan toe en bewaar het document met .VBS als extensie. De voorbeelden zijn heel ruim opgezet en kunnen in sommige gevallen misschien wel wat efficiënter, maar alleen op deze manier zie je de overeenkomsten tussen de talen en hun grammatica (*syntax*), wat uiteindelijk de opzet is van dit boek. Op deze wijze zal je door het hele boek voorbeelden tegenkomen:

```
schrik = 0;

if ((((int) (eejj / 100) * 100) != eejj) &&
    (((int) (eejj /   4) *   4) == eejj)) {
        schrik = 1;                              /* Jaar */
} else if (((int) (eejj / 400) * 400) != eejj) {
        schrik = 1;                              /* Eeuw */
```

```
}
```

Voorbeeld 2: Schrikkeljaartest in de taal C

```
schrik = 0

if (((int(eejj / 100) * 100) <> eejj) and _
    ((int(eejj /   4) *   4) = eejj)) then
        schrik = 1                              ' Jaar
elseif ((int(eejj / 400) * 400) = eejj) then
        schrik = 1                              ' Eeuw
end if
```

Voorbeeld 3: Schrikkeljaartest in VBScript

```
schrik = 0

if (((eejj / 100) * 100).ne.eejj .and.
&   ((eejj /   4) *   4).eq.eejj) then
        schrik = 1                              ; Jaar
else if ((eejj / 400) * 400).eq.eejj
        schrik = 1                              ; Eeuw
```

Voorbeeld 4: Schrikkeljaartest in de taal Synergy Language

set schrik **to** 0

if ((round (eejj / 100) rounding down) * 100) **is not equal to** eejj and ((round (eejj / 4) rounding to nearest) * 4) is equal to eejj **then**
        set schrik **to** 1
else if ((round (eejj / 400) rounding to nearest) * 400) is equal to eejj **then**
        set schrik **to** 1
end if

Voorbeeld 5: Schrikkeljaartest in AppleScript

```
        Z-ADD   *ZERO   SCHRIK  10

EEJJ    DIV     100     EE1     40
        MULT    100     EE1
EEJJ    DIV     4       JJ      40
        MULT    4       JJ
EEJJ    DIV     400     EE2     40
        MULT    400     EE2

        SELEC
EE1     WHNE    EEJJ
JJ      ANDEQ   EEJJ
        Z-ADD   1       SCHRIK
EE2     WHEQ    EEJJ
        Z-ADD   1       SCHRIK
        ENDSL
```

Voorbeeld 6: Schrikkeljaartest in de taal RPG/400

```
<script language="JavaScript">

function compute(form) {
        eejj = form.year.value;

        if (((parseInt(eejj / 100) * 100) != eejj) &&
```

```
            ((parseInt(eejj /   4) *   4) == eejj))
               form.result.value = "SCHRIKKELJAAR!";
        else if ((parseInt(eejj / 400) * 400) == eejj)
               form.result.value = "SCHRIKKELJAAR!";
        else   form.result.value = "GEEN SCHRIKKELJAAR!";
}

</script>
```

Voorbeeld 7: Schrikkeljaartest in JavaScript

Zoals in bovenstaande voorbeelden goed is te zien, zijn er overeenkomstige structuren te herkennen in de verschillende talen. Deze herkenning is de basis voor het meertalig kunnen programmeren.

Een opmerking over de taal Synergy Language. Deze taal heette voorheen DIBOL en was de huistaal van DEC (Digital Equipemnt Corporation). DEC bestaat ondertussen niet meer en heeft indertijd de taal DIBOL overgedaan aan een concurrent die al jaren een variant van DIBOL, geheten DBL, onderhield. Dit bedrijf heet nu Synergex en DIBOL/DBL heet nu Synergy Language. De taalnaam korten we in het boek af tot SL.

2. PROGRAMMEREN

Wat is er nou zo leuk aan programmeren, dat er mensen zijn die het liefst de hele dag met de computer doorbrengen en computerprogramma's maken? Is het omdat het zo'n verschrikkelijke kick geeft een apparaat iets te laten doen wat jij graag wilt dat het doet? Is het omdat het zo'n verschrikkelijke kick geeft dat je een bedrijf helpt de administratie te vereenvoudigen? Is het omdat het zo'n verschrikkelijke kick geeft dat de hele wereld jouw computerspel speelt? Of is het gewoonweg omdat er zoveel verschillende leuke programma's te maken zijn? Van programma's die modeltreinen en kerncentrales besturen tot programma's waarmee je je boekhouding en je CD-verzameling kan administreren. Er is zoveel leuks te programmeren.

Wat moet een toekomstige programmeur dan wel niet weten en kunnen? Het allerbelangrijkste is wel een goed inzicht hebben in oorzaak en gevolg. Iemand die dat goed kan, kan vaak ook andersom goed redeneren: van gevolg naar de oorzaak. En daar draait het voornamelijk om bij het bouwen van programma's. Een 'gevolg' komt overeen met een ontwerp of idee over het eindresultaat en de 'oorzaken' zijn één of meer stappen om tot het eindresultaat te komen. Daar uit volgt dat logisch na kunnen denken, dus het stapsgewijs kunnen analyseren en kunnen beredeneren van de gang van oorzaak naar gevolg, ook een belangrijke eigenschap van een programmeur moet zijn. De laatste eigenschap is de wil: de wil om het voor elkaar te krijgen, zonder te vloeken en te schelden; daarmee laat je alleen maar zien hoeveel fouten je maakt.

De vragen die een programmeur zichzelf altijd moet stellen zijn de volgende drie:

- Waarom ga ik het doen?
- Hoe werkt het besturingssysteem waarop ik ga programmeren?
- Ben ik bekend met de in gebruik zijnde terminologieen?

Vervolgens ga moet er een programmeertaal gekozen worden die geschikt lijkt voor wat er gemaakt moet worden en waarvan er een compiler voor het besturingssysteem te krijgen is. Wat heb je nodig om in een 3GL te gaan programmeren? Een programma waarmee platte tekstbestanden kunnen worden gemaakt zoals Notepad of BBEdit, want dat is wat een compiler kan inlezen. Een compiler begrijpt niets van vette of cursieve tekst of van een standaard Excel of Ragtime document. Om er voor te zorgen dat een programmeur hier niet de fout mee ingaat, zit er bij de compiler ook een tekstverwerker, eventueel ingebouwd, waarmee veilig de sourcecode in kan worden getikt, de *source editor*. Als er naast een compiler allerlei ontwikkelgereedschappen worden bijgeleverd wordt dat een ontwikkelomgeving genoemd (*SDE: Software Development Environment of SDK: Software Development Kit*). Om met een SDK software te kunnen ontwikkelen is er bij de toekomstige programmeur een bepaalde basiskennis nodig over zaken die in elke programmeertaal worden gebruikt.

ROM en RAM

ROM staat voor *Read Only Memory* en daar is bij de fabricage van het moederbord

software in gebrand die er voor zorgt dat de hardware functioneert zonder dat het besturingssysteem geladen hoeft te zijn. Zonder dat zou een computer nooit van een schijf op kunnen starten. ROM kan alleen gelezen worden, er kan niets aan worden veranderd. Er bestaan echter ook PROM's, *Programmable Read Only Memory*, waarvan de opgeslagen software in zijn geheel vervangen moet worden om wijzigingen door te voeren. De afkorting RAM staat voor Random Access Memory wat al zegt dat het geheugen willekeurig toegankelijk is. Er kan uit gelezen worden, maar er kan ook naar geschreven worden zonder dat hele stukken eerst verwijderd moeten worden. De hoeveelheden ROM en RAM worden uitgerdukt in bytes.

Een computer laadt het ROM in het RAM en pas dan kan de computer ècht opstarten. Dat betekent dat er aan het geactiveerde ROM, wat nu in RAM draait, wel dingen kunnen worden veranderd; net zolang tot de computer crasht. Dan start het ding weer met een schone lei op en kan er opnieuw begonnen worden met expirimenteren.

Van een programma dat wordt gestart, wordt een groot deel in een eigen stuk RAM geladen en door het besturingssysteem verwerkt. Een programma heeft RAM nodig voor zichzelf en extra RAM voor de bestanden die het programma moet openen. Notepad of Simple Text hebben weinig RAM nodig, Photoshop daarentegen heeft eigenlijk nooit genoeg. Een programma bestaat eigenlijk uit niets meer of minder dan een reeks instructies voor de verschillende processoren in de computer. Een programma dat actief is, moet bepaalde dingen 'onthouden' zolang het actief is, bijvoorbeeld welke bestanden de gebruiker heeft geopend. Dat 'onthouden' vindt plaats door middel van zogenaamde *variabelen*.

Variabele

Het woord geeft het al een beetje aan. Bij de programmeertalen is het een begrensde ruimte in het RAM die met een naam te benaderen is en waarvan de inhoud veranderd kan worden. Een variabele kan worden gezien als een luciferdoosje waarin initieel lucifers zitten, maar waar na verloop van tijd, door verschillende oorzaken, de lucifers plaats maken voor andere dingen, bijvoorbeeld mooie steentjes. Een andere, al genoemde, analogie met programmavariabelen is dat een luciferdoosje een begrensde ruimte is en er nooit grotere stenen in zullen passen. Daar is een grotere ruimte voor nodig, wat ook geldt voor variabelen. Een ander aspect van een variabele betreft het type. Er zijn variabelen die tekst of een afbeelding kunnen bevatten en er zijn variabelen die uitsluitend numerieke gegevens kunnen bevatten voor de nodige berekeningen. Een variabele kan in principe alleen gegevens bevatten van zijn eigen soort. Wanneer er een type-conversie plaats moet vinden, zoals de inhoud van een numerieke variabele in een alfanumerieke variabele copiëren, zorgt de compiler meestal voor de omzetting van de ene soort naar de andere.

Alle bewerkingen in een programma, zoals optellen, aftrekken, delen, vermenigvuldigen en vergelijken, hebben betrekking op de inhoud van een variabele. Deze inhoud kan bewerkt, gecopieërd, gewist of bewaard worden. In de voorbeelden van de schrikkeljaarberekening in het vorige hoofdstuk worden minimaal drie variabelen gebruikt, te weten: SCHRIK, EEJJ en FACTOR. Dit zijn numerieke variabelen omdat de gegevens waarmee ze worden gevuld uitsluitend numerieke gegevens zijn. SCHRIK wordt gebruikt om aan te geven dat het jaartal in de variabele EEJJ een schrikkeljaar is. De variabele FACTOR bevat de factor waarmee bepaald wordt of de inhoud van EEJJ een schrikkeljaar is. De variabele SCHRIK had ook een

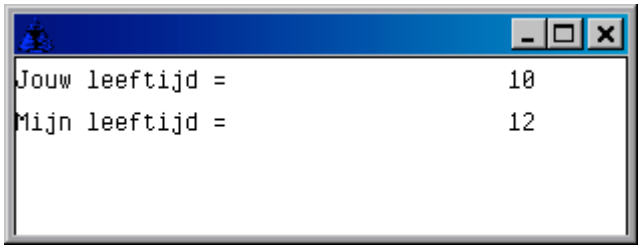
alfanumerieke variabele mogen zijn met 'N' of 'J' in plaats van 0 of 1.

Dan zijn er, naast de verschillende typen, twee manieren om een variabele te definiëren: locaal (*local*) of algemeen (*common, global, external*). Locale variabelen kunnen alleen worden gebruikt in het programma waarin ze worden gedefinieerd. Algemene variabelen kunnen daarentegen ook buiten het programma waar ze zijn gedefinieerd worden gebruikt. Elke taal heeft daar zo z'n regels voor om dat te bewerkstelligen. Het zijn gevaarlijke variabelen waar voorzichtig mee om moet worden gesprongen omdat een wijziging van de inhoud betrekking heeft op alle routines die door het programma worden gebruikt en die deze variabele ook gebruiken. Wanneer bijvoorbeeld een algemene variabele voor een regelnummer op het beeldscherm wordt gebruikt en er zijn verschillende routines die deze positie gebruiken om iets op het beeldscherm te zetten, dan zorgt een verandering van de inhoud ervoor dat al die routines hun gegevens op een andere positie op het beeldscherm plaatsen. En naar zo'n fout kan je soms heel lang zoeken.

Voorbeelden

Hier volgen weer wat voorbeelden in diverse talen over het bewerken, copiëren en uitlezen van variabelen.

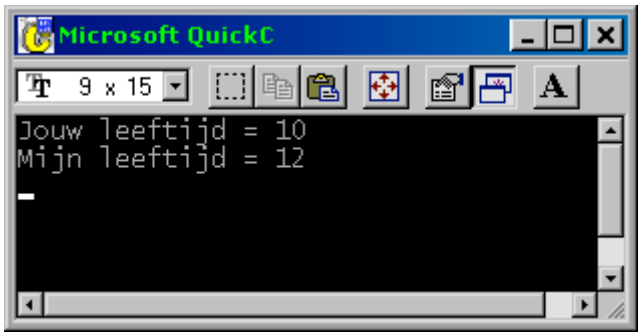
```
record
        alfanum,        a35     ; Alfanumeriek, 35 tekens lang
        numonly,        d2      ; Numeriek, 2 bytes lang.
        numcopy,        d2      ; Idem.

.proc
        open (1, i, 'tt:')

        alfanum = 'Jouw leeftijd = '
        numonly = 10

        display (1, alfanum, %string(numonly), 10, 13)

        alfanum = 'Mijn leeftijd = '
        numcopy = numonly + 2

        display (1, alfanum, %string(numcopy) , 10, 13)

        reads (1, alfanum)
        close 1
.end
```



Voorbeeld 8: Bewerken van variabelen in de taal SL

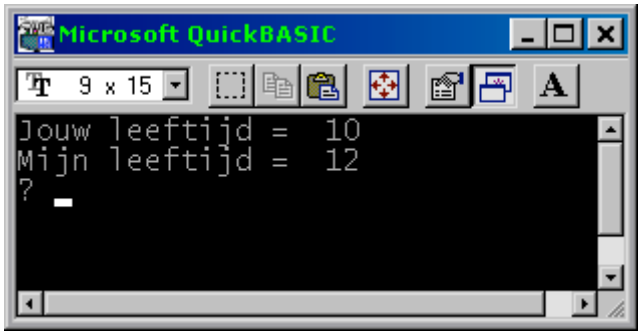
```
#include <stdio.h>
```

```
#include <string.h>

char alfanum[35];               /* Alfanumeriek, 35 tekens lang */
int numonly;            /* Numeriek (minimaal 2 bytes) */
int numcopy;            /* Numeriek (minimaal 2 bytes) */

main() {
        strcpy (alfanum, "Jouw leeftijd = ");
        numonly = 10;

        printf ("%s%d\n", alfanum, numonly);

        strcpy (alfanum, "Mijn leeftijd = ");
        numcopy = numonly + 2;

        printf ("%s%d\n", alfanum, numcopy);

        numonly = (int) getchar();
};
```



Voorbeeld 9: Bewerken van variabelen in de taal C

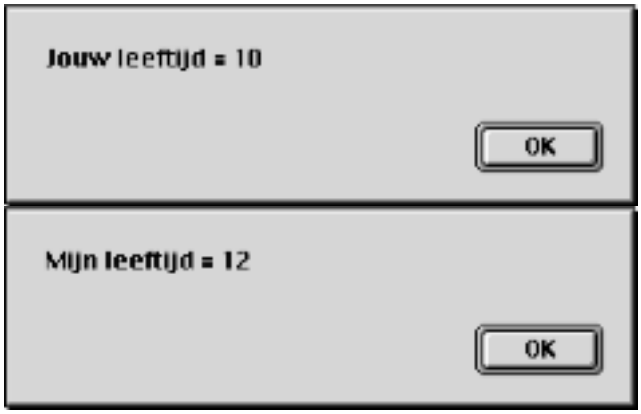
```
alfanum$ = "Jouw leeftijd = "   'Alfanumeriek, variabele lengte
numonly% = 10                   'Integer (minimaal 2 bytes)

PRINT alfanum$;numonly%

alfanum$ = "Mijn leeftijd = "
numcopy% = numonly% + 2

PRINT alfanum$;numcopy%

INPUT alfanum$
END
```



Voorbeeld 10: Bewerken van variabelen in een taalvariant van BASIC

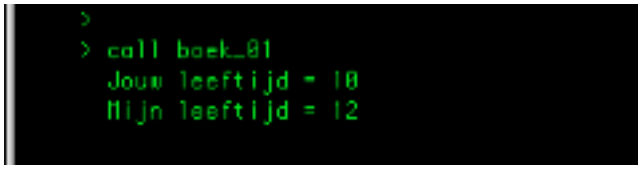
set alfanum **to** "Jouw leeftijd = " *-- Alfanumeriek, variabele lengte*

```
set numonly to 10                    -- Numeriek, variabele grootte

display dialog alfanum & '"' & numonly & '"'

set alfanum to "Mijn leeftijd = "
set numcopy to numonly + 2

display dialog alfanum & '"' & numcopy & '"'
```



Voorbeeld 11: Bewerken van variabelen in de taal AppleScript

```
pgm
        dcl &alfanum *char 35   /* Alfanumeriek 35 tekens */
        dcl &numonly *dec (2 0) /* Numeriek, 2 bytes */
        dcl &numcopy *dec (2 0) /* Numeriek, 2 bytes */
        dcl &toalfa  *char 2    /* Alfanumeriek 2 tekens */

        chgvar &alfanum ('Jouw leeftijd = ')
        chgvar &numonly (10)
        chgvar &toalfa  (&numonly)      /* Type conversie */

        sndpgmmsg msg(&alfanum *tcat ' ' *cat &toalfa) topgmq(*prv)


        chgvar &alfanum ('Mijn leeftijd = ')
        chgvar &numcopy (&numonly + 2)
        chgvar &toalfa  (&numcopy)      /* Type conversie */

        sndpgmmsg msg(&alfanum *tcat ' ' *cat &toalfa topgmq*prv)
endpgm
```



Voorbeeld 12: Bewerken van variabelen in de taal CL (opdrachttaal van OS/400)

```
<html>
<head>
        <title>Jouw leeftijd</title>
</head>

<body bgcolor="FFFFCC">
```

```
<table border=2 width=400>
<tr><td>
        <center>
        <script language="javascript">
        alfanum = "Jouw leeftijd = ";
        numonly = 10;

        document.writeln ("<br>" + alfanum  + numonly + "<p>");

        alfanum = "Mijn leefttijd = ";
        numcopy = numonly + 2;

        document.writeln (alfanum  + numcopy + "<p>");
        </script>
        </center>

</td></tr>
</table>

</body>
</html>
```

Voorbeeld 13: Bewerken van variabelen in HTML met JavaScript

De getoonde voorbeelden laten zien hoe:

de inhoud van een variabele verandert :

```
ALFANUM = "Jouw leeftijd = "
ALFANUM = "Mijn leeftijd = "
```

de inhoud van een variabele wordt gecopieerd:

```
NUMCOPY = NUMONLY                (NUMCOPY wordt ook 10)
```

de inhoud van een variabele wordt bewerkt:

```
NUMCOPY = NUMONLY + 2   (NUMCOPY wordt 10 + 2 = 12)
```

De overige taalelementen (*statements*) zijn ten behoeve van de uitvoer naar het beeldscherm en om er een compleet programma van te maken. Zoals te zien is, lijken de statements in de verschillende talen erg veel op elkaar en is de weergave van de resultaten in elke taal anders. Dat is afhankelijk van de gebruikte ontwikkelomgeving en het gebruikte besturingssysteem waaronder het programma draait. Een ding om rekening mee te houden bij het programmeren is hoe iets in een bepaalde taal moet worden gedaan zodat het later in een

andere taal kan worden nagebouwd. Daarvoor moet je wel de beide talen kennen om de overeenkomsten en de onoverkoombare verschillen te kunnen onderscheiden. Hoe meer ervaring iemand opbouwt in het programmeren met verschillende talen, hoe makkelijker dat gaat.

Constante en letterlijke waarde

(Constant and Literal)

Een constante is een onveranderlijke waarde met een naam. De waarde blijft constant gelijk. Neem bijvoorbeeld de woorden POND en ONS. Deze vertegenwoordigen, als we het over gewicht hebben, beide een waarde die nooit veranderd. Pond staat voor *500 gram*, ons staat voor *100 gram*. Pond en ons zijn daarom constanten. Constanten worden ook gedefinieerd voor waarden die in normaal spraakgebruik niet te hanteren zijn, zoals bijvoorbeeld PI (symbool = π), welke een waarde vertegenwoordigd van *3,141593...* tot en met een tot nu toe nog onbekend aantal decimalen. In een programmeertaal lijkt het op een variabele, de waarde kan echter niet worden veranderd. Er zijn programmeertalen die niet echt onderscheid maken tussen de termen constante en literal. Voor een programmeertaal zijn ze beide onveranderbaar en kunnen daarom als gelijken worden beschouwd en behandeld.

Een letterlijke waarde is de waarde zoals die wordt uitgeschreven. Als men 100 schrijft dan is dat ook gewoon 100 en verder niets. De tekst “Hallo, allemaal” is ook een letterlijke waarde; het is geen naam van een constante, variabele, subroutine, etc. De waarde kan door een programma niet worden aangepast, omdat jij als programmeur het zo hebt ingetypt. Er volgt een voorbeeld, in de programmeertaal BASIC, van de declaratie van een constante en een poging de inhoud van de constante te veranderen:

```
CONST fout1$ = "Er is iets mis..."
fout1$ = "Oeps...dit kan niet"
```

In de eerste regel van bovenstaand voorbeeld is CONST het BASIC-statement voor het definiëren van een constante, FOUT1$ de constante en "Er is iets mis..." de letterlijke waarde. De regel erna zal een foutmelding genereren omdat FOUT1$ als een constante is gedefinieerd en om die reden niet van waarde kan veranderen. De compiler signaleert deze foutsituatie gelukkig al in een vroeg stadium en mocht dat niet zo zijn, dan treed er jammergenoeg tijdens het draaien van het programma een foutsituatie op die waarschijnlijk zorgt voor een stevige crash. Een ander voorbeeld dat de letterlijke waarde illustreert is het volgende, voor iedereen klip en klare, statement:

```
"Er is iets mis..." = "Oeps...dit kan niet"
```

Ook dit resulteert in een foutmelding van de compiler of er ontstaat een foutsituatie tijdens het draaien van het programma waarin dit is geprogrammeerd.

Voorbeelden

Er volgen nu een paar voorbeelden in diverse talen hoe de constanten PI, HALLO en POND gedefinieerd en met een literal gevuld moeten worden:

```
CONST pi# = 3.141593, hallo$ = "Hallo, is er iemand?"
CONST pond% = 500
```

Voorbeeld 14: Constanten en letterlijke waarden in een taalvariant van BASIC

```
.define PI,             3.141593
.define HALLO           'Hallo, is er iemand?'
.define POND            500
```

Voorbeeld 15: Constanten en letterlijke waarden in de taal SL

```
const double pi = 3.141593;
const char hallo[] = {"Hallo, is er iemand?"};
const int pond = 500;
```

Voorbeeld 16: Constanten en letterlijke waarden in de taal C

```
I              3.141593              C         PI
I              'Hallo, is er iemand?'C         HALLO
I              500                   C         POND
```

Voorbeeld 17: Constanten en letterlijke waarden in de taal RPG/400

Constanten zijn handig. Juist als vaak dezelfde letterlijke waarde in een programma moet worden gebruikt, is het verstandig er een constante van te maken. Het voordeel zit'm in de lees- en onderhoudbaarheid van de source. Het is duidelijker 2 * PI dan 2 * 3,141593 te programmeren. Een ander voorbeeld is bijvoorbeeld het adres bovenaan een factuur. De naam, straat of postbus, postcode en land beginnen allemaal op dezelfde horizontale positie, de kolom. Wanneer je nu geen constante gebruikt en de letterlijke waarde met het kolomnummer moet worden veranderd, dan moeten er een of meerdere sources doorzocht worden naar gebruik van de oude letterlijke waarde en deze vervangen door de nieuwe. Definieer je echter een constante, dan is er maar één plaats waar de oude letterlijke waarde vervangen kan worden door de nieuwe.

Subroutine en functie

De bedoeling van het maken van subroutines of functies is het eenmalig programmeren van een regelmatig terugkerende reeks handelingen, zodat deze reeks handelingen in het vervolg kan worden vervangen door de veel kortere code van de aanroep van de externe subroutine. Een externe subroutine of functie kan als object in een zogenaamde subroutinebibliotheek worden bewaard. Zulke subroutinebibliotheken kunnen uitsluitend voor eigen gebruik zijn, maar men kan ze ook verkopen.

Subroutines en functies verschillen qua gedrag en programmering niet veel van elkaar. Ze bestaan beide uit een aantal statements van een bepaalde programmeertaal die gegroepeerd zijn onder een naam. Waar zit dan het essentiële verschil tussen een subroutine en een

functie? Dat verschil uit zich tijdens het gebruik: een functie laat een waarde achter (a = left$(a$, 3)) op de plaats waar hij wordt geprogrammeerd en een subroutine nooit (call moveto (x,y)). Een subroutine wordt zo genoemd omdat het programma dat de subroutine aanroept zelf ‘routine’ wordt genoemd; vandaar het voorvoegsel 'sub'. Een functie heet functie omdat in den beginne wiskundige berekeningen met een computer wilde maken. En de wiskunde zit boordevol functies. Subroutines en functies kunnen waarden ontvangen van de aanroepende routine, door gebruik te maken van aanroep parameters (*call parameters*). En alleen subroutines kunnen ook via deze parameters waarden terug sturen naar de aanroepende routine. De parameter waarin je een waarde terug verwacht, moet een variabele zijn en geen constante of literal.

Doordat een functie een waarde achterlaat, kan je de functie aan de rechterkant van het '=' teken, als parameter en als vervanger voor bepaalde variabelen gebruiken. Een ander voordeel is de verbetering van de leesbaarheid van een source. In het volgende voorbeeld, in de taal SL, is SIZE een externe subroutine (1e voorbeeld) en een functie (2e voorbeeld). In SL wordt een functie-aanroep door een %-teken aangegeven:

```
xcall size (datum, len)
if len.gt.6 begin
        xcall externe_subr (datum)
        ...
        ...
end
```

is identiek aan:

```
if %size(datum).gt.6 begin
        xcall externe_subr (datum)
        ...
        ...
end
```

EXTERNE_SUBR is een subroutine die buiten het programma ligt en apart moet worden gecompileerd en meegelinkt; een externe subroutine. Een externe subroutine of functie heeft de beschikking over eigen variabelen en eigen interne subroutines; het is eigenlijk een gewoon programma, dat echter niet onafhankelijk kan worden gebruikt. In RPG/400 zijn de externe subroutines wèl op zichzelf staande programma's die vanaf de opdrachtregel kunnen worden gestart en kunnen er geen eigen functies worden geprogrammeerd. In C bestaan er geen externe subroutines, maar alleen functies.

Voorbeelden

Ter verduidelijking volgt hieronder een voorbeeld waarbij het nut van een subroutine duidelijk naar voren komt. Er vinden in een programma verschillende tests plaats op allerlei landcoderingen. In het eerste voorbeeld wordt voor elke landcode de test apart geprogrammeerd:

```
regel = 3
kolom = 23
foutindicator = 0

als (afz.land is gevuld) dan
       lees landenbestand met afz.land
       als (leesfout) dan
              doe toon (regel, kolom, '???')
              foutindicator = 1
       anders
              doe toon (regel, kolom, landnaam)
       einde als
einde als

als (ontv.land is gevuld) dan
       lees landenbestand met ontv.land
       als (leesfout) dan
              doe toon (regel+1, kolom, '???')
              foutindicator = 1
       anders
              doe toon (regel+1, kolom, landnaam)
       einde als
einde als
```

De test of een landcode bestaat is in principe altijd dezelfde, het enige dat verschilt is de variabele waarmee het landenbestand wordt gelezen. De ene keer is dat `afzender.land` en de andere keer `ontvanger.land`. Als we die nu van te voren in een algemenere variabele stoppen en die in de test gebruiken waardoor de code in beide gevallen dezelfde wordt, kunnen we de landcode-test afzonderen in een eigen subroutine of functie. Het eindresultaat zie je in het volgende voorbeeld:

```
regel = 3
kolom = 23

als (afz.land is gevuld) dan
       doe test_land (afz.land, regel, kolom, fout)
       als (fout) dan
              ...
              ...
       einde als
einde als

als (ontv.land is gevuld) dan
       doe test_land (ontv.land, regel+1, kolom, fout)
       als (fout) dan
              ...
              ...
       einde als
einde als

...
...

subroutine test_land
parameter     sleutel         ' in: te testen landcode
parameter     y               ' in: regelnummer (y-as)
parameter     x               ' in: kolomnummer (x-as)
parameter     fout            ' uit: foutindicator

       fout = 0
```

```
        lees landenbestand met sleutel
        als (leesfout) dan
                doe toon (y, x, '???')
                fout = 1
        anders
                doe toon (y, x, landnaam)
        einde als
keer_terug
```

De subroutine TEST_LAND is een externe subroutine waaraan je variabelen mee kan geven waarvan de inhoud in de routine wordt gebruikt en gewijzigd. De parameters SLEUTEL, X en Y zijn in dit geval invoer parameters: de subroutine heeft deze nodig om te kunnen functioneren. De parameter FOUT is een uitvoer parameter; middels deze parameter krijgt het aanroepende programma een signaal terug, zodat het kan constateren dat de verwerking in de subroutine goed of fout is afgelopen. Een andere optie is om van de landentests geen subroutine maar een functie te maken. Dan kan bijvoorbeeld de parameter FOUT vervallen en het signaal wordt dan als teruggave waarde gebruikt. Een functie geeft namelijk altijd een waarde terug, daardoor is een functie rechts van het '='-teken te gebruiken:

```
als (afz.land is gevuld) doe
        als (test_land(afz.land, regel, kolom) <> goed) dan
                ...
                ...
        einde als
einde doe

als (ontv.land is gevuld) doe
        als (test_land(ontv.land, regel+1, kolom) <> goed) dan
                ...
                ...
        einde als
einde doe

...
...

functie test_land
parameter       sleutel         ' in: te testen landcode
parameter       Y               ' in: regelnummer
parameter       X               ' in: kolomnummer

        fout = 0

        lees landenbestand met sleutel
        als (leesfout) dan
                doe toon (y, x, '???')
                fout = 1
        anders
                doe toon (y, x, landnaam)
        einde als
keer_terug (met fout)
```

Door de voorbeelden komt duidelijk naar voren hoe een subroutines en functies de herhaling van blokken sourcecode kunnen vervangen en voorkomen, waardoor de source van je programma uiteindelijk korter, en daardoor duidelijker en makkelijker te onderhouden wordt. Tevens wordt het gebruik van parameters geïllustreerd. De waarde `regel+1` zal tijdens het draaien van het programma worden geëvalueerd en het resultaat van die berekening wordt

aan parameter Y doorgegeven. In de routine, die alleen Y kent, wordt deze waarde gebruikt als positiebepaling op het beeldscherm.

De test '<> goed' die wordt uitgevoerd had natuurlijk ook '= fout' kunnen zijn. Echter, dan moet het wel zeker zijn dat 'fout' geen andere waarden kan aannemen dan die funtie terug kan geven. Een situatie waarbij deze manier van vergelijken belang is is bijvoorbeeld bij een J/N veld. De vergelijking '<> goed' is dan alles wat niet 'J' is, terwijl '= fout' de waarde 'N' of ' ' zou kunnen zijn.

Interne subroutine

Naast externe subroutines bestaan er ook interne subroutines. Interne subroutines zijn routines die zich in het programma zelf bevinden, waar je geen parameters aan mee kunt geven en die geen eigen locale variabelen kennen. Alle variabelen die in een interne subroutine worden gebruikt zijn ook buiten deze routine aanwezig. Behalve de eerstgenoemde reden om subroutines te bouwen is de tweede reden het leesbaarder maken van een source door afgebakende handelingen af te zonderen van de overige code, waardoor deze korter en daardoor overzichtelijker wordt. De aanroep van een interne subroutine verschilt ook per taal. In BASIC wordt een interne routine aangeroepen met het overbekende GOSUB (GO to SUBroutine) statement. In RPG is gebeurt dat met het EXSR (EXecute SubRoutine) statement, in COBOL is er het PERFORM statement en in SL het CALL statement.

Hieronder volgt een voorbeeld met subroutine TEST_LAND, maar nu is TEST_LAND een interne subroutine zodat duidelijk te zien is dat zo'n routine geen parameters kent, maar daarentegen gewoon de programmavariabelen `regel` en `kolom` kan gebruiken die buiten de subroutine zijn gedefinieerd:

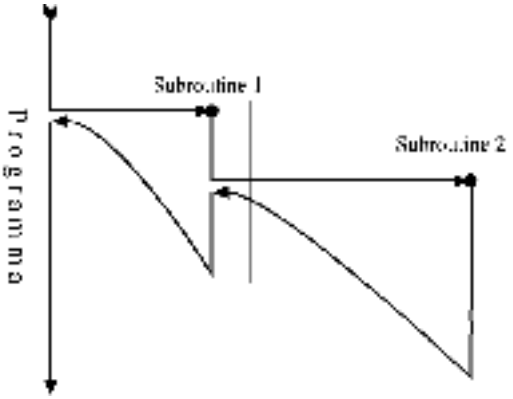
```
regel = 3
kolom = 23

als (afz.land is gevuld) dan
       sleutel = afz.land
       doe test_land
       als (leesfout) dan
               ...
               ...
       einde als
einde als

als (ontv.land is gevuld) dan
       regel = regel + 1

       sleutel = ontv.land
       doe test_land
       als (leesfout) dan
               ...
               ...
       einde als
einde als

...
...

test_land,
       leesfout = 0
```

```
        lees landenbestand met sleutel
        als (leesfout) dan
                doe toon (regel, kolom, '???')
        anders
                doe toon (regel, kolom, landnaam)
        einde als
return
```

Het leuke aan programmeren is dat er steeds opnieuw bekeken moet worden hoe een probleem opgelost moet worden. Wanneer de vorige voorbeelden allemaal eens goed bekeken worden komt duidelijk naar voren dat TEST_LAND in de vorm van een functie de meest ideale oplossing is. Is de variabele LEESFOUT echter een algemene variabele, in plaats van zoals nu een locale, dan vervalt de doelmatigheid van TEST_LAND als functie want er wordt geen waarde meer geretourneerd, en is de oplossing TEST_LAND als externe subroutine (zonder parameter FOUT) te bouwen. Een andere oplossing is om in TEST_LAND een andere variabele voor de leesfouten te definiëren, waardoor het een functie kan blijven, wat nog altijd de voorkeur geniet. Het zijn uiteindelijk persoonlijke keuzen en werkwijzen, wel of niet door een organisatie opgelegd. En het ligt uiteraard ook voor een groot deel aan de mogelijkheden die een programmeertaal de programmeur biedt.

Werking van een subroutine of functie

Het aanroepen van een subroutine of een functie is niets meer dan het tijdelijk onderbreken van het lopende programma en het springen naar het begin van de subroutine of functie, net als een gewoon GOTO statement waarvan het gebruik zo wordt afgeraden. Bij een goed geprogrammeerd 3GL programma gebruik je GOTO ook niet of nauwelijks. Door het (X)CALL, GOSUB of EXSR statement wordt in het geheugen het adres opgeslagen waar het programma was vóórdat het naar het begin van de routine sprong. En dat gebeurt bij een simpel GOTO statement niet. Vervolgens gaat het programma in de routine verder met uitvoeren van de aldaar geprogrammeerde code tot het een terugkeer-statement tegekomt. Het terugkeerstatement kan RETURN, XRETURN, FRETURN, ENDSR of END SUB zijn, afhankelijk van de gebruikte taal. Het geeft aan dat teruggesprongen moet worden naar de aanroepende routine en verder moet worden gegaan met de eerstvolgende bewerking na het aanroeppunt. Dat wordt mogelijk gemaakt doordat het startpunt werd bewaard:



Voorbeeld 18: Schematische weergave van de werking van een subroutine of functie

Vanuit het hoofdprogramma wordt SUBROUTINE 1 aangeroepen en in SUBROUTINE 1 wordt naar SUBROUTINE 2 gesprongen. De betreffende statements worden uitgevoerd en door bijvoorbeeld een RETURN wordt vanuit SUBROUTINE 2 teruggesprongen naar SUBROUTINE 1. Vervolgens wordt daar verder gegaan et de uitvoering ervan en wordt op een bepaald moment teruggesprongen naar PROGRAMMA en wordt dat verder uitgevoerd.

Lus

(Loop)

Een lus is het onderdeel waar het bij het bouwen programma's toch wel vaak om te doen is. Een lus zorgt namelijk voor de *herhaling van statements*. Juist de dingen die men wenst te automatiseren. Denk eens aan het volgen van de positie van de muis-aanwijzer op het scherm of het aftasten van het toetsenbord of er een toets is ingedrukt, etc.. Dat doet een besturingssysteem niet slechts één keer, maar bij herhaling. Een lus kan daarom uit één of meer statements bestaan die de kans krijgen één of meerdere keren uitgevoerd te worden. Er zijn twee soorten lussen waarmee herhaling te creëren is. De eerste soort kent geen einde, behalve als het hele programma wordt gestopt. Dat is de eindeloze lus (*endless loop*):

```
Label:                  repeat begin            do *hival
        ...                     ...             ...
        ...                     ...             ...
        GOTO Label      end                     enddo
```

Voorbeeld 19: Drie varianten van eindeloze lussen in BASIC, SL en RPG/400

Uiteraard zijn er statements in een programmeertaal waarmee uit elke soort lus gesprongen kan worden, zoals BREAK, EXIT, EXITLOOP of LEAVE, afhankelijk van de taal. Verlaat een lus nooit met een GOTO statement, tenzij de lus is opgebouwd uit een GOTO statement. Het besturingssysteem dat het programma uitvoerd, bewaart, net als bij een subroutine, het adres van het begin van de lus als *returnadres* voor het einde van de lus. Wanneer de lus dan onjuist wordt erlaten, blijft dit returnadres hangen. Tegenwoordig signaleren de compilers deze programmeerfouten op voorhand.

De tweede soort lussen zijn lussen met een voorwaardelijk einde of een voorwaardelijke start (*conditional loop*). Dat wil zeggen dat de lus helemaal niet, minimaal één keer of keer op keer wordt uitgevoerd zolang of totdat er aan één van de geprogrammeerde voorwaarden wordt voldaan. De voorwaardelijke lus kent drie typen.

'while' lus

Dit type heeft de voorwaarde aan het begin van de lus, nog voordat er statements in de lus worden uitgevoerd. We noemen dit de (doe-)zolang (*(do-)while*) lus. Omdat de voorwaarde zich bovenaan de lus bevindt, moet ervoor worden gezorgd dat de variabelen die in de voorwaarde worden gebruikt, vóórdat de voorwaarde wordt uitgevoerd, de juiste waarden bevatten zodat de voorwaarde kan bepalen of de lus wel of niet moet worden uitgevoerd. Vervolgens moeten diezelfde variabelen ook in de lus van waarde kunnen veranderen om aan

te geven dat de lus moet stoppen met herhalen. Hier volgen twee voorbeelden ter verduidelijking, waarbij de cursieve weergave de statements aangeeft die de variabele IN99 instellen vóórdat de voorwaarde wordt geëvalueerd:

```
call ReadsFile                          READ FILE      99
while .not.IN99 begin           *IN99   DOWEQ*OFF
        ...                             ...
        ...                             ...
        call ReadsFile                  READ FILE      99
end                                     ENDDO
...                                     ...
...                                     ...
close channel                           SETON   LR
stop                                    RETRN

ReadsFile,
        IN99 = 1
        onerror RF01
        reads (channel, record)
        clear IN99
RF01,   offerror
return
```

Voorbeeld 20: Twee voorwaardelijke lussen in SL en RPG/400

In beide gevallen wordt hier sequentieel een bestand gelezen. Zoals je ziet wordt vóór de lus al bepaald, door de eerste lees-actie, of de deze uitgevoerd moet worden of niet. Mocht in dit geval het bestand leeg zijn, dan wordt de lus dus nooit uitgevoerd omdat al bij de eerste READ(S) de variabele IN99 de waarde 1 krijgt. Het grote verschil met beide voorbeelden is dat aan het RPG standaard een indicator gekoppeld moet worden voor het afvangen van een einde-bestand (*EOF=End Of File*) fout. In andere talen moet dat apart worden geprogrammeerd.

Om te voorkomen dat deze lussen eindeloos worden uitgevoerd, worden dezelfde tests die vóóraf gedaan zijn, weer gedaan aan het einde van de lus. Het eerstvolgende statement dat op de test volgt, is dan de voorwaarde van de lus.

'until' lus

Bij dit type lussen bevindt de voorwaarde zich aan het einde van de lus. Dit zijn de doe-totdat (*do-until*) lussen. Omdat de voorwaarde pas aan het einde van de lus kan worden geëvalueerd, worden de statements in de lus altijd één keer uitgevoerd. Bijvoorbeeld:

```
tel = 0                         tel = 0;
DO                              do {
   PRINT tel                       printf ("%d\n", tel);
   tel = tel + 1                           tel = tel + 1;
LOOP UNTIL tel = 10             } while (tel < 10);
```

Voorbeeld 21: Twee voorwaardelijke lussen in BASIC en C

In deze twee voorbeelden zien we dat de variabele TEL altijd tenminste één keer naar het

scherm wordt geprint en met de waarde 1 wordt verhoogd. Pas daarna wordt gekeken of TEL nog eens geprint en verhoogd moet worden. Hier komt tevens een groot verschil in programeertaal naar voren: in de BASIC versie van het voorbeeld geldt de uitvoering *tot* de variabele TEL de waarde 10 heeft. In de C versie geldt dat de lus wordt uitgevoerd *zolang* de variabele TEL een waarde bevat die kleiner is dan 10.

'for' lus

Deze laatste is eigenlijk een vereenvoudigde variant van de while-lus. Het doel ervan is de statements een voorwaardelijk aantal keer te herhalen, waarbij de stapgrootte gevarieerd kan worden: de voor-volgende (*for-next*) lus. Deze zeer bekende lus voert de tussen het begin en eind liggende statements een 'vastgesteld' aantal malen uit, door bij de start van de lus een begin- en eindwaarde en stapgrootte te specificeren en een variabele welke bij elke iteratie wordt opgehoogd met de stapgrootte van de lus. Zonder verdere opgave van een stapgrootte is deze altijd 1. Zodra de waarde van de tellervariabele de eindwaarde overschrijdt, wordt de lus beëindigd:

```
for teller from 0 thru 9                        for teller = 0 to 9
        writes (1, %string(teller))           PRINT teller
                                            next teller
```

Voorbeeld 22: Twee 'for' lussen in SL en BASIC

De uitvoer is hetzelfde als de eerdere voorbeelden met de DO-UNTIL lus, en kost duidelijk minder statements.

Al deze lussen zijn ontstaan ter vervanging van de walgelijke constructies om een lus te programmeren die sommigen onder ons met IF-THEN-ELSE en GOTO statements wisten te produceren.

Als-dan-anders

(if - then - else)

Een niet te onderschatten onderdeel van een programma is de ALS-DAN-ANDERS constructie. Met deze statements bouwt men voorwaarden in een programma in die tijdens het draaien van het programma bepalen welke statements uitgevoerd dienen te worden. Als aan een voorwaarde wordt voldaan dan doet het programma iets, anders doet het iets niet of iets anders. Een IF-THEN-ELSE constructie is in zijn geheel een taalelement. Het is daardoor mogelijk in een IF weer een IF op te nemen, en daarbinnen weer, enzovoort. Date heten geneste IF-THE-ELSE-constructies (*nested if-then-else*). De basisversie van het IF-statement ziet er als volgt uit:

```
IF <voorwaarde> THEN
        ...
        ...
ELSE
```

```
        ...
        ...
END IF
```

Een geneste versie kan er als volgt uitzien:

```
IF <voorwaarde1> THEN
        IF <voorwaarde2> THEN
                ...
                ...
        ELSE
                ...
                IF <voorwaarde3> THEN
                        ...
                        ...
                ELSE
                        ...
                        ...
                END IF
                ...
        END IF
        ...
        ...
ELSE
        ...
        IF <voorwaarde4> THEN
                ...
                ...
        ELSE
                ...
                ...
        END IF
        ...
        IF <voorwaarde5> THEN
                ...
                IF <voorwaarde6> THEN
                        ...
                        ...
                ELSE
                        ...
                        ...
                END IF
                ...
        ELSE
                ...
                ...
        END IF
END IF
```

Deze vergelijkende constructie komt in verschillende varianten voor: omdat sommige uitgebreide voorwaarden ellenlange geneste constructies nodig hebben die zeer onoverzichtelijk worden naarmate ze groeien, zijn er nieuwe constructies bedacht voor een vereenvoudigde en duidelijkere weergave. Omdat 3GL toch al veel tikken is, proberen de ontwikkelaars van een taal deze zo eenvoudig mogelijk te krijgen.

Selectie-blokken

Deze statements beginnen niet met IF maar hebben een eigen syntax en zijn per taal verschillend. Zo bestaan er USING SELECT, SELECT CASE, CASE, SWITCH, SELEC, etc. Hier volgen diverse voorbeelden waarbij als eerste een IF-THEN-ELSE constructie wordt gegeven en vervolgens wordt diezelfde IF-THEN-ELSE omgebouwd naar een SELECTIE-blok. Getracht is de uit de verschillende talen overeenkomende statements zoveel mogelijk op één regel te krijgen zodat de overeenkomsten redelijk zichtbaar zijn:

```
IF (a = 1 OR a = 3) THEN                if (a == 1 || a == 3) {
        ...                                     ...
ELSEIF a = 2 THEN                       } else if (a == 2) {
        ...                                     ...
ELSEIF a = 4 THEN                       } else if (a == 4) {
        ...                                     ...
ELSE                                    } else {
        ...                                     ...
END IF                                  }



SELECT CASE a                           switch (a) {
   CASE 1,3                                case 1,3:
        ...                                     ...
        ...                                     break;
   CASE 2                                  case 2:
        ...                                     ...
        ...                                     break;
   CASE 4                                  case 4:
        ...                                     ...
        ...                                     break;
   CASE ELSE                               default:
        ...                                     ...
        ...                                     break;
END SELECT                              }
```

Voorbeeld 23: Twee IF-statements en een nettere vervanger in BASIC en C

```
A       IFEQ 1                          if a.eq.1 .or.
A       OREQ 3                          &  a.eq.3 then begin
        ...                                     ...
        ELSE                            end
A       IFEQ 2                          else if a.eq.2 then begin
        ...                                     ...
        ELSE                            end
A       IFEQ 4                          else if a.eq.4 then begin
        ...                                     ...
        ...                             end
        ELSE                            else begin
        ...                                     ...
        ENDIF                           end
        ENDIF
        ENDIF



        SELEC                           using a select
A       WHEQ 1                          (1,3),  begin
A       OREQ 3                                  ...
        ...                             end
A       WHEQ 2                          (2),    begin
        ...                                     ...
        ...                             end
A       WHEQ 4                          (4),    begin
        ...                                     ...
```

```
...                     end
OTHER                   (),     begin
...                             ...
...                     end
ENDSL                   endusing
```

Voorbeeld 24: Twee IF-statements en een nettere vervanger in RPG/400 en SL

Zoals te zien is, heeft elke taal een eigen oplossing voor eenzelfde soort probleem dat past binnen de structuur en de opbouw van die taal. De enige echt afwijkende is die in RPG/400, want daar wordt niet vóóraf bepaald welke expressie of variabele wordt geëvalueerd in het SELEC statementblok. Een WHEQ vergelijking mag dus ook over iets heel anders gaan, dan de vorige WHEQ. Bij de overige drie kan dat niet omdat daar van te voren is aangegeven dat het om de inhoud van variabele A gaat. RPG/400 kent ook nog het CAS-statement, wat overeenkomt met CASE. Echter is CAS minder flexibel in het gebruik dan SELEC.

Statement-blok

Wat ondertussen ook opgevallen moet zijn is dat elk IF-statement, of een variant ervan, soms een begin en einde heeft, aangegeven door { en } of BEGIN en END of alleen een ENDIF. Deze uitbreiding is een logische ontwikkeling op het oorspronkelijke IF-statement dat eigenlijk slechts één statement kan uitvoeren:

```
IF (a = 1) b = 4                    if (a == 1) b=4;
```

Door het gebruik van begin-einde aanduidingen kan het geregeld worden dat er meerdere statements gegroepeerd worden tot één statement-blok waaraan de voorwaarde gekoppeld kan worden. Dat zelfde geldt ook voor de eerder besproken lussen. In het volgende voorbeeld wordt een simpel statement-blok gedemonstreerd:

```
IF a = 1 THEN BEEP              IF a = 1 THEN
BEEP                                    BEEP
                                        BEEP
                                END IF
```

Bij het linker voorbeeld piept het programma altijd één keer, tenzij de inhoud van A gelijk is aan de waarde 1, dan wordt er twee keer gepiept. Bij het rechter voorbeeld piept het programma nooit, tenzij A de waarde 1 bevat: dan piept het programma twee keer. Elke taal heeft de beschikking over aanduidingen om een statement-blok te creëren. Als je terugblikt naar de voorgaande voorbeelden heb je al statement-blokken in gebruik gezien.

In SL begint een blok met BEGIN en eindigt met END. In C begint een blok met { en eindigt met }. Merk op dat bij deze twee talen de aanduidingen voor een statement-blok altijd dezelfde zijn, ongeacht het statement:

```
if a.eq.1 then begin            if (a == 1) {
        ...                             ...
```

```
        ...                             ...
end                                 }
else begin                          else {
        ...                             ...
        ...                             ...
end                                 }


while a.eq.1 begin                  while (a == 1) {
        ...                             ...
        ...                             ...
end                                 }
```

Voorbeeld 25: Statement-blokken in SL en C

RPG/400 en BASIC hebben voor elk statement waaraan een statement-blok gekoppeld kan worden een bijbehorend END statement. In RPG mag een kale END ook, maar het wordt vaak zeer onduidelijk waar de END bijhoort:

```
IFEQ        SELEC       DO          BEGSR       CASxx
...         ...         ...         ...         ...
...         ...         ...         ...         ...
ENDIF       ENDSL       ENDDO       ENDSR       ENDCS
```

Voorbeeld 26: Statement-blokken in RPG/400

```
IF <> THEN      SELECT CASE     WHILE           SUB
...             ...             ...             ...
...             ...             ...             ...
END IF          END SELECT      WEND            END SUB
```

Voorbeeld 27: Statement-blokken in BASIC

IF-statements, lussen en routine-definities zijn in RPG/400 altijd een statement-blok. Bij de andere talen kun je in de meeste situaties de begin en einde aanduiding weglaten, met dien verstande dat er dan ook slechts één statement wordt uitgevoerd:

```
if a.eq.1 then                  if (a == 1)
        b=4                             b=4;
else    b=6                     else    b=6;
```

Voorbeeld 28: Enkele statements in een IF-statement in SL en C

```
while a.eq.1 call subA          while (a == 1) subA();


while b.ne.4                    while (b != 4)
        if a.eq.1 then                  if (a == 1)
                b=4                             b=4;
        else    b=6                     else    b=6;
```

Voorbeeld 29: Enkele statements in een WHILE-statement in SL en C

In deze voorbeelden wordt slechts één statement uitgevoerd in de IF of de ELSE. Vergis je niet in het laatste voorbeeld, want zoals gezegd is het IF-statement, met of zonder ELSE en met of zonder begin-einde aanduidingen, één statement. Om die reden hoeven er geen begin-einde aanduidingen bij het WHILE-statement te worden geprogrammeerd. Je kunt statement-blokken natuurlijk ook gewoon door elkaar toepassen:

```
if a.eq.1 then                  if (a == 1)
        call subA                       subA();
else begin                      else {
        call subB                       subB();
        call subC                       subC();
        call subD                       subD();
end                             }
```

Voorbeeld 30: Statement-blok in een IF-statement in SL en C

```
while a.eq.1                    while (a == 1)
        if a.eq.1 then                  if (a == 1)
                call subA                       subA();
        else begin                      else {
                call subB                       subB();
                call subC                       subC();
                call subD                       subD();
        end                             }
```

Voorbeeld 31: Statement-blok in een WHILE-statement in SL en C

Voorwaarde

(condition)

In voorgaande voorbeelden is al diverse keren een vergelijking gebruikt als TEL < 10 of .NOT.IN99 en andere. Eén of meer vergelijkingen tesamen noemen we een ‘voorwaarde’ (*condition*). De voorwaarden in eerdere voorbeelden waren tamelijk eenvoudig en probeer dat ook zo te doen, want in sommige talen kan het soms tamelijk onoverzichtelijk worden.

Nadat alle vergelijkingen geëvalueerd zijn, moet het resultaat 0, wat *niet waar* betekent, of niet-nul, wat *waar* betekent, zijn. Een vergelijking bestaat uit het op binair niveau vergelijken van de inhoud van een variabele met de inhoud van een andere variabele, literal of constante. De resultaten van de individuele vergelijkingen kunnen worden gekoppeld door middel van EN (*and*) en OF (*or*) operatoren, waardoor weer nieuwe vergelijkingen ontstaan. Het resultaat van de vergelijkingen kan daarnaast ook nog beïnvloed worden door gebruik te maken van '(' en ')'. Met deze ronde haken wordt de volgorde bepaald waarin de vergelijkingen moeten worden geëvalueerd. De vergelijking die zich tussen de binnenste haken bevindt wordt het eerst geëvalueerd, vervolgens het volgende niveau, en zo verder totdat de hele voorwaarde is verwerkt. Bijvoorbeeld:

```
tel < 10 EN tabel(1) = 1 OF tabel(1) = 10
```

Wat nu? Gaat EN eerst en dan OF? Of gaat OF eerst en dan EN? Horen de vergelijkingen

met EN ertsussen bij elkaar of juist die met OF ertussen? De EN operator verbindt de vergelijkingen links en rechts ervan en moet resulteren in één waarde. De OF vergelijking komt erna. Hier staat: als TEL kleiner 10 *en* TABEL(1) = 1 dan doe IETS *of* als TABEL(1) = 10 doe dan datzelfde IETS. Wanneer er ronde haken rond de laatste twee vergelijkingen worden geplaatst…

```
tel < 10 EN (tabel(1) = 1 OF tabel(1) = 10)
```

… gebeurt er iets heel anders. Nu is aangegeven dat de OF voorwaarde als eerste moet worden uitgewerkt. Dat geeft een bepaalde uitkomst die pas daarna wordt gebruikt voor de EN vergelijking. In het hier volgende stuk wordt hier dieper op in gegaan.

Een voorbeeld : het plannen van een feest

Goede voorwaarden samenstellen kan een lastig karwei zijn. Er komt veel logica bij kijken en men moet redelijk probleem-deducerend en analyserend kunnen denken. Daar komt nog bij dat het heel duidelijk voorop moet staan wat men wil bereiken om zo tot een voorwaarde te komen waarvan men weet hoe die wordt geëvalueerd. Wanneer begrepen wordt wat er gebeurd, wordt het voor velen al een stuk makkelijker. Daarom gaan we een programma schrijven ten behoeve van een groot feest voor komende zaterdag waarmee dagelijks de resultaten van de verzonden uitnodigingen worden ingevoerd. Met die, een andere externe, gegevens moet het programma bepalen of het zin heeft het feest door te laten gaan. De programmadelen die gegevens halen uit het adressenbestand met vrienden en kennissen en die via het internet de laatste weersvoorspellingen voor de komende week ophalen, zijn al geschreven en kunnen in het programma worden gebruikt.

Het eerste waar iemand aan denkt bij het plannen van een feest is… wat wordt het voor weer. Als het prachtig weer wordt is er niets aan de hand. Gaat het echter sneeuwen of regenen met onweer, dan zou het kunnen zijn dat je moet afzeggen. Want je zou ook een partytent kunnen regelen, of het feest binnenshuis houden, vooropgesteld dat er dan niet te veel gasten komen. Daarom moeten de genodigden laten weten of ze komen of niet.

Wanneer duidelijk is geworden dat het feest door kan gaan komt het volgende probleem naar voren: hoeveel eten en drinken moet je inslaan? In theorie komen alle genodigden, het kan echter ook zijn dat slechts de helft komt opdagen, of zelfs dat er meer komen dan genodigd. De meesten zullen keurig laten weten dat ze wel of niet komen, het kan echter nooit van te voren precies worden vastgesteld. Daarom moet er een voorraad worden aangelegd met zaken die je kan bewaren. Een andere vraag is bij hoeveel afzeggingen het feest toch nog door mag gaan? Dat is namelijk voor een groot deel afhankelijk van wie er wèl komen, want met een kleine groep goede vrienden is het ook reuze gezellig. Hier volgt een samenvatting van de vragen die we ons hebben gesteld en die interessant zijn om op door te gaan:

- Goed weer of niet.
- Tent beschikbaar of niet.
- Komt iedereen.
- Komen er meer.

- Hoeveel komen er niet.
- Blijven er, van de mensen die wèl komen, leuke mensen over of niet.

Het eerste wat geprogrammeerd moet worden is de eerste vraag: of het mooi weer zal zijn of niet. Hiervoor introduceren we een variabele in de vorm van een tabel (*array*) van 7 elementen die elk de de waarde 1 of 0 kunnen krijgen. Elk element stelt een dag van de week voor en 1 betekent dat het gaat regenen en 0 dat het droog blijft. De array heet REGEN. Voor de dagen van de week zijn constanten gedefinieerd (maa = 1, din = 2, etc). Een hier niet verder uit te werken routine haalt regelmatig de weersvoorspellingen op en vult de betreffende elementen van de tabel REGEN met 1 of 0. Aangezien er periodiek bekeken moet worden of het die zaterdag regent of niet, maken we gebruik van een *while*-lus met een voorwaarde waarin steeds het zesde element van de tabel REGEN wordt getest op de waarde 0, door gebruik te maken van de constante ZAT:

```
WHILE regen(zat) = 0
        ...
        ...
WEND
```

Stel dat het zou gaan regenen op die dag, zou de WHILE-lus eindigen en geeft het programma aan dat het feest definitief niet door kan gaan. Er waren echter nog meer voorwaarden gesteld waaraan ook voldaan moet worden en daarom koppelen we nu de tweede vergelijking aan de voorwaarde van de lus.

We hebben alleen de beschikking over een tent nodig als het die dag regent. De grote vraag hierbij is natuurlijk of er wel een tent beschikbaar ís op die dag. We benoemen een variabele TENT die ook de waarden 1 en 0 kan krijgen. De waarde 1 wil zeggen dat we aan een tent kunnen komen en de waarde 0 geeft aan dat we het hoe dan ook zonder tent moeten stellen. We gaan er vanuit dat zodra de situatie met betrekking tot de beschikking verandert, dit op een of andere manier aan het programma wordt doorgegeven, bijvoorbeeld door middel van een functie.

Verder kunnen we stellen dat de beschikking over een tent alleen interessant is als het regent op die dag. Daarom wordt deze vergelijking met een OF-afvraging aan de reeds bestaande vergelijking gekoppeld. En omdat de nieuw ontstane voorwaarde uit drie vergelijkingen bestaat waarvan er slechts twee echt bij elkaar horen, plaatsen we ronde haken om die twee vergelijkingen zodat die vergelijkingen als eerste worden geëvalueerd en als één voorwaarde worden behandeld:

```
WHILE regen(zat) = 0 OR (regen(zat) <> 0 AND tent <> 0)
        ...
        ...
WEND
```

De voorwaarde zoals die nu is zegt het volgende: zolang het zaterdag niet regent *OF* (als het wel regent *en* we hebben de beschikking over een tent) blijft de lus actief en gaat het feest door.

Bij de volgende punten op de lijst wordt het wat lastiger om aan te geven of het feest nog door moet of kan gaan, want of het nou prachtig weer is of niet, als niemand wenst te komen is er eigenlijk ook geen feest. Als houvast kan het aantal afzeggingen worden genomen en kan voor dat aantal een drempelwaarde worden vastgesteld: indien het aantal afzeggingen groter is dan bijvoorbeeld 75% van het totale aantal genodigden, laten we het feest niet doorgaan, ongeacht het resultaat van de eerdere vergelijkingen. Deze nieuwe vergelijking is dus tamelijk bepalend voor het laten voortduren van de WHILE-lus. En we weten dat de dat het feest niet door gaat als de lus wordt beëindigd. We willen echter de lus actief houden en dat betekent dat we de vergelijking rond de drempelwaarde om moeten draaien zodat deze niet aangeeft dat het feest niet doorgaat, maar juis aangeeft dat het wèl doorgaat: in dat geval moet het aantal afzeggingen niet groter, maar kleiner of gelijk (<=) blijven aan 75% van het totale aantal genodigden. Een percentage van 75 verkrijg je door een waarde met 0,75 te vermenigvuldigen. Zijn getallen met een komma lasitg te programmeren in de taal die je gebruikt, dan werkt de volgende berekening in ieder geval bij alle talen: (waarde * 75) / 100.

Om er voor te zorgen dat wanneer het aantal afzeggingen boven de 75% komt, de hele voorwaarde niet meer waar mag zijn, moeten we de huidige voorwaarde ombouwen tot één vergelijking en deze door middel van een EN-afvraging koppelen aan de nieuwe vergelijking. Dat ombouwen tot één vergelijking is snel voor elkaar door om de voorwaarde ronde haken te plaatsen:

```
WHILE (regen(zat) = 0 OR (regen(zat) <> 0 AND tent <> 0)) AND
      (afzeggingen <= (genodigden * 0,75))
        ...
        ...
WEND
```

De voorwaarde beweert nu dat: zolang het (zaterdag niet regent *of* (als het wel regent *en* we hebben de beschikking over een tent)) *EN* (het aantal afzeggingen is kleiner of gelijk aan 75% van de genodigden) het feest doorgaat. Als een voorwaarde onoverzichtelijk is kan men de structuur verduidelijken door de tussenliggende vergelijkingen weglaten:

```
WHILE (... OR (... AND ...)) AND (...)
```

Maar we hebben nog één vraag over de bezoekers niet verwerkt. Stel nou eens dat 80% afzegt, maar dat de overblijvende 20% zulke goede familieleden, vrienden en kennissen zijn waarmee het altijd goed verpozen is, dat het gewoonweg zonde zou zijn om het feest níet door te laten gaan. In zo'n geval moet het mogelijk zijn de 75%-grens niet bepalend te laten zijn voor het beëindigen van de lus. Er moet daarom aan de 75%-vergelijking een nieuwe vergelijking worden gekoppeld die test of de overblijvers een acceptabel geheel vormen. De koppeling vindt plaats met OF en tesamen vormen zij één vergelijking.

De test op acceptabele overblijvers wordt door een, hier niet uitgewerkte, funtie gedaan die het adressenbestand raadpleegt op vriendelijkheids- en lolbroekattributen. Bevatten de overblijvers een hoog gehalte van deze attributen, stel een drempel van 35%, dan wordt de variabele OVERBLIJVERS met de waarde 1 gevuld anders met 0:

```
WHILE (regen(zat) = 0 OR (regen(zat) <> 0 AND tent <> 0)) AND
      (afzeggingen <= (genodigden * 0,75) OR
      (afzeggingen > (genodigden * 0,75) AND overblijvers = 1)
        ...
        ...
WEND
```

Optimaliseren van een voorwaarde

Zoals de voorwaarde er nu uitziet is die redelijk leesbaar doch zeer onvriendelijk in het gebruik: hij is tamelijk lang, wel drie regels, en er vinden bewerkingen in plaats zoals vermenigvuldigingen, en notabene nog wel twee keer dezelfde vermenigvuldiging ook. Nu zou je de namen van de variabelen kunnen inkorten, dan zie je de structuur misschien iets beter, maar dat is nou net niet de bedoeling. Korte namen voor variabelen is vaak wel makkelijk en ze worden ook redelijk snel begrepen, maar ze kunnen ook voor een zeer grote onduidelijkheid zorgen.

Voordat we aan de namen gaan sleutelen moet er eerst met wat logica getracht worden de voorwaarde in te korten. We beginnen met de regen: zolang het er naar uit ziet dat het gaat regenen willen we een tent. De variabele REGEN(ZAT) zal dan ongelijk nul zijn om aan te geven dat er voor zaterdag regen is voorspeld. Gaat het niet regenen, dan willen we ook geen tent. De variabele REGEN(ZAT) zal dan gelijk aan nul zijn. Dat betekent dat het feest dus alleen doorgaat als het niet regent *of* als er een tent beschikbaar is, want in dat geval doet het er eigenlijk helemaal niet toe of het regent of niet. Als we deze beredenering verwerken in het eerste deel van de voorwaarde, gaat dat er als volgt uit zien:

```
WHILE (regen(zat) = 0 OR tent <> 0) .........
        ...
        ...
WEND
```

Dit kan zelfs nog iets korter, want vergelijken op <> 0 is hetzelfde als het alleen plaatsen van de variabele. Want heeft de variabele een waarde die ongelijk is aan nul, dan hoef je dat niet ook nog eens te programmeren. Zo wordt de voorwaarde verder ingekort:

```
WHILE (regen(zat) = 0 OR tent) .........
        ...
        ...
WEND
```

Dezelfde beredeneringen gaan we toepassen op de rest van de in de voorwaarde aanwezige vergelijkingen.

Zo komt er twee maal dezelfde berekening in voor, namelijk (genodigden * 0,75). Die berekening geeft in alle gevallen hetzelfde resultaat, het aantal genodigden ligt al vast, en we

kunnen de berekening dus vervangen door een variabele. Deze heet DREMPEL en wordt vóóraf en tijdens de lus gevuld met het resultaat van de berekening:

```
drempel = genodigden * 0,75
```

Of, zoals eerder gezegd, gebruik de volgende constructie als je '0,75' niet kunt toepassen:

```
drempel = (genodigden * 75) / 100
```

Altijd eerst vermenigvuldigen en dan pas delen, anders raak je decimalen kwijt bij het gebruik van velden met een vast aantal decimalen. De variabele AFZEGGINGEN kan eigenlijk ook vervangen worden, en wel door de hier niet uitgewerkte functie AFZEGGINGEN(). Ook vervangen we de berekeningen in de voorwaarde door DREMPEL:

```
drempel = genodigden * 0,75

WHILE (...) AND (afzeggingen() <= drempel OR (afzeggingen() >
           drempel AND overblijvers = 1))
        ...
        ...
WEND
```

Twee maal dezelfde functie in een voorwaarde werkt ook vertragend en dus passen we dezelfde redenering toe op het aantal afzeggingen als bij de eerste vergelijkingen met betrekking tot het weer: zolang het aantal afzeggingen kleiner of gelijk is aan de gestelde grens óf de overblijvers zijn genodigden die we sowieso altijd graag zien, dán mag het feest doorgaan. De variabele OVERBLIJVERS bouwen we ook om naar een functie waardoor het resultaat variabel worden kan, want het kan natuurlijk zijn dat iemand die altijd aardig was, plotseling niet meer aardig gevonden wordt. De test `wie_komt_niet() > drempel` is dus overbodig. Dat resulteert in de volgende verkorte voorwaarde:

```
WHILE (...) AND (afzeggingen() <= drempel OR overblijvers() = 1))
        ...
        ...
WEND
```

Het kan nog verder worden ingekort door '= 1' te laten vervallen bij de vergelijking met de fundtie OVERBLIJVERS(). Het gaat er eigenlijk alleen om dat de functie geen waarde nul retourneerd en dus doen de andere waarden er niet toe. Meer valt er niet in te korten en daarom laten we nu in één keer de hele bijgewerkte voorwaarde zien:

```
WHILE (regen(zat) = 0 OR tent) AND (afzeggingen() <= drempel OR
       overblijvers()))
        ...
        ...
```

```
WEND
```

Ter verduidelijking: zolang (het zaterdag niet regent *of* we hebben een tent) *EN* (het aantal afzeggingen kleiner of gelijk aan een gestelde grens *of* het percentage overblijvers is geschikt) kan het feest doorgaan.

Elk probleem is in een programma te vatten, als er van te voren maar goed over wordt nagedacht, alles goed op een rijtje wordt gezet. En, veel belangrijker, dat er achteraf wordt gekeken of er geen verbeteringen aangebracht kunnen worden, zowel in de source als aan de functionaliteit. Zo ontstaan de updates en upgrades.

Evaluatie van een voorwaarde

Hoe vindt nu de evaluatie van de voorwaarde in het WHILE-statement plaats? Om dat te verduidelijken krijgen de gebruikte variabelen een waarde:

```
regen(zat)      = 0             ' Het regent niet op zaterdag
tent            = 0             ' Geen beschikking over een tent
genodigden      = 100           ' 100 mensen uitgenodigd
afzeggingen()   = 30            ' 30 komen waarschijnlijk niet
drempel                 = 75            ' 100 * 0,75
overblijvers()  = 0             ' Nog niet genoeg 'leuke' gasten
```

Vervolgens substitueren we de variabelen functies door hun waarde. Het resultaat van een vergelijking is waar (*true*) of niet waar (*false*). De voorwaarde wordt nu stap voor stap uitgewerkt:

```
while (0 = 0 OR 0) AND (30 <= 75 OR 0)

1:      while (true OR 0) AND (true OR 0)
2:      while (true) AND (true)
3:      while (true)
```

… en dus wordt de WHILE-loop nog eens uitgevoerd. De vergelijking REGEN(ZAT) = 0 is wáár is omdat REGEN(ZAT) de waarde NUL heeft en deze waarde met NUL vergeleken wordt.

Bij een OR operatie maakt het niet uit welke vergelijking, links of rechts van de OR, waar is. Zodra één van de twee waar is, is het resultaat van de OR ook 'waar'. Zijn ze beide NUL dan is het resultaat van de OR 'niet waar':

```
0 OR 0  is false
0 OR 1  is true
1 OR 0  is true
1 OR 1  is true
```

Bij de AND operatie moéten beide kanten waar zijn wil het resultaat van de AND als geheel waar worden:

```
0 AND 0 is true
0 AND 1 is false
1 AND 0 is false
1 AND 1 is true
```

Stel de variabele TENT krijgt de waarde 1. Uit het voorgaande blijkt dat dat geen invloed heeft op het eindresultaat van de OR omdat in dit geval REGEN(ZAT) de waarde nul heeft. Zet TENT weer op nul en laat nu de functie OVERBLIJVERS() eens een 1 teruggeven. Ook dan blijkt dit geen invloed te hebben, omdat de linkervergelijking van de OR al *true* is. Laat OVERBLIJVERS() weer nul teruggeven en zet dan het array-element REGEN(ZAT) eens op 1:

```
regen(zat)      = 1             ' Het regent op zaterdag
tent            = 0             ' Nog geen tent kunnen vinden
genodigden      = 100           ' 100 mensen uitgenodigd
afzeggingen()   = 60            ' Er komen er nu al 60 niet
drempel                 = 75            ' 100 * 0,75
overblijvers()  = 0             ' Nog niet genoeg 'leuke' gasten
```

dan wordt de voorwaarde als volgt geëvalueerd:

```
while (1 = 0 OR 0) AND (60 <= 75 OR 0)

1:      while (false OR 0) AND (true OR 0)
2:      while (false) AND (true)
3:      while (false)
```

De WHILE-loop wordt niet verder uitgevoerd en het programma geeft aan dat het feest niet door kan gaan. De vergelijking REGEN(ZAT) = 0 is nu *niet* waaren dus heeft de vergelijking als resultaat de waarde *false*.

Voorwaarden in berekeningen

De evaluatie-regels gelden overal waar voorwaarden kunnen worden toegepast. Nu we ook weten dat het resultaat van een evaluatie *true* of *false* is, zou het erg handig zijn wanneer we deze voorwaarden ook in situaties toe kunnen passen waarbij het resultaat in berekeningen rechts van het '=' teken gebruikt kan worden. Gelukkig kan dat in veel programmeertalen. Om te laten zien wat er wordt bedoeld en hoe een voorwaarde toegepast kan worden, volgt hier een voorbeeld met een traditionele situatie en een nieuwe:

```
IF toets = f3 THEN
        modus = 1
```

```
ELSEIF toets = f5 THEN
        modus = 3
ELSEIF toets = f6 THEN
        modus = 4
ELSE    modus = 2
```

Dit kan natuurlijk ook zo:

```
SELECT CASE toets
   CASE f3:
        modus = 1
   CASE f5:
        modus = 3
   CASE f6:
        modus = 4
   CASE ELSE:
        modus = 2
END SELECT
```

Maar het kan ook zo:

```
modus = (toets = f3)
```

Dit is nog niet het complete statement, maar slechts een deel ervan. De voorwaarde rechts van het '=' teken heeft de rol van een variabele gekregen. Als `toets=f3` wordt geëvalueerd op het moment dat TOETS een identieke waarde heeft als variabele of constante F3, is het resultaat *true*. De *waarde* van *true* wisselt echter per taal. In sommige talen is het 1 en in andere is het –1. Dit moet men van te voren weten als voorwaarden op deze manier gaan worden toegepast. We gaan uit van 1 als waarde voor *true*; bij –1 moet de voowaarde met –1 worden vermenigvuldigd om weer 1 te krijgen:

```
     modus = -1 * (toets = f3)
of:  modus = -(toets = f3)
```

Volgens de eerste, klassieke IF-ELSEIF moet MODUS de waarde 1 krijgen als TOETS gelijk is aan F3. Op deze wijze wordt de voorwaarde 1 als de voorwaarde wáár is en dus krijgt MODUS de waarde 1:

```
modus = (toets = f3)

1:      modus = (1)
2:      modus = 1
```

Nu doen we hetzelfde met variabele F5, die volgens het voorbeeld de waarde 3 moet krijgen. Aangezien de voowaarde resulteert in 1 of 0, kunnen we dit resultaat vermenigvuldigen met 3:

```
modus = 3 * (toets = f5)

1:      modus = 3 * (1)
2:      modus = 3 * 1
3:      modus = 3
```

Als de voorwaarde resulteerde in 0, kreeg modus ook de waarde 0. Deze voorwaarden kunnen natuurlijk niet als afzonderlijke statements worden uitgevoerd, dan zou de volgende evaluatie het resultaat van de vorige overschrijven. Nee, alles moet in één statement worden verwerkt:

```
modus = 2 - (toets = f3) + (toets = f5) + (2 * (toets = f6))
```

Hier volgt de uitwerking voor diverse toetsaanslagen om te bewijzen dat dit statement werkt:

```
als     toets = f3

1:      modus = 2 - (1) + (0) + (2 * (0))
2:      modus = 2 - 1   +  0  +  0
3:      modus = 2 - 1
4:      modus = 1


als     toets = f5

1:      modus = 2 - (0) + (1) + (2 * (0))
2:      modus = 2 -  0  +  1  +  0
3:      modus = 2 + 1
4:      modus = 3


als     toets = f6

1:      modus = 2 - (0) + (0) + (2 * (1))
2:      modus = 2 -  0  +  0  +  2
3:      modus = 2 + 2
4:      modus = 4


als     toets = f1

1:      modus = 2 - (0) + (0) + (2 * (0))
2:      modus = 2 -  0  +  0  +  0
3:      modus = 2
```

Op deze manier kan de voorwaarde van de WHILE-loop van het feestplannings-programma ook wat leesbaarder worden gemaakt:

```
goedweer = (regen(zat) = 0 OR tent)
mensen   = (afzeggingen() <= drempel OR overblijvers())
```

```
WHILE goedweer AND mensen
        ...
        ...

        goedweer = (regen(zat) = 0 OR tent)
        mensen   = (afzeggingen() <= drempel OR overblijvers())
WEND
```

Een voordeel van deze manier van programmeren is dat de variabelen GOEDWEER en MENSEN ook op andere manieren van een waarde kunnen worden voorzien, terwijl de voorwaarde in de WHILE-lus niet wordt aangetast. De beide statements vormen een herhaling en herhalingen horen in een interne subroutine thuis:

```
GOSUB "Criteria"
WHILE goedweer AND mensen
        ...
        ...

        GOSUB "Criteria"
WEND

...
...

"Criteria"
        goedweer = (regen(zat) = 0 OR tent)
        mensen   = (afzeggingen() <= drempel OR overblijvers())
RETURN
```

Recursie

(recursion)

Recursie duid op het 'zichzelf herhalen vanuit zichzelf'. Een analogie is het plaatsen van twee spiegels tegenover elkaar en er dan tussenin gaan staan. Je ziet hetzelfde beeld eindeloos herhaald. Bij een computerprogramma wordt bedoeld dat een routine, functie of het programma zelf, zichzelf weer aanroept.

Recursie kan heel gevaarlijk zijn en een niet mis te verstaan onderwerp bij sommige programmeertalen, omdat die er van in de war kunnen raken en daardoor het programma laten crashen. Er zijn ook talen waar rekening gehouden is met de mogelijkheid dat een routine zichzelf weer aanroept en er zijn zelfs talen die speciaal hiervoor geschreven zijn, de talen om Artificiële Intelligentieprogramma's mee te schrijven. De meeste talen, zeker die talen welke gebruik maken van algemene variabelen, staan recursie niet toe. De ene reden is dat een eigen geschreven routine misschien nooit stopt met het zichzelf aanroepen, dat er zo een eindeloze lus ontstaat. De andere reden is dat de algemene variabelen die gebruikt worden, door de volgende aanroep kunnen worden veranderd. Nog een andere reden is dat als de subroutine of functie al geladen is, hij niet nog eens aangeroepen kan worden, tenzij er een tweede instantie van wordt gestart, zoals bij talen als C het geval is. De routine is dan twee keer in het geheugen geladen en elke instantie heeft dus ook zijn eigen locale variabelen. Recursie is met name toe te passen als er boomstructuren moeten worden verwerkt. Voor elke tak van de structuur geldt namelijk dezelfde verwerking, en een tak ontspruit uit een voorgaande tak of

de stam. Een boomstam, daar groeien takken aan, waar weer takjes aan groeien, waar twijgjes aan groeien, enzovoort. Een boom weet echter wanneer de recursie, het zichzelf herhalen, moet stoppen en een blaadje moet doen groeien. Stopt hij niet, dan wordt de boom dus groter en groter en groter en groter. Kettingbrieven en pyramide-spelen hebben ook een boomstructuur en zijn ook zichzelf herhalende handelingen. Ook het zoeken naar de juiste weg door een routeplanner bestaat uit recursieve routines die steeds een knooppunt bekijken en beslissen of er dieper op ingegaan moet worden.

Een wat eenvoudiger probleem om te programmeren is het volgende: een routine die een deur moet openen, de er achterliggende ruimte moet verkennen en als er in die ruimte geen ongeopende deuren te vinden zijn, moet er een schilderij worden opgehangen. Zolang er echter gesloten deuren zijn, moeten die eerst worden geopend en moet de ruimte die erachter ligt worden verkend. De deuren worden van links naar rechts verwerkt en kan er geen misverstand ontstaan over welke deuren al afgehandeld zijn en welke niet. Wanneer het schilderij opgehangen is, moet de ruimte weer worden verlaten door de deur waardoor de ruimte werd betreden en moet de volgende deur, indien aanwezig, worden verwerkt:

- Open een deur.
  - In de ruimte erachter bevinden zich twee deuren.
  - Open de eerste deur.
    - In de ruimte erachter bevinden zich drie deuren.
    - Opent de eerste deur.
      - In de ruimte erachter bevinden zich geen deuren.
      - Hang een schilderij op.
      - Ga terug door de deur.
    - Open de tweede deur.
      - In de ruimte erachter bevindt zich één deur.
      - Open de deur.
        - In de ruimte erachter bevinden zich geen deuren.
        - Hang een schilderij op.
        - Ga terug door de deur.
      - Ga terug door de deur.
    - Open de derde deur.
      - In de ruimte erachter bevinden zich geen deuren.
      - Hang een schilderij op.
      - Ga terug door de deur.
    - Hang een schilderij op.
    - Ga terug door de deur.
  - Open de tweede deur.
  - etc...

Dit is nog tamelijk simpel en toch raak je al snel het spoor bijster. Door het inspringen van de code kan je nog enigszins zien op welk niveau er wordt gewerkt. Stel nu dat iemand een programmeerfout heeft gemaakt waardoor achter elke deur de ervoor liggende ruimte wordt herhaald. Dan blijft het programma dus eeuwig dezelfde deuren openen tot het of het besturingssysteem crashed.

Hier is ook weer duidelijk te zien dat steeds dezelfde handelingen optreden en die schreeuwen er gewoon om om geautomatiseerd te worden door middel van één mooie routine, die zichzelf steeds kan aanroepen. Dat gebeurt in C omdat deze taal recursieve aanroepen aan kan. Er komt een functie zoek_deuren() en we doen net alsof er gesloten deuren gevonden

zijn, door een waarde terug te geven afhankelijk van het aantal keer dat de functie is aangeroepen:

```
#include "c:\programs\c\inc\stdio.h"

/* Globale variabelen */
int teller;
int niveau;

main() {
        teller = 0;
        niveau = 0;

        /* Een functie als parameter van een functie */
        open_deur(zoek_deuren());
}


open_deur(deuren)
int deuren;
{
        niveau = niveau + 1;

        while (deuren) {
                /* Hier roept functie zichzelf aan */
                open_deur(zoek_deuren());

                deuren = deuren - 1;
        }

        niveau = niveau - 1;

        hang_schilderij();

        return;
}


int zoek_deuren () {
        int deuren;

        teller = teller + 1;

        switch (teller) {
                case 1:
                        deuren = 2;
                        break;
                case 3:
                        deuren = 3;
                        break;
                default:
                        deuren = 0;
                        break;
        }

        return (deuren);
}


hang_schilderij() {
        printf ("\nSCHILDERIJ op niveau %d.", niveau);
        return;
}
```

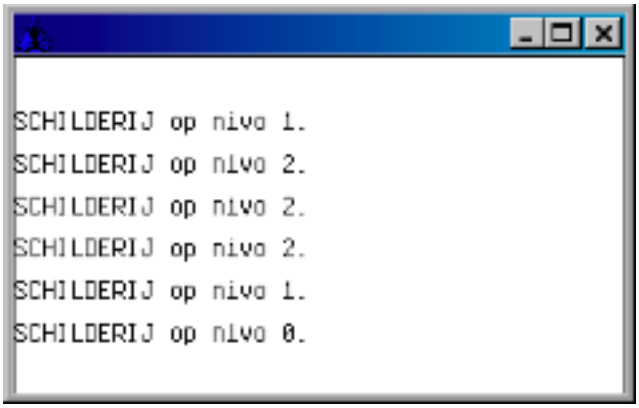
Voorbeeld 32: Programma met recursieve functies in C

Zoals gezegd wordt er in C steeds een nieuwe instantie van de functie geladen en de variabele DEUREN is daardoor steeds een *andere* variabele. Na vier keer OPEN_DEUR aangroepen te hebben, zijn er vier variabelen DEUREN met elk hun eigen waarde in het toegewezen geheugen aanwezig, elk op een ander geheugenadres.

Het onderstaande voorbeeld is in SL. Deze taal staat recursie niet toe. Het programma geeft een '*recursive external xcall*' foutmelding *op het moment* dat OPEN_DEUR zichzelf aanroept. Door gebruik te maken van de optie REENTRANT bij het .SUBROUTINE statement wordt recursie wel mogelijk, echter wordt steeds *dezelfde instantie* van de routine aangeroepen. Dus elke keer dat je in de routine aankomt, bevatten de variabelen de waarden die ze hadden toen de routine werd verlaten om zichzelf aan te roepen. Dat is oppassen geblazen, want alle variabelen waarvan de waarde veranderd, krijgen hun oude waarde niet meer terug als er naar een aanroeppunt wordt teruggekeerd:

```
.main   DEUREN

external function
        zoek_deuren,    d

common
        teller,d1
        niveau,d1

.proc
        open (1, i, 'tt:')

        teller = 0
        niveau = 0

        xcall open_deur(%zoek_deuren)

        close 1
.end


.subroutine open_deur, reentrant

deuren, d       ; parameter

external function
        zoek_deuren,    d

common
        niveau,d1
```

```
.proc
        niveau = niveau + 1

        while (deuren) begin
                xcall open_deur(%zoek_deuren)
                deuren = deuren - 1
        end

        niveau = niveau - 1
        xcall hang_schilderij

        xreturn
.end


.function zoek_deuren

record
        deuren,d2

common
        teller,d1

.proc
        teller = teller + 1

        using (teller) select
        (1),    deuren = 2
        (3),    deuren = 3
        (),     deuren = 0
        endusing

        freturn (deuren)
.end


.subroutine hang_schilderij

common
        niveau,d1

.proc
        display (1, 10,13, "SCHILDERIJ op niveau ",
        &       %string(niveau), ".")
        xreturn
.end
```



Voorbeeld 33: Programma met een recursieve subroutine in SL

De uitvoer is gelukkig precies gelijk aan het voorbeeld in C en dat wilden we ook. De taal RPG/400 staat helemaal geen recursie toe, zelfs niet door een programma dat zichzelf wil aanroepen. Maar een programmeur vindt voor alles een oplossing: het programma moet zichzelf klonen, dus een nieuwe instantie van zichzelf maken, en iedere kloon een andere naam geven. Dit is eigenlijk hetzelfde als wat er bij C gebeurt, echter nu niet in het geheugen maar op de harde schijf.

Aangezien op dit niveau geen globale variabelen mogelijk zijn moeten de variabelen als parameters worden doorgegeven. Maak een programma OPENDEUR en met *ENTRY parameters waaraan het NIVEAU, de TELLER en het aantal DEUREN kan worden doorgegeven:

```
*ENTRY          PLIST
                PARM            NIVEAU  20
                PARM            TELLER  20
                PARM            DEUREN  20
```

Stel in het RPG programma een Command Language-opdracht samen die het programma OPENDEUR moet dupliceren met de opdracht CRTDUPOBJ, met als nieuwe naam OPENDEURxx, waarbij 'xx' de inhoud van NIVEAU bevat:

```
IDUPOBJ DS
I I     'crtdupobj obj(OPENDE'          1  20 DO01
I I     'UR) fromlib(*LIBL) o'          21 40 DO02
I I     'bjtype(*PGM) tolib(Q'          41 60 DO03
I I     'TEMP) newobj('                 61 73 DO04
I I     'OPENDEUR'                      74 81 DO05
I                               S       82 830NIVEAU
I I     ')'                             84 84 DO06
I                                       74 83 PROGRM
```

Voorbeeld 34: CL-opdracht in een RPG/400 programma

Het leuke is dat de variabele PROGRM, de onderste variabele, de complete nieuwe programmanaam bevat. PROGRM beslaat namelijk de posities 74 t/m 83 van de datastructuur DUPOBJ. Stel nu dat de parameter NIVEAU de waarde 10 bevat, dan bevat de datastructuur DUPOBJ de volgende tekst:

```
crtdupobj obj(OPENDEUR) fromlib(*LIBL) objtype(*PGM) tolib(QTEMP)
newobj(OPENDEUR10)
```

De eerste keer dat het programma wordt gestart bevat de parameter NIVEAU de waarde 0. Vervolgens wordt er verwerkt wat er verwerkt moet worden en als er weer een deur wordt gevonden die open moet, moet er een duplicaat van het programma worden gemaakt, ten behoeve van de verwerking van het volgende niveau. Als eerste verhogen we NIVEAU met 1 en betreden we de lus:

```
        ADD     1       NIVEAU
DEUREN  DOWGT*ZEROS
```

Aangezien de functie ZOEK_DEUR in RPG niet als functie geprogrammeerd kan worden, wordt het een interne subroutine die de variabele AANTAL vult met het aantal te verwerken deuren. Vervolgens moet de opdracht worden uitgevoerd om het programma te dupliceren als het AANTAL groter is dan nul:

```
          EXSR    ZKDEUR
AANTAL    IFGT *ZEROS
          Z-ADD   83      LENGTH  155
          CALL    'QCMDEXC'                99
          PARM    DUPOBJ
          PARM    LENGTH
```

Als deze opdracht is uitgevoerd, is er van het lopende programma een duplicaat gemaakt onder een andere naam. Bij de CALL moet de foutindicator 99 worden geprogrammeerd want als het duplicaat al bestond, ontstaat er een foutsituatie. Deze moet worden afgevangen, anders crashed het programma. Vervolgens kan het nieuwe programma PROGRM worden gestat met als parameter het nieuwe niveau, de teller en het te verwerken aantal en volgt de rest van het programma:

```
          CALL    PROGRM                   99
          PARM    NIVEAU
          PARM    TELLER
          PARM    AANTAL
          ENDIF

          SUB     1       DEUREN
          ENDDO

          SUB     1       NIVEAU
          EXSR    HNGSCH

          SETON                                    LR
          RETRN
```

De routine HNGSCH is de afkorting van HANG_SCHILDERIJ omdat RPG/400 geen lange namen toestaat.

Bij een recursie van vier niveaus diep (0 t/m 3), bevinden zich in bibliotheek QTEMP op den duur drie extra programma's: OPENDEUR01, OPENDEUR02 en OPENDEUR03. Deze recursietruc is natuurlijk ook in andere talen te gebruiken met een SPAWN of SHELL opdracht. Of in een andere toepassing, zoals een dynamisch menu waarbij naar een volgend menu kan worden gesprongen en waarna weer in het oorspronkelijke menu moet worden teruggekeerd.

Globaal, locaal, algemeen, publiek, record en extern

(global, local, common, public, record and external)

Met deze voorvoegsels duiden we het gebruik van bepaalde objecten binnen een

programma, samengevoegde programma's (*bound programs*) of deelbare bibliotheken (*shareable library*, *dynamic link library (DLL)*). Deze objecten kunnen variabelen, literals, subroutines en functies zijn.

Local, record, static

Locale objecten zijn objecten die uitsluitend binnen het programma, de externe functie of externe subroutine beschikbaar zijn waar ze zijn gedeclareerd.

Global, common, public, external

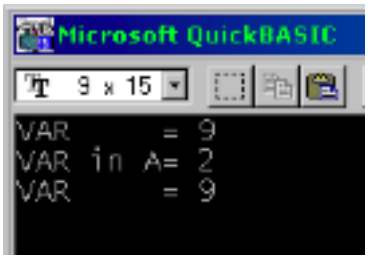
Deze typen staan buiten de externe routines en zijn van binnenuit de externe functies en routines te benaderen. De beschikbaarheid van deze typen is zeer afhankelijk van de programmeertaal. Neem bijvoorbeeld Synergy Language. Indien je een COMMON variabele in een externe subroutine of functie wilt gebruiken, moet je deze in de hoofdroutine declareren, ook al gebruik je hem daar niet. De variabele kan dan in alle externe routines worden gebruikt. In C gaat het heel anders. Een, in een eigen sourcefile, buiten alle functies gedeclareerde variabele is alleen toegankelijk door de functies in *diezelfde* sourcefile. Wanneer vanuit SOURCE1.C een variabele uit SOURCE2.C benaderd moet worden, dan moet in SOURCE1.C de variabele als EXTERNAL worden gedefinieerd. Sommige BASIC varianten kennen de type-aanduiding GLOBAL voor overal toegankelijke variabelen. Sommige talen gebruiken zelfs GLOBAL COMMON om aan te duiden dat ze van buitenaf wel heel erg toegankelijk zijn.

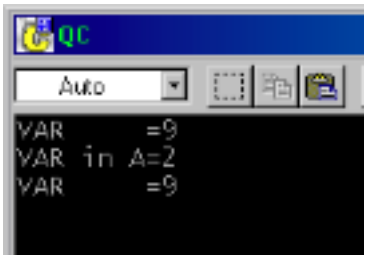
Voorbeelden

Hier volgen voorbeelden over het gebruik van locale, globale en externe variabelen.

```
DECLARE SUB A ()
' Demonstratie LOCALE
' variabele...

CLS
var = 9
PRINT "VAR     =";var
CALL a
PRINT "VAR     =";var
END

SUB a
  var = 2
  PRINT "VAR in A=";var
END SUB
```

```
/* Demonstratie LOCALE */
/* variabele           */

main() {
   int var = 9;
   printf("VAR     = %d\n", var);
   a();
   printf("VAR     = %d\n", var);
};

a() {
   int var = 2;
   printf("VAR in A= %d\n", var);
};
```

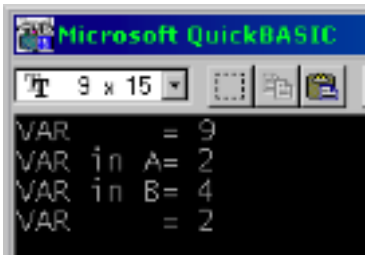




Voorbeeld 35: Locale variabelen in BASIC en C

```
DECLARE SUB A ()
DECLARE SUB B ()
' Demonstratie van
' GLOBALE en LOCALE
' variabelen...

COMMON SHARED var

CLS
var = 9
PRINT "VAR     =";var
CALL A
CALL B
PRINT "VAR     =";var
END

SUB a
  var = 2
  PRINT "VAR in A=";var
END SUB

SUB b
  STATIC var
  var = 4
  PRINT "VAR in B=";var
END SUB
```

```
/* Demonstratie van  */
/* GLOBALE en LOCALE */
/* variabelen...     */

int var;

main() {
  var = 9;
  printf ("VAR     = %d\n", var);
  a();
  b();
  printf ("VAR     = %d\n", var);
};

a() {
  var = 2;
  printf ("VAR in A= %d\n", var);
};

b() {
  int var;
  var = 4;
  printf ("VAR in B= %d\n", var);
};
```



Voorbeeld 36: Locale en globale variabelen in BASIC en C

Als de twee C-voorbeelden worden vergeleken, zie je dat bij het tweede programma de aanduiding INT is verdwenen voor VAR in de functies MAIN() en A(). In B() staat INT er nog wel en daardoor is VAR in functie B() een locale variabele en gebruiken de andere routines de andere variabele VAR, die welke helemaal bovenin de source is gedeclareerd. Bij het BASIC programma gaat het op dezelfde wijze. Omdat er een variabele VAR globaal is gedefinieerd, moet er in de subroutine aangeven worden dat het daar een locale variabele moet zijn, door middel van het voorvoegsel STATIC.

Iets anders wat bij sommige HLL's goed in de gaten moet worden gehouden is dat de definitie van een common of globale variabele overal gelijk blijft. Gebeurt dat niet, dan treden er zulke vreemde fouten op die niet licht zullen worden gevonden. Een computer programma gebruikt reeksen bytes om waarden in op te slaan. Als je nu een variabele, klant-omzet bijvoorbeeld, overal, in elke source, als integer hebt gedefinieerd en een INT blijkt te klein om de omzet van de klant in op te slaan, dan moet *overal* de type aanduiding INT in LONG veranderen:

```
/* Sourcefile main.c */

        long OMZET;

        main() {
                ...
                ...
        };


/* Sourcefile subr.c */

        external int OMZET;

        subrA () {
                ...
                ...
        };
```

Voorbeeld 37: Foute declaratie van globale variabelen in verschillende sources in C

Om dit soort fouten te voorkomen moeten de globale variabelen in een aparte sourcefile, een zgn. header- of includefile, worden geprogrammeerd. Overal waar deze variabele(n) dan nodig zijn, wordt dit bestand ingesloten en er is dan ook slechts één plaats waar de variabelen aangepast hoeven te worden, mocht er wat aan moeten veranderen.

3. ONTWERPEN

Nu je wat van de technische en tastbare kant van het maken van programma's hebt geproefd, wordt het tijd om de ongrijpbare kant, het ontwerpen, te behandelen. De bouw van een programma of een softwarepakket begint, zoals ook bij gebouwen en auto's het geval is, met een probleemstelling of idee. Vervolgens ga je, als enthousiaste programmeur, gelijk voor je beeldscherm zitten en beginnen met programmeren. Dit is volkomen normaal gedrag van programmeurs. Die hebben snel last van ongeduldige vingers.

Beheersing

De eerste versie van een programma doet altijd ongeveer wat je in gedachten had. Naarmate je meer ervaring krijgt, doet een eerste versie steeds meer wat je in gedachten had. Tijdens het programmeren heb je al weer nieuwe ideën opgedaan over allerlei bijkomende zaken die bij het bedenken van de eerste versie niet naar voren zijn gekomen. Je begint je programma uit te breiden met de nieuw bedachte zaken. Op den duur krijg je een onoverzichtelijk geheel waardoor je niet eens meer een plek kan vinden om zonder risico bij te bouwen. Je begint daarom maar helemaal opnieuw. Dat kan, omdat je nu weet wat er allemaal nog bij had gemoeten en je dat dus nu in één keer er aan vast kan programmeren. En zo modder je lekker verder. Ook ik heb het zo gedaan. En het is maar goed dat er op deze manier geen huizen of auto's meer worden gemaakt, want dat zou op een regelrechte ramp uitdraaien.

Houdt er rekening mee dat je dit 'hup, dat maken we effe' gevoel, die drang om gelijk te gaan programmeren, nooit, maar dan ook nooit meer kwijt zal raken. Je moet daarom leren jezelf af te remmen door altijd eerst keurig een ontwerp(je) te maken.

Keuzen

Even voor alle duidelijkheid: we bespreken hier het functionele ontwerp. Een specifiek technisch ontwerp is vaak te veel gevraagd en daarom worden de noodzakelijke technische aspecten opgenomen in het functionele ontwerp. Een functioneel ontwerp beschrijft wat de software over moet nemen van het menselijke handelen, hoe het dat moet gaan doen, waar en in wat voor omgeving het moet gaan werken en wat de doelgroep is. De keuze met welke ontwikkelomgeving in zee wordt gegaan, hangt sterk samen met de soorten software die ontwikkeld moeten worden. Administratieve programmatuur bijvoorbeeld, schrijf je makkelijk en snel in talen als COBOL, SL, RPG of BASIC, gebruik makend van een eventuele visuele tool. Software om apparaten te besturen, bijvoorbeeld drivers en besturingssystemen, schrijf je in C, C++ of Pascal en een assembleertaal. Het gebruik van verschillende ontwikkelomgevingen om één doel te bereiken komt steeds vaker voor, naarmate de complexiteit van de mogelijkheden toeneemt.

Houdt tevens voor ogen dat in een ontwerp nooit en te nimmer alles vastgelegd kan worden wat er geprogrammeerd moet gaan worden omdat we nooit álles vooraf kunnen bedenken. Zelfs wanneer je een programma helemaal voor jezelf maakt, zal je een maand

later iets veranderen om het te verbeteren met dingen die je van te voren niet hebt bedacht. Beschrijf een programma of software pakket dan ook niet uit den treure, maar neem de hoofdzaken op. Documenteer ingewikkelde berekeningen of procedures apart en voeg die als aanhangsel bij. Tijdens de uitvoering van een ontwerp vormt zich een beter beeld van de eisen die gesteld gaan worden aan de programmatuur, bij de programmeur en bij de opdrachtgever. Onbeantwoorbare vragen, problemen over hardware, nieuwe ideeën, nieuwe inzichten en nog vele andere zaken komen naar voren zodat al tijdens de bouw van de eerste versie van de programma's het ontwerp aan wordt gepast en gecorrigeerd, waarna er wordt begonnen de software weer aan het ontwerp aan te passen. Deze cyclus zal altijd blijven, de frequentie wordt echter steeds lager naarmate de software en de eisen die er aan worden gesteld steeds meer overeenstemmen met het gewenste eindresultaat.

Documentatie

Wanneer voor het ontwerpen van software gebruik gemaakt wordt van ontwerpsoftware, dan kan die software over het algemeen acceptabele documentatie genereren over de opbouw van relaties en gegevensstromen- en structuren. Deze documentatie kan als aanhangsel aan het ontwerp worden toegevoegd. Ook richtlijnen die voor de te bouwen programmaobjecten en bestanden gelden, denk aan de gebruikers interface, de naamgeving van objecten, versiebeheer en meer van dat soort zaken, behoren als aanhangsel aan een ontwerp te worden toegevoegd.

Laat de opdrachtgever goed zien dat er gedegen te werk gegaan wordt en zet wat zinvols op papier. Een goed ontwerp is beknopt, verhalend en verhelderend en laat tevens zien dat getracht is het hele concept van het te bouwen systeem te bevatten. Laat het vooral door de belanghebbende partijen lezen, en vergeet niet om de garantie te krijgen dat het ontwerp niet zonder uitdrukkelijke toestemming van de maker aan derden ter beschikking mag worden gesteld. Vraag of ze commentaar op het ontwerp willen geven en mee willen denken met de voorgestelde oplossingen zodat er een optimaal ontwerp kan ontstaan.

Het gebeurt vaak genoeg dat de opdrachtgever aan het eind zegt dat hij of zij zo'n programma nooit heeft gevraagd, het had heel anders gemoeten, we betalen niet en er klopt niets van, enzovoort. Wanneer het ontwerp ondertekend is, mag er vanuit worden gegaan dat de opdrachtgever het heeft gelezen en het grotendeels heeft begrepen. Maak een ontwerp daarom niet onoverzichtelijk door allerlei zaken door elkaar te beschrijven, van de hak op de tak te springen, waardoor de lezers het spoor bijster kunnen raken. Daar krijg je als ontwerper zelf ook last van wanneer er wat veranderd moet worden aan een bepaald onderdeel. Dan moet steeds het hele ontwerp doorgelezen worden om uit te vinden waar het onderdeel allemaal wordt genoemd. Splits een ontwerp bijvoorbeeld op in segmenten die elk een bepaald deel behandelen van een te bouwen programma of softwarepakket. De documenten kan je naar elkaar laten verwijzen. Met de hierboven beschreven werkwijze voorkom je later een hoop menigsverschillen en een hoop stress.

Dialoog

Een openlijke en regelmatige dialoog tussen bouwer en gebruiker is het allerhoogste goed.

Alleen dan bestaat er een kans op goede software, want als je niet met iemand er over praat, hoe denk je dan te weten wat iemand nou precies bedoeld? Omdat deze dialoog tussen automatiseerders en niet-automatiseerders een schijnbaar onoverkomelijk probleem schijnt te zijn, vanwege de technische inslag van de programmeur en de functionele, toepassingsgerichte inslag van de gebruiker die over het algemeen alleen over zijn eigen problemen en wensen kan praten. Als automatiseerder moet je daarom met de mensen mee kunnen klagen over de toestand waarin en waarmee ze moeten werken, je moet een zeer goed luisterend oor hebben en tegelijkertijd de problemen kunnen analyseren op hun emotionele en hun werkelijke waarde.

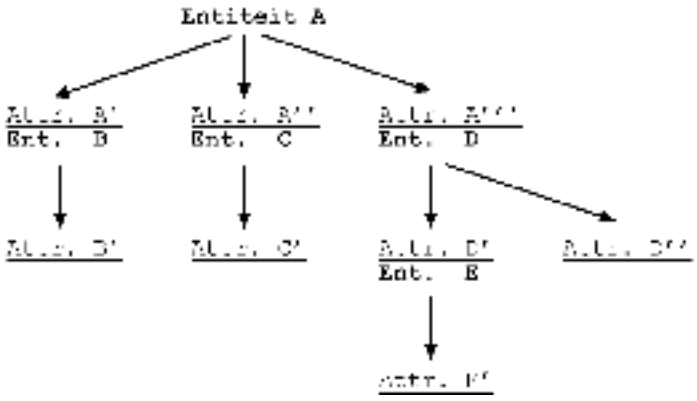
In meerdere pogingen aan de chronische communicatiestoornis een einde te maken zijn velen bezig geweest met het verzinnen van schema's om de werking van software 'begrijpelijk' te maken voor de buitenwereld. Schema's en systemen waar een normaal mens nog steeds niets van snapt. Er zijn er echter twee uit deze wereldwijde brainstorm overgebleven die een hoge 'begrijpelijkheidswaarde' hebben en daarom heel veel worden toegepast. De ene is het Entiteit Relatie Diagram (*Entity Relation Diagram*), afgekort tot ERD. De andere is het Gegevensstroom diagram (*Data Flow Diagram*), afgekort tot DFD. Deze grafische voorstellingen van relaties tussen entiteiten en gegevens kosten niet veel tijd om op te zetten en, nog belangrijker, het kost ook niet veel tijd om ze aan te passen aan nieuwe of gewijzigde situaties.

Entiteit-Relatie Diagram

Een entiteit is een beschrijving van een uniek 'iets' dat ontstaat door het verenigen van allerlei gegevens tot een geheel. De entiteit kan van alles voorstellen. Een stoel is bijvoorbeeld een entiteit, een wasmachine ook, een geestverschijning ook, een broek ook en een artikel ook. En let wel, er is maar één entiteit 'stoel', ongeacht hoe deze er uit ziet. En twee compleet verschillende wasmachines vallen tóch beide onder die ene entiteit 'wasmachine'. Entiteiten worden altijd genoemd in enkelvoud. Dus KLANT en niet KLANTEN, STOEL en niet STOELEN.

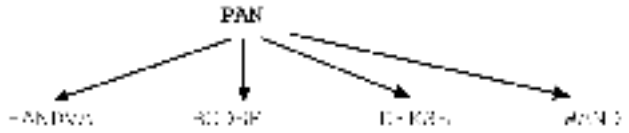
De verenigde gegevens die tesamen een entiteit vormen, noemen we de attributen. De kromming, de kleur, de omschrijving, de voorraad, de postcode, de woonplaats, etc.. De attributen die de entiteit STOEL vormen zijn onder andere poten, rugleuning, armleuningen, bekleding, breedte, hoogte en kleur. Attributen die bijvoorbeeld deel uitmaken van een entiteit ARTIKEL zijn de artikelcode, een omschrijving, prijs, voorraad, laatste inkoopdatum, enzovoort.

De attributen kunnen ook zelf weer entiteiten zijn die door andere attributen worden vormgegeven. En die attributen ook weer, en die daarvan ook weer. En zo zie je een boomstructuur ontstaan:



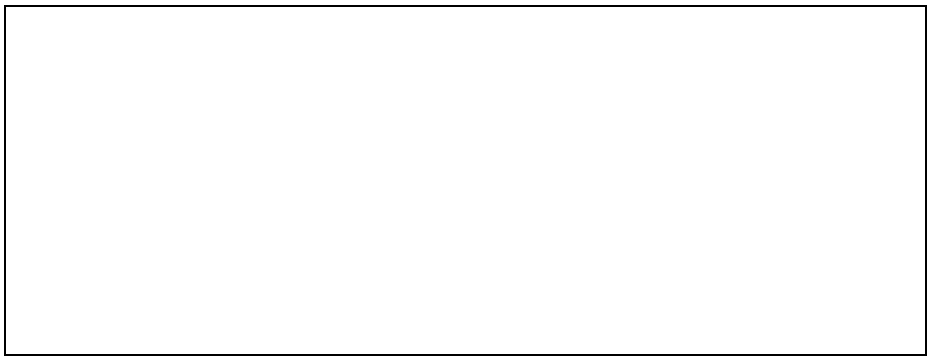
Voorbeeld 38: Entiteiten met entiteiten als attributen

In bovenstaand schema is attribuut A' tevens entiteit B. A" is entiteit C, A"' is entiteit D, enzovoort. Als je er even voor gaat zitten en er over nadenkt, zie je dat bijna alles om je heen tegelijkertijd entiteit en attribuut is. We gaan een voorbeeld uitwerken van een pan en zijn deksel.



Voorbeeld 39: Een pan heeft als attributen de handvatten, een deksel, een bodem en een wand

Noem dit maar niveau 1. De entiteit PAN is uitgesplitst naar vier attributen. Let wel, kleur en vorm zijn geen attributen van PAN maar van de individuele onderdelen. Het is namelijk niet de pan als geheel die een kleur heeft, het zijn de individuele onderdelen die gekleurd zijn. Onze hersenen geven de vorm de kleur die de boventoon voert. Zo kunnen we praten over 'de blauwe pan', terwijl die blauwe pan best een geel deksel en rode handgrepen kan hebben. Kortom, KLEUR is een attribuut van HANDVAT, van BODEM, van DEKSEL en van WAND. We gaan verder met de entiteit HANDVAT.



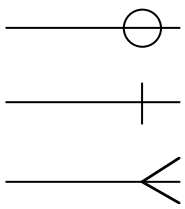
Voorbeeld 40: KLEUR als attribuut en entiteit

Hier komt naar voren dat KLEUR eigenlijk een entiteit is die als attribuut aanwezig is bij alle entiteiten die attributen zijn van entiteit PAN. De bedoeling van de uitsplitsing naar

entiteiten en attributen is dat attributen van een attribuut niet bij elke entiteit worden herhaald. De attributen van KLEUR neem je niet allemaal op bij de attributen van PAN. Het aanwezig zijn van identieke attributen bij de verschillende entiteiten noemen we *redundantie*. Het reduceren van redundantie van attributen noemen we *normaliseren*. Het normaliseren kent drie zogenaamde *normaalvormen*. De normaalvormen zijn eigenlijk niveau's die de mate van normalisatie bepalen. Er zijn uitgebreide verhandelingen over dit onderwerp te vinden in de bibliotheek. Stelregel is echter: zorg dat een attribuut-entiteit in het hele systeem slechts éénmaal voorkomt.

Relaties

Neem bijvoorbeeld de entiteit DEKSEL. Deze heeft als attributen de entiteiten HANDVAT en KLEUR en dus heeft DEKSEL een relatie met twee andere entiteiten. De relaties worden weergegeven door tussen de entiteiten horizontale of verticale lijnen te trekken. Diagonaal is niet gebruikelijk wegens het snel onoverzichtelijk worden van een ERD, teken dan een lijn met hoek van 90 graden. Aan deze lijnen kan je echter niet zien wat voor sóórt relatie er tussen de entiteiten bestaat. En dat is belangrijk om te weten als automatiseerder, want het illustreert voor een groot deel de toekomstige opbouw van gegevensbestanden. Want kan het DEKSEL één kleur hebben of meerdere kleuren? Heeft de PAN één, twee of meer handvatten? Om de antwoorden op deze vragen weer te kunnen geven heeft men een tekenmethode ontwikkeld voor ERD's waarmee de relaties en soorten relaties zichtbaar zijn. Deze tekenmethode is internationaal gangbaar:



Voorbeeld 41: Relatie-typen in een ERD

De horizontale lijnen stellen de relaties voor. Een relatie tussen twee entiteiten gaat altijd beide kanten op. De tekens op de lijn worden altijd aan de kant van de doel-entiteit geplaatst. De bron-entiteit ligt bij dit voorbeeld dus links en het doel rechts.

De cirkel staat voor nul. Daarmee wordt bedoeld dat de bron nul relaties kan hebben, of heeft, met het doel. De streep haaks op de relatie-lijn staat voor één. Dit geeft aan dat de bron een relatie heeft met één doel. De schuine strepen staan voor meer dan één. In dit geval heeft de bron een relatie met veel doelen. Bovenstaande tekens kunnen ook worden gecombineerd. Entiteiten worden weergegeven door een rechthoek, dat zien we verderop.

Voor textuele notatie van relaties, wordt het verhoudingsteken ':' gebruikt. Wanneer een bron nul relaties heeft met het doel, wordt dat als 1:0 genoteerd. Lees: één verhoudt zich tot nul, of één staat tot nul. En als één bron een relatie heeft met één doel wordt dat geschreven als 1:1, één staat tot één. En 1:n staat voor de relatie die één bron heeft met vele doelen. En n:m betekent dat vele bronnen een relatie hebben met vele doelen. Een entiteit kan ook een

relatie met zichzelf hebben. Denk hierbij eens aan een klantenbestand waarbij elke klantrecord een verwijzing heeft naar het record van het hoofdkantoor.

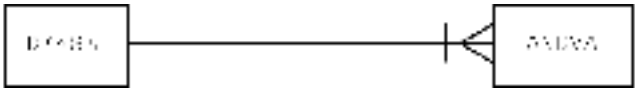
We gaan verder met de schematechniek. Een relatie tussen bijvoorbeeld DEKSEL en HANDVAT kan als volgt worden beschreven:

- Een deksel heeft minimaal **één** handvat.
- Een deksel *kan* **meerdere** handvatten hebben.

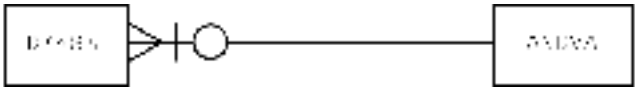
Andersom geredeneerd moet het óók kloppen:

- Een handvat kan op **nul** deksels voorkomen.
- Een handvat kan op **één** deksel voorkomen.
- Een handvat kan op **meerdere** deksels voorkomen.

We maken van beide beschrijvingen een ERD ter verduidelijking:

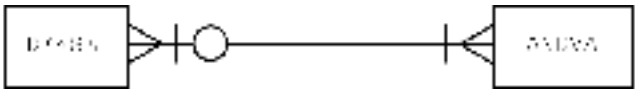


Voorbeeld 42: Relatie van entiteit DEKSEL in verhouding tot HANDVAT



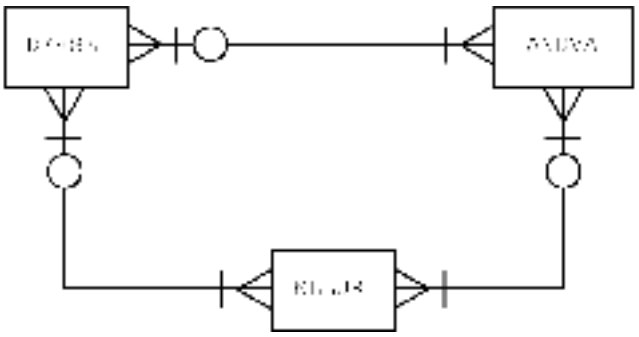
Voorbeeld 43: Relatie van entiteit HANDVAT in verhouding tot DEKSEL

Het mooie van deze tekentechniek is dat de twee onafhankelijke ERD's samengevoegd kunnen worden tot één schema:



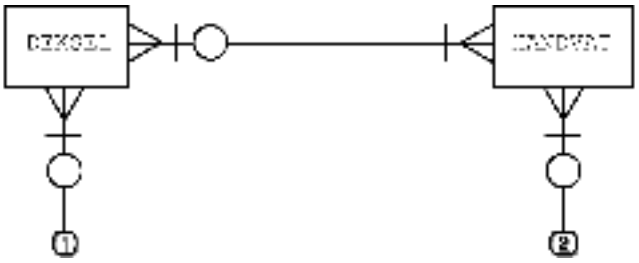
Voorbeeld 44: De relaties van DEKSEL en HANDVAT gecombineerd tot één ERD

Op dezelfde wijze kan ook de entiteit KLEUR worden toegevoegd. Deze heeft met de beide entiteiten dezelfde soort relatie. Een DEKSEL heeft minimaal één kleur, maar het kunnen er ook meer zijn. Hetzelfde geldt voor de entiteit HANDVAT. Ook moet er andersom geredeneerd worden. Hier volgt het ERD van de relaties tussen de drie entiteiten:



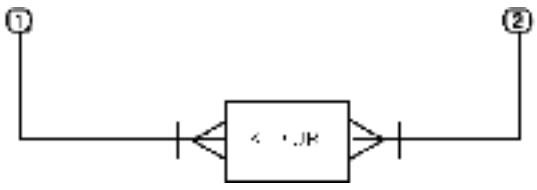
Voorbeeld 45: ERD van de entiteiten DEKSEL, HANDVAT en KLEUR

Dit is een klein ERD en vele kleine maken een grote. Wanneer je een heel software pakket gaat ontwerpen, kunnen er hele grote ERD's ontstaan die niet meer op één vel papier passen. Met een tekenprogramma, speciaal ontworpen voor het maken van ERD's of niet, kan de hele ERD getekend en uitgeprint worden. Alle A4-tjes moeten dan aan elkaar worden geplakt tot één grote tekening. De moeilijkheid hierbij is dat dat lastig uit te delen is. Beter is het te proberen doelgerichte ERD's te maken die op één A4 passen, met verwijzingen naar de andere delen van het ERD. Een verwijzing wordt gemaakt door aan het einde van een relatielijn een cirkel met een cijfer erin te tekenen. Een andere cirkel met datzelfde cijfer moet dan op een ander ERD terug te vinden zijn. Zo heeft de relatie tegelijkertijd een nummer toegewezen gekregen. Het is sowieso verstandig ook de andere relaties te nummeren of van een andere unieke identificatie te voorzien. De relatie kan dan apart gedocumenteerd worden. Er volgt een voorbeeld waarbij het bovenstaande diagram is gesplitst in twee delen, die naar elkaar verwijzen door middel van de besproken methode:



Voorbeeld 46: Deel 1 van de ERD van entiteiten DEKSEL, HANDVAT en KLEUR

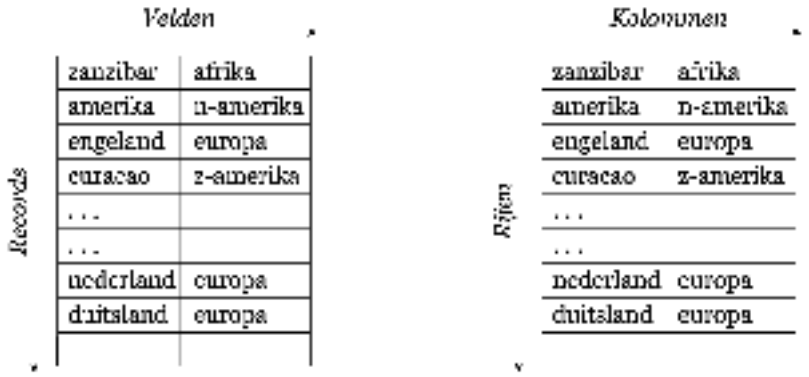
De relatie DEKSEL <--> KLEUR heeft nummer ① gekregen en de relatie HANDVAT <--> KLEUR het nummer ②. Als het eerste ERD via nummer ① wordt verlaten moet het tweede deel via nummer ① worden betreden:



Voorbeeld 47: Deel 2 van de ERD van entiteiten DEKSEL, HANDVAT en KLEUR

Verwerking van een Entiteit-Relatie Diagram

Als een ERD klaar is, wat moet er dan mee gebeuren? De bedoeling is dat de ERD als handleiding geldt voor de programmeur bij het creeren van bestanden en/of database-tabellen, waarbij de attributen de velden worden die tesamen een *record* in een bestand of *rij* (*row*) in een tabel vormen. Een veld wordt in databasetermen ook wel kolom (*column*) genoemd. Ook een werkblad in een spreadsheet bestaat uit rijen en kolommen en kan daardoor ook als kleine database worden ingezet. Hetzelfde geldt voor een tabel in bijvoorbeeld MS-Word. Als programmeur is het verstandig de termen record en veld te blijven hanteren:



Voorbeeld 48: Bestand of Tabel?

Een relatie tussen twee bestanden of tabellen wordt vastgelegd door het unieke kenmerk van de ene entiteit, de *primary key*, op te nemen als attribuut bij de andere entiteit, de *foreign key*. Dit is de manier waarop een entiteit als attribuut aan een andere entiteit wordt gekoppeld. Om te laten zien hoe je een entiteit omzet naar een record volgt hier een voorbeeld met DEKSEL en KLEUR. De vet gedrukte velden zijn de primary keys waarmee de gegevens uniek identificeerbaar zijn. De cursief gedrukte velden stellen de foreign keys van KLEUR voor:

```
record deksel
   code,        a12
   kleur,       a6
   handvat,     a12
   materiaal,   a12
   diameter,    d4
```

```
TYPE deksel
   code      AS STRING * 12
   kleur     AS STRING * 6
   handvat   AS STRING * 12
   materiaal AS STRING * 12
   diameter  AS INTEGER
END TYPE
```

Voorbeeld 49: Bestandslay-out (recordlay-out) in SL en QuickBASIC

```
create table deksel(code varchar (12), kleur varchar (6), handvat varchar
(12), materiaal varchar (12), diameter integer not null, primary key (code))
```

Voorbeeld 50: Bestandslay-out voor een REALdatabase in REALbasic

Het attribuut CODE bevat de unieke identificatie van een DEKSEL. KLEUR, HANDVAT, MATERIAAL en DIAMETER zijn de attributen van DEKSEL. Attribuut KLEUR bevat de unieke sleutel waarmee in het bestand KLEUR een record te vinden is met

additionele gegevens betreffende de kleur. Hieronder volgt de recordlay-out van het bestand KLEUR. Het vet gedrukte veld is weer de primary key:

```
structure KLEUR {
        char code[6];
        int rgb[3];
        char htm[3,2];
        int cmy[3];
        char pant[15];
        char naam[32];
};
```

```
R KLEUR
  KLCODE     6A
  KLRGBR       3S 0
  KLRGBG       3S 0
  KLRGBB       3S 0
  KLHTMR       2A
  KLHTMG       2A
  KLHTMB       2A
  KLCMYC       3S 0
  KLCMYM       3S 0
  KLCMYY       3S 0
  KLCMYK       3S 0
  KLPANT      15A
  KLNAAM      32A
```

Voorbeeld 51: Recordlay-out in C en RPG/400

```
create table kleur(code varchar (6), rgb_red integer not null, rgb_green
integer not null, rgb_blue integer not null, htm_red varchar (2) , htm_green
varchar (2) , htm_blue varchar (2), cmyk_cyan integer not null, cmyk_magenta
integer not null, cmyk_yellow integer not null, cmyk_black integer not null,
primary key (code))
```

Voorbeeld 52: Recordlay-out voor een REALdatabase in REALbasic

Bij de definitie van KLEUR in de taal C in bovenstaand voorbeeld zie je dat RGB als [3] is gedefinieerd. Dat is de notatie voor een *array* van 3 elementen, waarbij het eerste element de waarde voor rood bevat, het tweede de waarde voor groen en het derde de waarde voor blauw. Bij een database zijn arrays over het algemeen niet mogelijk, daar moet elk element als apart veld wordden gedefinieerd. Dat is de reden in het OS/400 bestand en de REALdatabase de arrays niet voorkomen en alle velden individueel zijn gedefinieerd. Het veld KLRGBR staat voor RGB_RED, KLRGBB staat voor RGB_BLUE, etc.. De korte namen hebben te maken met een beperking in de lengte van veldnamen, maximaal 6 tekens, door de *Data Description Specifications* van OS/400. Bij zulke korte veldnamen is het gebruikelijk de eerste twee tekens te wijden aan de naam van het bestand. In dit geval is gaat het om KLEUR en worden de eerste twee letters, die zijn duidelijk genoeg, genomen als *prefix* voor de veldnamen.

We hebben het nu toevallig over kleuren en daar valt wel wat meer over te vertellen. Rood, Groen en Blauw zijn *additieve* primaire kleuren. Ze worden samengevoegd, opgeteld, om wit licht te maken (op beeldschermen, tv's, etc). HTML kleuren zijn ook RGB kleuren, echter in een hexadecimale notatie (*zie het hoofdstuk over Talstelsels*). Cyaan, Magenta en Geel zijn de *subtractieve* primaire kleuren. Het zijn de kleuren die in de natuur voorkomen doordat alles in de natuur een bepaalde mate van lichtstraling *absorbeert*. Het restant wordt door de objecten (bladeren, verf, boeken, bloemen, stenen, flessen, etc.) teruggekaatst en bereikt ons netvlies. Een rode roos is dus eigenlijk een witte roos die een deel van het licht opslurpt en licht in het rode deel van het spectrum terugkaatst. In de grafische wereld kom je vaak de afkorting CMYK tegen. De C, M en Y staan voor de genoemde kleuren, de K staat voor BLACK en is ten behoeve van het drukken toegevoegd om mooi zwart te krijgen, want dat lukt niet met CMY alleen. Zwart is een stof die al het licht absorbeert en niets terugkaatst

en daarom dus *geen* kleur is. Daarom wordt er in het drukproces vaak een hele donkere kleur blauw aan het zwart toegevoegd.

Normaliseren

Wanneer alle benodige informatie voor een te bouwen systeem is vergaard en gedocumenteerd, moet deze informatie per soort worden gegroepeerd. Alles wat met een artikel te maken heeft samenvoegen, alles wat met een klant te maken heeft samenvoegen, en zo verder. Als dat is gedaan, komt vanzelf naar voren dat sommige identieke zaken meerdere malen opduiken, onder dezelfde naam of met een geheel andere terminologie. Probeer vooral op dat laatste te letten, zodat er geen dubbel werk wordt gedaan omdat er op twee verschillende manieren over is gesproken. Het filteren van dubbele zaken en het reduceren ervan tot één nieuwe entiteit noemen we *normaliseren*.

Neem het eenvoudige voorbeeld van PAN. KLEUR komt daar wel vier keer voor: bij DEKSEL, HANDVAT, WAND en BODEM. Wanneer KLEUR bij alle vier genoemde entiteiten volledig zou worden opgenomen in de bestandslay-out, spreken we van *redundantie*. Redundantie is vaak een probleem omdat het extra geheugencapaciteit vraagt en het programmeren bemoeilijkt. Stel dat KLEUR vier keer volledig, dus alle velden RGB*, HTM*, CMY*, etc., zou zijn opgenomen bij de verschillende entiteiten en de RGB-code blijkt op een gegeven moment niet te kloppen, terwijl die in alle velden dezelfde is. In de eerste plaats is er onnodig schijfruimte ingenomen en in de tweede plaats moet de RGB-code vier keer worden aangepast. Het mag duidelijk zijn dat de foutkans in deze situatie erg groot is; er hoeft er maar één te worden vergeten of fout te worden aangepast en je hebt een probleem. Kortom, in plaats van KLEUR vier keer volledig op te nemen bij de entiteiten, moet er een nieuwe entiteit KLEUR worden gecreeerd en moeten er relaties naar de andere entiteiten gelegd. Hoe dat in zijn werk is gegaan hebben we hierboven gezien, daar is KLEUR als foreign key in de bestanden opgenomen.

Als nú de RGB-code verkeerd blijkt te zijn hoeft die maar op één plaats te worden aangepast. Overigens, niet alle dubbele velden hoeven een storende redundantie te zijn die weggewerkt moet worden. Als bijvoorbeeld de adresgegevens van een klant volledig in de orderkop worden overgenomen is dat wel redundantie van gegevens, maar met een bepaald doel: het verzamelen van historische gegevens. Niet alleen voor het eigen systeem, maar misschien ook wel voor de belastingdienst. Ook de verkoopprijs van een artikel moet worden opgenomen in de orderregel want dat is de prijs die wordt gefactureerd en in de historie wordt opgeslagen. Zo zijn er meer zaken waarbij redundantie zelfs vanzelfsprekend en misschien wel noodzakelijk is.

Gegevensstroom Diagram

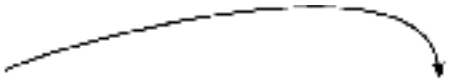
Nu het licht verteerbare theoretische deel over het vastleggen van entiteiten en de relaties tussen de entiteiten achter de rug is, wordt het tijd deze relaties en entiteiten vorm te geven door er theoretische gegevens aan te koppelen, waardoor het geheel wat meer gestalte krijgt. De relaties tussen gegevens en *processen* die de gegevens creeren, bewerken of verwijderen worden vastgelegd in het *Data Flow Diagram*, afgekort het DFD. De entiteiten geef je in

DFD weer door twee horizontale strepen met de naam van de entiteit er tussenin:

DEKSEL

Voorbeeld 53: Weergave van een entiteit in een Data Flow Diagram

Met een DFD geef je weer uit welke entiteit welke gegevens naar welke andere entiteit, of dezelfde eintiteit, worden verplaatst of gecopiëerd. De relaties uit de ERD worden de gegevensstromen van het DFD. Een gegevensstroom geef je weer door middel van een lijntje, kronkelig of gebogen, met een pijltje aan het einde van de gegevensstroom dat aangeeft waar de gegevens naar toe stromen:

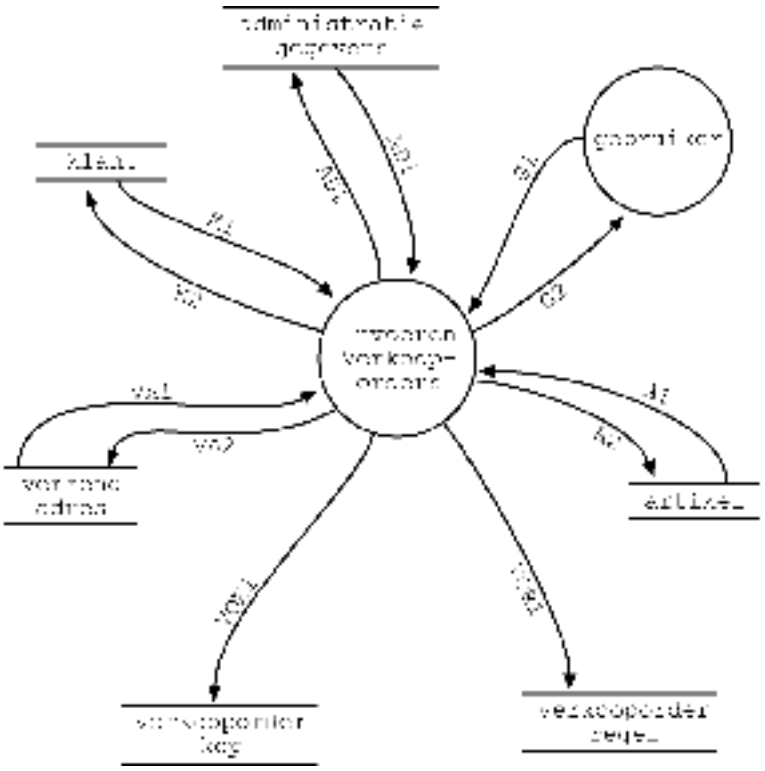


Voorbeeld 54: Weergave van een gegevensstroom in een DFD

Gegevens verplaatsen, veranderen en copiëren zichzelf niet zomaar. Het zijn processen zoals computerprogramma's, of externe processen als de invoer van gegevens door mensen, die de gegevens gebruiken of nieuwe gegevens maken. Processen worden weergeven door een cirkel met de naam van het proces in het midden:

Voorbeeld 55: Weergave van een proces in een DFD

Laten we dit proces eens uitbreiden met gegevensstromen van en naar entiteiten en externe processen. We definiëren een compact verkooporderssysteem. Er hoort een entiteit ARTIKEL bij, een entiteit KLANT, een entiteit VERZENDADRES, de entiteiten VERKOOPORDER_KOP en VERKOOPORDER_REGEL. Tevens wordt het proces bestuurd door gegevens als basisvaluta en BTW percentages die uit een entiteit met stuurgegevens komen: ADMINISTRATIE_GEGEVENS. Ook is er een proces gekoppeld dat het ingeven van orders door mensen voorsteld. Hier volgt de DFD van het systeem:



Voorbeeld 56: Een DFD van een verkooporderssysteem

In dit DFD hebben ook alle gegevensstromen een eigen kenmerk of identiteitscode gekregen. De gegevensstromen worden aan de hand van de identificatie gedocumenteerd. In een groot systeem, dat bestaat uit vele bestanden en programma's, kan een bepaald soort gegevensstroom meerdere malen voorkomen in verschillende disciplines. Een klantcode is nodig bij een verkooporder en ook bij een verzendadres. Van de gegevensstroom '*haal klantcode*' kan een subroutine geprogrammeerd worden die op meerdere plaatsen inzetbaar is en refereert aan de identicifatie van de gegevensstroom. Wanneer alle documentatie 100% up-to-date is met de actuele software, is ook snel terug te vinden waar een gegevensstroom gebruikt wordt. Wat nu volgt is een voorbeeld van een manier van documenteren van individuele gegevensstromen, op volgorde van identificatiecode:

AD1 Ophalen van administratiegegevens:
- ADM.Eerstvrije ordernummer
- ADM.BTW percentages
- ADM.Etc...

AD2 Uitvoeren van:
- ADM.Eerstvrije ordernummer

G1 Ontvangen van gebruikersgegevens:
- Klantcode
- Artikelcodes
- Aantallen
- Printerkeuze
- Etc...

G2 Uitvoer naar gebruiker van:

- KLANT .NAW (Naam Adres Woonplaats)
- ARTIKEL.Prijs, korting
- Berekende ordertotalen
- VZD.NAW
- Ordernummer
- Melding van overschrijding van zijn kredietlimiet
- Etc...

K1 Ophalen van klantenrecord:
- KLANT.Code
- KLANT.NAW
- KLANT.Telefoonnummer
- Etc...

K2 Uitvoeren van:
- KLANT.Te factureren bedrag
- KLANT.Laatste verkoopdatum
- Etc...

VA1 Ophalen van verzendadresgegevens:
- VZD.NAW

VA2 Uitvoeren van:
- VZD.Aantal maal gebruikt

A1 Ophalen van artikelgegevens:
- ARTIKEL.Code
- ARTIKEL.Omschrijving
- ARTIKEL.Prijs
- ARTIKEL.Technische voorraad
- ARTIKEL.Etc...

A2 Uitvoeren van:
- ARTIKEL.Laatste verkoopdatum
- ARTIKEL.Technische voorraad
- ARTIKEL.Etc...

VOK1 Uitvoeren van verkooporderkopgegevens:
- VOK.Klantcode
- VOK.Ordernummer
- VOK.Verzendadrescode
- VOK.Leverdarum
- VOK.Etc...

VOR1 Uitvoeren van verkooporderregelgegevens:
- VOR.Klantcode
- VOR.Ordernummer
- VOR.Regelnummer
- VOR.Artikelcode
- VOR.Aantal
- VOR.Prijs
- VOR.Etc...

De punt tussen de bestandsnaam en de veldnaam is het algemeen aanvaarde scheidingsteken voor attributen. Aan het attribuut CODE in de entiteit KLANT refereer je met

de notatie 'KLANT.CODE'.

Bestanden

Alles wat op een harde of zachte schijf bewaard kan worden door middel van een computer heet een bestand. Er zijn tekst-bestanden, gegevens-bestanden, programma-bestanden, opdracht-bestanden, etc.. Op de keper beschouwd is een bestand gewoonweg een berg bytes voorzien van een begin- en een eindkenmerk. De berg bytes wordt echter in een bepaalde structuur op schijf bewaard. Er zijn gestandaardiseerde structuren zoals Tekstbestanden, JPEG, GIF en PICT en er zijn programma-eigen structuren zoals bestanden van de programma's Word, Freehand, Photoshop, Excel, etc.. De bytes in een afbeeldingsbestand zijn heel anders gestructureerd dan de bytes die met elkaar een personeelsbestand vormen.

Het onderscheid tussen de verschillende gestructureerde hoeveelheden bytes wordt gemaakt door een type-aanduiding toe te passen. Elk systeem heeft zijn eigen wijze om de type-aanduiding op te slaan. Bij Windows zit de aanduiding in de bestandsnaam verwerkt, wat heel gevaarlijk kan zijn als de aanduiding veranderd en daardoor voorwend een ander soort bestand te zijn. Andere methoden zijn om het bestandstype in het bestand zelf op te nemen, naast de gegevens die het bestand al bevat, eventueel met een unieke code van het programma dat het bestand heeft gemaakt. Apple doet dat met het MacOS en IBM doet dat met OS/400. Die aparte reeks bytes waarin deze gegevens worden opgeslagen wordt ook wel de *resource fork* genoemd en het deel waar de data zich bevindt heet de *data fork*. Wat ook veel wordt gedaan, en dat zie je vooral bij afbeeldingsbestanden, is een aantal bytes aan het begin in het bestand op te nemen, de bestands*header*, die aangeven met wat voor soort bestand het programma te maken heeft. Een programma moet dan wel zo geprogrammeerd zijn dat het op zoek gaat naar een bestandsheader. Ook worden andere attributen van een bestand niet direct zichtbaar voor een gebruiker aan het bestand gekoppeld, zoals bijvoorbeeld de laatste backupdatum of een signaal dat het bestand gewijzigd is sinds de laatste backup. Deze gegevens worden echter niet in het bestand opgeslagen maar elders in het systeem.

Er zijn drie bestandsstructuren waarop alle bestanden zijn gebaseerd, namelijk de sequentiële bestanden, de geïndexeerde sequentiële bestanden en de direct toegankelijke bestanden. De laatste twee typen zijn alleen toegankelijk door óf zelf een bestandsbeheerssysteem te programmeren of gebruik te maken van reeds bestaande software. Een bestandsbeheerssysteem kan deel uit maken van het besturingssysteem, bijvoorbeeld RMS (Record Management Services) onder VMS of het bestandssysteem DB2 onder OS/400. Het kan ook géén systeem zijn, er kunnen dan alleen sequentiële bestanden worden gemaakt en worden gelezen, zoals het standaard bestandssysteem van Windows, MacOS of Unix. Het kan echter ook een product van derden zijn, zoals een DBMS (Database Management System) als Oracle, MS-Access, Filemaker of MySQL, of een in de programmeertaal verwerkt systeem zoals in REALbasic een REALdatabase, of een los te koop zijnde set C-subroutines en functies die je mee moet linken aan je programma.

Sequentieel bestand

Sequentële bestanden (*Sequential file*) zijn bestanden waarbij de opgeslagen gegevens vanaf het begin van het bestand moeten worden ingelezen. Bij sequentiële bestanden is de recordlengte over het algemeen variabel, denk maar aan een simpel tekstbestandje waarbij elke paragraaf een record is. Elke paragraaf heeft een andere lengte en daarom moet elk record een herkenbaar einde hebben. Dit teken of deze tekens zijn normaal niet zichtbaar in de verwerkingsprogramma's en verschillen per besturingssysteem. Onder MacOS is het ASCII 13 en onder DOS en Windows is het ASCII 13 en ASCII 10 direct na elkaar.

Het probleem met sequentiële bestanden is dat je niet zomaar gegevens kunt veranderen zonder eerst het hele bestand van begin tot eind in te lezen. Neem we als voorbeeld een tekstbestand of een afbeeldingsbestand. Dat wordt geheel ingelezen en vervolgens aan de gebruiker ter beschikking gesteld. Die wijzigt wat in het bestand door schrijffouten te corrigeren of kleuren te veranderen en vervolgens moet het hele bestand óver de vorige versie heen worden teruggeschreven. Het plaatselijk bijwerken van een sequentieel bestand is vrijwel onmogelijk. De wijzigingen zijn vaak erg groot en verdeeld over het hele bestand: je voegt stukken tekst tussen, schuift tekst op, maakt plaatjes kleiner of groter, je veranderd de kleur rood in paars, etc..

Bestanden kunnen heel groot worden, zoals bijvoorbeeld een gedigitaliseerde foto van 100 megabytes. Er zijn inmiddels slimme methoden uitgevonden om dit soort hele grote bestanden niet in zijn geheel in het geheugen te hoeven laden om het te kunnen bewerken; dat zou de gemiddelde computer gewoonweg niet aankunnen. Die methoden zijn echter niet gestandaardiseerd want ze hebben te maken met de inhoudelijke structuur van een bestand en moeten zelf door de programmeur worden geprogrammeerd; op het internet is er voldoende informatie over te vergaren.

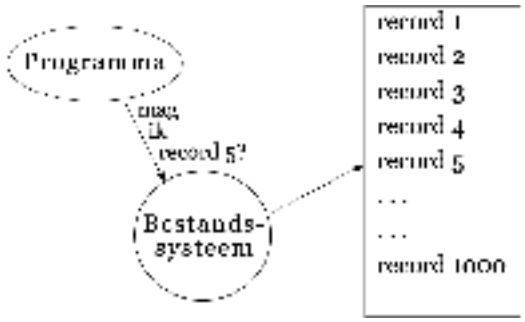
Record-adresseerbaar bestand

Andere namen voor dit type bestand zijn Willekeurig Toegankelijk (*Random Access file*), Relatief (*Relative file* of *Record Address file*) of Direct Toegankelijk (*Direct Access file*). Bij deze bestanden hoef je niet bij de eerste byte te beginnen met lezen om gegevens te vinden die zich een paar duizend bytes verderop in het bestand bevinden. Dat komt doordat alle records in het bestand dezelfde lengte hebben. En dat kan alleen als een leeg alfanumeriek veld niet leeg is maar spaties bevat, het neemt dan altijd dezelfde ruimte in. Vanwege die vaste recordlengte hebben de records geen herkenningsteken(s) nodig om het einde van het record aan te geven. Een record-adresseerbaar bestand opent de mogelijkheid het gootste deel van het bestand gewoon op schijf te laten staan terwijl je gegevens aan het bewerken of creëren bent, dit in tegenstelling tot een simpel sequentieel bestand. Juist omdat elk record uit een gelijk aantal bytes bestaat, is uit te rekenen waar zich bijvoorbeeld het 100e record bevindt. Dit record lees je in je programma in, je bewerkt het en je schrijft het op precies dezelfde plaats terug, zonder dat de rest van het bestand daarvoor ingelezen hoeft te worden. Neem als voorbeeld het bestand KLEUR met de eerder getoonde RPG/400 recordlay-out. Het record beslaat precies 80 bytes. Als recordnummer nemen we simpelweg de omgerekende HTML-kleurcode, die loopt van hexadecimaal "000000" tot en met hexadecimaal "FFFFFF" (*zie Talstelsels*). De decimale waarde van FFFFFF is 16.777.215. Met recht kan je dus spreken van miljoenen kleuren en een heel groot bestand: 16.777.215 * 80 bytes = 1.342.177.200 bytes. Dat komt neer op (((1342177200/1024 = 1310720 Kilobytes) / 1024 = 1280 Megabytes) / 1024) = 1,25 Gigabytes.

Recordnummer NUL kan niet bestaan, net als delen door NUL niet kan. Daarom moet er bij de omgerekende kleurcode 1 worden opgeteld om het juiste recordnummer te krijgen. De kleurgegevens voor helderrood bevinden zich op recordnummer "FF0000", wat omgerekend naar het decimale talstelsel de waarde 16.711.680 vertegenwoordigd. Vermeerder dat met 1 en je krijgt *recordnummer* 16.711.681. Het record begint echter op *bytenummer* (16.711.680 * 80) + 1 = 1.336.934.401. Het bytenummer kan eenvoudig uitgerekend worden met de volgende formule: ((recordnummer - 1) * recordlengte) + 1.

In de meeste hogere programmeertalen hoef je dat allemaal niet zelf te programmeren en biedt het bestandssysteem de mogelijkheid om op recordnummer het bestand te benaderen.

Hier volgt een schematische weergave van een record-adresseerbaar bestand en hoe het werkt:



Voorbeeld 57: Schematische weergave van de werking van een Record-Adresseerbaar bestand

Index-Sequentiële Toegangs Methode

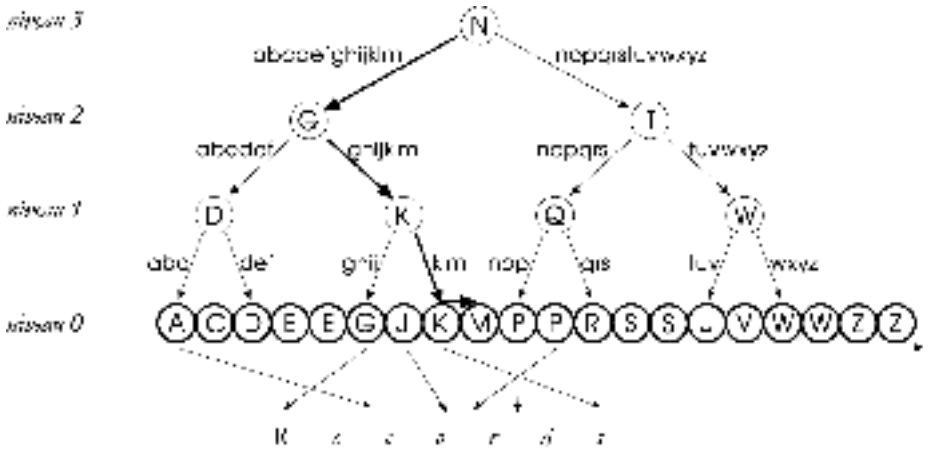
Het index-sequentiële bestand, in het kort ISAM (*Indexed Sequential Access Method*), bestaat uit twee delen, een sequentieel deel waar de gegevens in zijn opgeslagen en een of meer record-adresseerbare delen, de indexen, waarin elk record een uniek kenmerk bevat van elk record uit het sequentiële deel en een verwijzing naar de plaats op schijf van de bijbehorende records in het sequentiële deel. Een index is gesorteerd op volgorde van het unieke kenmerk, de sleutel of *key*, waardoor gegevens in het sequentiële deel snel gevonden kunnen worden. Daarom is het zaak uit de beschikbare velden in een record een goed uniek kenmerk te kiezen, waaruit volgt dat een index over het algemeen een vaste recordlengte heeft terwijl het gegevensdeel dat niet hoeft te hebben omdat via de index naar gegevensrecords wordt gezocht. Behalve de genoemde gegevens bevat een indexbestand ook nog twee andere verwijzingen, te weten een verwijzing naar het volgende indexrecord dat volgens de sortering het volgende indexrecord moet zijn en een verwijzing naar het vorige indexrecord. Deze verwijzingen zijn de schijfadressen van de schijf waarop het bestand ligt opgeslagen waar de gegevensrecords te vinden zijn. Voor het gebruik van een ISAM zijn aparte taalelementen nodig die met een index om kunnen gaan.

Een sleutel van een gegevensbestand kan een klantcode zijn die op alfabetische volgorde wordt gesorteerd. Een andere sleutel is bijvoorbeeld een datum die op aflopende volgorde,

van 9 naar 0, wordt gesorteerd om steeds snel de jongste datum te kunnen vinden. Neem we als voorbeeld een bestand dat alle adressen bevat van de inwoners van een grote stad. Dat kunnen tienduizenden adressen zijn. Als het een sequentieel of record-adresseerbaar bestand is, moet het vanaf het begin worden doorlopen om een adres te vinden: zo heb je geluk als het adres zich aan het begin bevindt en pech als het aan het einde zit.

Boomstructuur

Een index met alleen een de sleutelgegevens van een record is natuurlijk nog te traag. Er moet veel te lang worden gezocht omdat de index sequentieel moet worden doorlopen. Daarom worden indexen geïmplementeerd als een boomstructuur en bestaan derhalve uit meerdere niveaus (*levels*). Uit de bron groeien bijvoorbeeld twee takken, waaruit weer een tweetal takken groeien, tot het punt waarop de boom nog slechts zeer langzaam groter wordt. Bij een index bevatten deze laatste kleine takjes de verwijzingen naar de individuele records. Een ISAM met weinig records heeft vaak een index van twee niveaus en een met veel records al gauw drie tot vier niveaus. Nu volgt een schema van een boomstructuur met een geïndexeerd alfabet als voorbeeld:



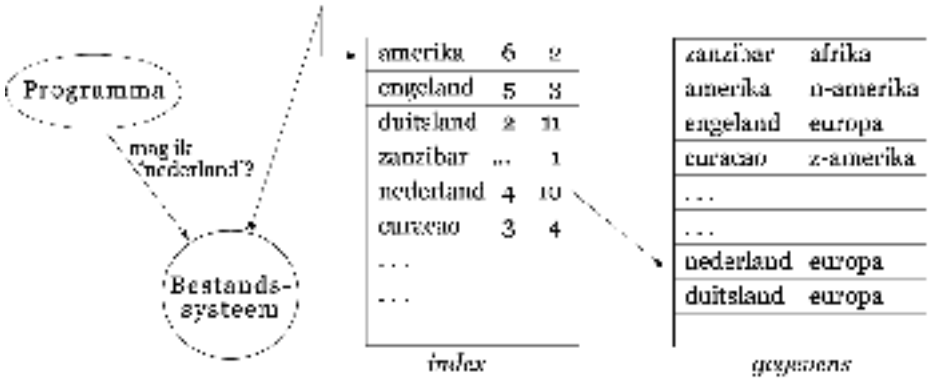
Voorbeeld 58: Een boomstructuur met zoekpad naar sleutels beginnend met een 'M'.

Aan de verdikte pijlen is te zien dat in vier stappen een record gevonden kan worden tegen acht stappen als het bestand sequentieel zou worden doorlopen op niveau nul.

Het andere voordeel van een index is dat als er een record wordt toegevoegd, het nieuwe indexrecord gewoon aan het eind kan worden toegevoegd. Vervolgens moet de plaats worden bepaald waar, volgens de sortering die bij de index gedefinieerd is, de sleutel tussengevoegd zou moeten worden. In bovenstaand schema is er nog geen gegevensrecord waarvan de sleutel met een (L) begint, en juist deze willen we tussenvoegen. Van het record ervóór, (K), wordt de verwijzing die naar het volgende record (M) wijst aangepast: deze verwijzing moet nu naar de positie in de index van het nieuwe record (L) wijzen. De oude verwijzing naar (M) moet in het nieuwe indexrecord worden opgenomen zodat de ketting niet wordt verbroken. Vervolgens moet indexrecord (M) gelezen worden en wordt de verwijzing naar (K) vervangen door een verwijzing naar het nieuwe record (L). De oude verwijzing naar (K) moet in (L) worden opgenomen Daarna is de ketting weer compleet: K wijst naar L, L wijst

naar M en de terugweg klopt ook. Bij het verwijderen van een record gebeurt precies het omgekeerde. Uiteraard is het mogelijk dat een sleutelwaarde wordt gewijzigd waardoor deze op een andere plaats in de sortering terecht moet komen. Dan hoeft het bestandssysteem slechts de verwijzingen aan te passen en het record is 'gesorteerd'.

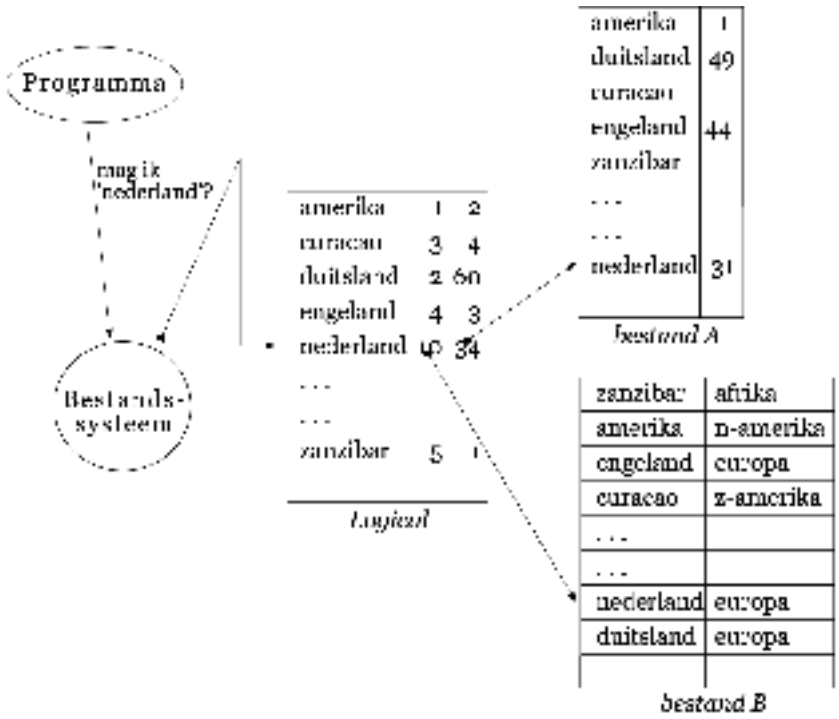
Hieronder volgt een vereenvoudigd voorbeeld van een ISAM van een landenbestand, met een index op landnaam, gesorteerd van laag naar hoog, wat wil zeggen van A B C D..... naar Z. De eerste kolom van de index bevat de *sleutelwaarde* ("amerika", etc), de tweede kolom bevat het recordnummer van het volgende indexrecord dat voldoet aan de sortering. De derde kolom van de index bevat de verwijzing naar een record in het sequentiële deel van de ISAM waar te vinden is in welk werelddeel het land ligt:



Voorbeeld 59: Schematische weergave van de werking van een Index-Sequentieel bestand

Logisch bestand, Bestand met een beperkt zicht

Dit zijn bestanden waarvan de records zijn opgebouwd uit één of meer velden van één of meer van de vorige twee genoemde typen bestanden. Een *logical*, of *view*, heeft eigenlijk dezelfde functie als een indexbestand van een ISAM, terwijl het zelf ook weer een soort ISAM is, echter met het grote verschil dat een logical permanent gekoppeld kan worden aan bepaalde gegevensvelden uit het originele bestand. Een logical die aan gegevensvelden van twee of meer verschillende bestanden is gekoppeld heet een *join logical*. In een join logical moeten de keys uit minstens twee gekoppelde bestanden overeenkomen. Zo is het mogelijk de velden naam en adres uit een klantenbestand te koppelen aan het orderbedrag van de orders van die klant door de klantcode van beide bestanden als koppeling te definiëren. Tegelijkertijd kan in de join logical aan de landnaam uit het landenbestand worden gekoppeld door de landcode uit het klantenbestand te koppelen aan de landcode uit het landenbestand:



Voorbeeld 60: Schematische weergave van de werking van een Join-Logical bestand

In een logical is men dus niet verplicht álle velden van de bronbestand(en) op te nemen, het mogen er ook slechts een paar zijn. Veel join logicals vertragen een bestandssysteem aanzienlijk omdat wanneer een record in een bronbestand wordt toegevoegd of verwijderd, het bestandssysteem alle gekoppelde (join) logicals ook moet bijwerken. Tegelijkertijd moeten de koppelingen met andere bestanden worden geactiveerd of gedeactiveerd. Bij 3GL's is het daarom vaak beter geen join logicals te gebruiken omdat wanneer het programma zelf de drie individuele records probeert te lezen het zelf adequate acties kan ondernemen mocht een record (nog) niet bestaan of in gebruik zijn door andere programma's. Een andere mogelijkheid is de index van de join pas op te laten bouwen op het moment dat er een programma is die de join opent. Daardoor hoeft het bestandssysteem deze join logical nooit permanent bij te werken. Echter heeft dat ook zijn nadelen want bij zeer grote bestanden vergt het opbouwen van een index ook weer aardig wat systeemcapaciteit wat andere processen weer vertraagd. Tevens kan het zijn dat het bestand zo vaak gebruikt wordt dat eigenlijk zinloos is steeds de indez opnieuw op te laten bouwen. Zeker als er meerdere gebruikers tegelijkertijd van het bestand gebruik willen maken. Logische bestanden vindt je veel bij database(-achtige) omgevingen en een goed voorbeeld van een logical waarbij de index pas wordt opgebouwd op het moment van de vraag is een pass-through query in MS-Access of andere permanente SQL queries die als 'gegevensbestand' fungeren voor programma's:

```
Select land.naam, continent.naam
Where land.continent = continent.code
Order by continent.naam, land.naam
```

Voorbeeld 61: Creatie van een join middels SQL

Deze SQL levert een recordset op met de naam van het land en de naam van het continent in één record, verzameld uit de bestanden LAND en CONTINENT en gesorteerd op de naam

van het continent en daar binnen verder gesorteerd op de landnaam. Bij de AS/400 kan je bijvoorbeeld een join-logical programmeren met DDS (*Data Description Specifications*) en deze permanent actief laten zijn, zodat de gegevens altijd beschikbaar zijn. Zo'n join-logical source ziet er als volgt uit:

```
R JLAND                 JFILE(CONT LAND)
J                       JOIN(1 2)
                        JFLD(COCODE LNCONT)
  CONAAM
  LNNAAM

K CONAAM
K LNNAAM
```

Voorbeeld 62: Join logical source voor een bestand onder OS/400

Je hebt enkel de beschikking over de velden CONAAM en LNNAAM, ongeacht het overige aantal velden in de bronbestanden CONT en LAND.

Databanken

Een echte relationele databank (*database*), welke een set bestanden en/of tabellen is waarbij alle relaties vastgelegd kunnen worden volgens de gemaakte Entity Relation Diagrams, is één grote join logical. Zodra bijvoorbeeld een klantenrecord gelezen wordt door een programma, zorgt het Relational Database Management System (*RDMBS*) voor het beschikbaar zijn van alle gerelateerde gegevens in andere bestanden of tabellen. Het programma behoeft daardoor geen extra programmering om gerelateerde gegevens in andere bestanden bij elkaar te zoeken.

Tabellen en bestanden kunnen verspreid liggen over meerdere computers en toch tesamen een database vormen (*distributed database*). Deze tabellen en bestanden kunnen onderdeel zijn van meerdere gedistribueerde of locale databanken. Het zijn de bestandsbeheerssystemen en database management systemen die ervoor moeten zorgen dat de gegevensstromen vlekkeloos verlopen.

Een database *bevat* tegenwoordig geen gegevens meer, het *omvat* alle benodogde gegevens, hoe en waar die ook mogen zijn opgeslagen. Dat betekent dat men een database kan ontwerpen die uit tabellen bestaat uit andere, reeds bestaande, databases tesamen met bestanden van diverse pluimage. Door interfaces als ODBC weet de programmatuur niet eens uit wat voor database de gegevens komen. Het is daardoor mogelijk een database te definiëren die bestaat uit MS-Access tabellen, Oracle en SQL-Server tabellen, RMS en OS/400 bestanden, FileMaker Pro bestanden, etc.. en hiervoor ook nog programma's te schrijven. In de praktijk gebeurt dit ook wel, echter op een zeer kleine schaal met slechts twee, maximaal drie, verschillende soorten tabellen of bestanden.

4. TALSTELSELS

Een andere zaak waarvan kennis genomen moet worden is hoe er gerekend wordt in een computer en wat voor notaties er voor getallen worden gehanteerd. Er worden vier talstelsels gebruikt waaronder ons decimale stelsel. Ons decimale stelsel heeft echter niets met computers en hun logica te maken, maar omdat dát het talstelsel is waarmee wíj nou eenmaal gewend zijn te rekenen, worden getallen uit alle andere stelsels eerst tot waarden herleid naar het decimale stelsel zodat de meesten onder ons ook nog begrijpen wat ze zelf verzinnen. Die herleidingstechnieken worden in dit hoofdstuk uitgelegd.

De belangrijkste eigenschap van de gebruikte talstelsels is dat het plaatswaarde stelsels zijn. Daarmee wordt bedoeld dat de plaats van een cijfer in een getal bepaalt welke waarde wordt voorgesteld. In de computertechniek worden het tweetallige (binair), het achttallige (octaal) en het zestientallige (hexadecimaal) stelsel gehanteerd. Het aantal cijfers in een talstelsel wordt ook wel de *radix* genoemd omdat dat aantal bepaald hoe er naar het decimale stelsel terug moet worden gerekend. Het tweetallige stelsel ligt aan de basis van de computertechniek omdat het overeenkomt met de standen van een aan-uit schakelaar. De computerchips, processor en geheugen, werken uitsluitend met aan-uit schakelaars (*flip-flops*). Het tweetallig stelsel is op moment van dit schrijven al 312 jaar oud. Het is door Gottfried Wilhelm Leibniz in 1690 opgesteld.

Het achtallig en zestientallig stelsel zijn machten van twee ($8=2^3$ en $16=2^{\square}$) en daardoor ideaal om met een verkorte schrijfwijze de lange getallen van het tweetallige stelsel te kunnen weergeven. Uiteraard kun je ook een ander talstelsel maken, wat ik in dit hoofdstuk zal doen. De naam van de gangbare talstelsels geeft bijvoorbeeld al aan van hoeveel symbolen er in een talstelsel gebruik wordt gemaakt om getallen samen te stellen:

- Binair (tweetallig); 2 cijfers : 0 en 1
- Octaal (achttallig); 8 cijfers : 0 t/m 7
- Decimaal (tientallig); 10 cijfers : 0 t/m 9
- Hexadecimaal (zestientallig); 16 cijfers : 0 t/m F (0-9, A-F)

Bij alle talstelsels, ook die je zelf maakt, heb je een nul nodig. Getallen worden namelijk samengesteld uit cijfers: 10 is samengesteld uit een één en een nul. Uit het voorbeeld blijkt dat de waarde van het laatste cijfer van een talstelsel altijd één minder is dan de naam van het talstelsel. Het TIENtallige stelsel heeft de symbolen 0 t/m (10-1) = 9. Het TWEEtallige stelsel heeft de symbolen 0 t/m (2-1) = 1. Hier komt de term *Binary Digit* vandaan (binair cijfer), wat wordt afgekort tot de term *bit*. Het HEXADECIMALE stelsel heeft de symbolen 0 t/m (16-1) = 15 = F. Bij alle talstelsels geldt ook nog de volgende regel: wanneer op een positie in een getal het hoogste cijfer is bereikt en je telt één bij het getal op, dan dient het cijfer links van die positie met één te worden verhoogd en begint de betreffende positie weer met nul, net als bij een kilometerteller. Eigenlijk is het decimale stelsel een schriftelijke vorm van de Abacus. Ook die werkt met 10 ballen per waardeplaats. De eerste rij ballen zijn de ééntallen, de tweede rij de tientallen, enzovoort. Waarom werken plaatswaarde stelsels zoals ze werken? Dat is om practische en efficiënte redenen. Het is namelijk onpractisch om voor elk getal een cijfer te hebben. Vroegere stelsels, zoals als het romeinse, hadden wel cijfers voor bepaalde getallen om minder te hoeven beitelen en je ziet hoeveel ruimte ze toch nog nodig hadden om

grote getallen weer te geven. Plaatswaarde stelsels zijn de meest efficiënte stelsels in het gebruik. Om te begrijpen wat de overeenkomsten zijn tussen de verschillende plaatswaarde stelsels worden deze`aan de hand van voorbeelden verklaard:

Onze 9 is eigenlijk 00000000000000000000000009

Wij schrijven de nullen aan de linkerkant van een getal niet, terwijl er dus eigenlijk een oneindig aantal nullen zou moeten staan. Maar omdat dat natuurlijk onleesbaar is, laten wij de *voorloopnullen* gewoon weg. Wanneer je er één bij optelt, wordt, zoals gezegd, het linker cijfer met één opgehoogd en begint de kolom waarbij 1 werd opgeteld weer bij 0:

Dus 9+1 is eigenlijk 00000000000000000000000009
\+ 00000000000000000000000001
= 00000000000000000000000010

En als ik er nou 2 bij optel, zul je je afvragen? Wel, 2 is eigenlijk 1 + 1; de som wordt dan 9 + 1 = 10 + 1 = 11. De hier genoemde optelregel geldt voor alle plaatswaarde stelsels. In onderstaand voorbeeld begint de optelling bij nul en wordt er telkens één bij opgeteld, tot 10 is bereikt. De getallen worden voor de duidelijkheid met één voorloopnul geschreven, de som bevat het volledige aantal posities ter grootte van het talstelsel:

Binair	Octaal	Decimaal	Hexadecimaal	
	00	00	00	00
	+ 01	+ 01	+ 01	+ 01
	01	01	01	01
	+ 01	+ 01	+ 01	+ 01
	10	02	02	02
		+ 01	+ 01	+ 01
		03	03	03
		+ 01	+ 01	+ 01
		04	04	04
		+ 01	+ 01	+ 01
		05	05	05
		+ 01	+ 01	+ 01
		06	06	06
		+ 01	+ 01	+ 01
		07	07	07
		+ 01	+ 01	+ 01
		00000010	08	08
			+ 01	+ 01
			09	09
			+ 01	+ 01
			0000000010	0A
				+ 01
				0B
				+ 01
				0C
				+ 01
				0D
				+ 01
				0E
				+ 01
				0F
				+ 01

0000000000000010

In bovenstaand voorbeeld eindigen de kolommen bewust met 10. Wij zeggen 'tien'. Dat komt omdat wij in ons tientallige stelsel namen van cijfers verbasteren en samenstellen tot een nieuwe naam. Ook hebben we namen verzonnen voor de waarden per symboolpositie, zoals honderd, duizend, etc.. Zulke namen hebben we niet verzonnen voor de andere stelsels, dus dat wordt moeilijk. Maar in elk stelsel zou je de notatie 10 'tien' kunnen noemen, 100 'honderd', enz.. 't Zijn maar namen.

Rekenen we de waarde 10 uit een ander stelsel om naar ons tientallige stelsel dan wordt het heel anders. Je zegt dan 'twee' tegen 10 in het *twee*tallige stelsel, 'acht' in het octale stelsel en 'zestien' in het hexadecimale stelsel omdat je in decimale termen de waarde wilt benoemen.

Posities

De eerste positie van een geheel getal is het meest *rechtse* cijfer. De reden hiervoor is dat een getal aan de linkerkant rekt of krimpt en aan de rechterkant niet in lengte veranderd. In ons decimale stelsel bijvoorbeeld, geldt de derde positie *van rechts* als de waarde 'honderd' of een veelvoud ervan (maximaal 9), de vierde positie *van rechts* geldt voor de duizenden (0000001000 of 0000002000, etc.). Dat heeft te maken met het aantal maal dat 10 met zichzelf vermenigvuldigd kan worden (*machtsverheffen*), gezien van rechts naar links, te beginnen met de nulde macht voor de meest rechtse positie (1^e) en steeds 1 macht hoger naarmate je naar links gaat. De macht NUL is een afspraak onder wiskundigen, anders rekent het zo rot. Alles tot de macht 0 is gelijk aan 1. Nu dit vastligt kan het getal 111 snel worden uitgewekt. In elk stelsel levert het een andere *decimale* waarde op, juist omdat in de diverse talstelsels de posities een andere waarde hebben:

binair:	$1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$	$= 4 + 2 + 1$	$= 7$
octaal:	$1 \times 8^2 + 1 \times 8^1 + 1 \times 8^0$	$= 64 + 8 + 1$	$= 73$
decimaal:	$1 \times 10^2 + 1 \times 10^1 + 1 \times 10^0$	$= 100 + 10 + 1$	$= 111$
hexadecimaal:	$1 \times 16^2 + 1 \times 16^1 + 1 \times 16^0$	$= 256 + 16 + 1$	$= 273$

Een positie van een getal heeft daarom altijd de waarde:

cijfer op positie * talstelsel $^{\text{positienummer-1}}$

De derde positie heeft dus macht 2, de vierde positie macht 3 (1000 in decimale stelsel is 10 tot de derde macht, wat overeenkomt met 10 x 10 x 10).

Talstelsels en computers

Computers zijn rekenmachines, niets meer en niets minder. Dat er beelden en geluid uit lijken te komen doet niets af aan de basis: rekenen. Het idee om hulpmiddelen te hebben om berekeningen te maken is al zou oud als de mensheid. De Abacus, Pascal's Pascaline en daarna is het pas goed begonnen met de Analytical Machine van de heer Babbage. En elke

eeuw, elke decade, elk jaar kunnen deze machines meer en worden ze kleiner en kleiner. Sommige van deze machinetjes werk(t)en met het tientallig stelsel en andere met het tweetalige stelsel. Die laatsten hebben het pleit gewonnen.

Aan en uit kunnen, zoals al eerder gezegd, heel goed worden weergegeven door een 1 en een 0. Een opdracht aan een computer bestaat uit meerdere aan-uit schakelaars, de bits, in een bepaalde stand. Sommigen aan en sommigen uit. Acht bits vormen een *byte*:

```
01111110
01100000
01111100
01100000
11111111
```

Zoals je ziet is het ondoenlijk en onleesbaar om dit zo met je collega's te delen en te bespreken. Niemand kan ál deze combinaties van enen en nullen onthouden. De andere genoemde talstelsels bieden een steeds kortere weergave van deze binaire waarden, met gebruik van cijfers en letters in bepaalde combinaties die we gewend zijn te zien.

Even eentussendoortje. Nu je weet wat een byte is, stellen we een tabelletje voor je op waarin duidelijk wordt wat Kb, Mb, Gb en Tb zijn:

1 Kb (Kilobyte)		=	1024 bytes	
1 Mb (Megabyte)	=	1024 Kb	=	1.048.576 bytes
1 Gb (Gigabyte)	=	1024 Mb	=	1.073.741.824 bytes
1 Tb (Terabyte)	=	1024 Gb	=	1.099.511.627.776 bytes

Ooit was er een octaal, het achttallige, talstelsel in gebruik in de computerwereld. Je vindt dit nog wel eens terug in ASCII en EBCDIC tabellen en oude VT100 terminal handleidingen. Het octale stelsel is geheel verdrongen door het hexadecimale stelsel. Je moet het echter niet vergeten omdat het zeer van pas kan komen bij het omzetten van een binaire representatie naar een kortere, iets makkelijker te onthouden cijferreeks. RAM geheugen en processor instructies bestaan uit bytes en dus bits. Vroeger was alles 8 bits 'breed'. De processor instructies, de breedte van de gegevensdoorvoer, de lettertekens voor het beeldscherm en de printer, de kleuren, etc.. Kijk eens aan de achterkant van de computer. Je ziet daar allerlei contacten waar stekers in kunnen. De meeste van die pinnetjes en/of gaatjes representeren een bit. Er gaat stroom, in millivolts, door die pinnetjes en gaatjes en bij een bepaalde positieve spanning (bijvoorbeeld +5 millivolt) zet de (co-)processor de bit op 1 en bij -5 millivolt op 0. En de software kan deze pinnen uitlezen en zo een byte samenstellen die op dat moment een bepaalde waarde of instructie voorstelt. Dat is natuurlijk uitermate handig opgelost. Je weet zó wat er bij welke pin gebeurt.

Nu ik dit schrijf is 32 bits de standaard breedte voor gegevensdoorvoer naar de processor(en) en is 64 bits tot 128 bits in opmars. Hoe meer bits er naast elkaar, dus tegelijkertijd, naar de processor gestuurd kunnen worden, des te sneller het hele apparaat wordt. Dezelfde ontwikkeling vindt plaats voor kleuren. Nog niet zo lang geleden konden we met 8 bits zijn maximaal 256 kleuren op een beeldscherm weergeven; met 16 bits worden het er al duizenden en met 24 bits al miljoenen en met 32 bits kleuren miljarden. Hoe meer bits er voor een kleur of tint gebruikt kunnen worden, hoe meer nuances in verzadiging en helderheid er met nullen en enen vastgelegd kunnen worden. En voor dit alles heb je veel geheugen

nodig.

Nu volgt het aantal groepen van drie en vier elementen, in dit geval de bits, die met het tweetalige stelsel te maken zijn. Ernaast zijn de cijfers van het betreffende talstelsel geschreven die overeenkomen met de binaire waarde:

3 bits	4 bits
000 = 0	0000 = 0
001 = 1	0001 = 1
010 = 2	0010 = 2
011 = 3	0011 = 3
100 = 4	0100 = 4
101 = 5	0101 = 5
110 = 6	0110 = 6
111 = 7	0111 = 7
	1000 = 8
2^3 combinaties = 8	1001 = 9
(*octale* stelsel)	1010 = A
	1011 = B
	1100 = C
	1101 = D
	1110 = E
	1111 = F
	2^4 combinaties = 16
	(*hexadecimale* stelsel)

Een reeks bits kan zó naar het octale of hexadecimale stelsel worden geconverteerd, zonder moeilijke omrekeningen. De truc is deze: neem de groep bits en splits de reeks, *van rechts naar links*, in:

- groepjes van 3 bits voor het octale stelsel.
- groepjes van 4 bits voor het hexadecimale stelsel.

Zet elk groepje bits om naar het bijbehorende teken uit het betreffende talstelsel door gebruik te maken van de beide hierboven getoonde tabellen. Maar je kan het natuurlijk ook netjes uitrekenen en dat gebeurt in onderstaand voorbeeld:

Bitreeks:		1	0	1	1	0	1	0	1
Octaal:									
	(stap 1)	2^1	+0	2^2	$+2^1$	+0	2^2	+0	$+2^0$
	(stap 2)		2		4	+2		4	
	+1 (stap 3)			2		6			
	5 *(stap 4: 265)*								
Hexadecimaal:		1	0	1	1	0	1	0	1
	(stap 1)	2^3	+0	$+2^1$	$+2^0$	0	$+2^2$	+0	$+2^0$
	(stap 2)	8	+0	+2	+1	0	+4	+0	+1
	(stap 3)				B				5
	(stap 4: B5)								
Decimaal:		1	0	1	1	0	1	0	1
	(stap 1)	2^7	+0	$+2^5$	$+2^4$	+0	$+2^2$	+0	$+2^0$
	(stap 2)	128	+0	+32	+16	+0	+4	+0	+1
	(stap 3: 181)								

Om er zeker van te zijn dat de berekeningen kloppen, rekenen we octaal 265 en hexadicemaal B5 om naar het decimale stelsel. Dat kan helaas niet eenvoudig door het getal op te splitsen. Hier moet echt gerekend worden:

Octaal:	$2*8^2$	$+ 6*8^1$	$+ 5*8^0$	(stap 1)
	2*64	+ 6*8	+ 5*1	(stap 2)
	128	+ 48	+ 5	(stap 3)
Decimaal:	*181*			*(stap 4)*

Hexadecimaal:	$B*16^1$	$+ 5*16^0$		(stap 1)
	11*16	+ 5*1		(stap 2)
	176	+ 5		(stap 3)
Decimaal:	*181*			*(stap 4)*

Gelukkig klopt het. Uiteraard kan je deze weg ook terug bewandelen. Als je een octaal of hexadecimaal getal om wilt zetten naar een reeks bits, zoek je uit bovenstaande tabel de overeenkomende bitreeks uit en plak je deze, van rechts naar links, aan elkaar. Wanneer je wel eens de broncode van een HTML-pagina hebt gezien, is je misschien opgevallen dat kleuren met letters en cijfers worden geprogrammeerd, bijvoorbeeld:

```
<BODY BGCOLOR="#AA5BC3">
```

Dit is een hexadecimale notatie voor de kleuren Rood (de eerste twee posities na het #), Groen (de volgende twee) en Blauw (de laatste twee). Dit illustreert dat het nodig is dat je van het ene naar het andere talstelsel om kan rekenen, omdat je het overal wel eens tegenkomt. Als je in een fotobewerkingsprogramma alleen decimale waarden voor de RGB kleuren kan ingeven, dan moet je om kunnen rekenen. Onder MS-Windows werkt de kleurenkiezer met waarden van nul tot en met 255. Onder MacOs echter werkt de kleurenkiezer procentueel van nul tot 100%. De kleur AA is 170 (decimale stelsel) en dat kan dus in de MS-Windows kleurenkiezer zonder meer worden ingegeven. Zoals gezegd werkt de kleurenkiezer onder MacOS procentueel en komt de decimale waarde 170 overeen met 66,67% van 255 (hex: FF). Overigens kan je in beide kleurenkiezers ook direct de hexadecimale waarde ingeven. Verder zijn er nog andere leuke dingen te doen met het getal AA5BC3:

AA = **A**$*16^1$ + **A**$*16^0$ = 10*16 + 10*1 = 170 voor ROOD.
5B = **5**$*16^1$ + **B**$*16^0$ = 5*16 + 11*1 = 91 voor GROEN.
C3 = **C**$*16^1$ + **3**$*16^0$ = 12*16 + 3*1 = 195 voor BLAUW.

Het gehele getal omgerekend naar het decimale stelsel:

AA5BC3 =	A * 16^5 =	10 * 1048576	=	10.485.760 +
	A * 16^4 =	10 * 65536	=	655.360 +
	5 * 16^3 =	5 * 4096	=	20.480 +
	B * 16^2 =	11 * 256		= 2.816 +
	C * 16^1 =	12 * 16	=	192 +
	3 * 16^0 =	3 * 1	=	3 +
				11.164.611

Omgezet naar het tweetallige stelsel:

A	**A**	**5**	**B**	**C**	**3**
1010	1010	0101	1011	1100	0011

```
is:     101010100101101111000011
```

En dat levert een octaal getal op van:

101	**010**	**100**	**101**	**101**	**111**	**000**	**011**
5	4	2	5	5	7	0	3

```
is:     54255703
```

Nu zie je hoe lang een getal kan worden als het binaire stelsel wordt gebruikt. Computers, en dus ook de programma's, werken met deze rijen van enen en nullen en is het verklaarbaar dat er steeds meer RAM geheugen en steeds grotere harde schijven nodig zijn. Op een schijf, elke schijf, geldt namelijk hetzelfde principe omdat een schijf bedoeld is als extern RAM-geheugen: het is er magnetisch (1) of niet (0). Bij CD's, die niet magnetisch zijn, is er een diep gat (1) of een niet zo diep gat (0) gebrand.

Over geheugen gesproken, hoe wordt de inhoud van een variabele eigenlijk in het geheugen bewaard? Natuurlijk door middel van enen en nullen. Neem een makkelijk type variabele: alfanumeriek. Elke positie van deze variabele wordt vertegenwoordigd door een byte. Er zijn ook DBCS (*Double Byte Character Set*) tekensets, zoals ook Unicode, om Aziatische en Arabische tekens weer te kunnen geven, maar we gaan voor het gemak uit van de ASCII en EBCDIC tekensets die slechts één byte per teken nodig hebben. Die byte bevat een combinatie van enen en nullen die de waarde representeren van een teken uit de ASCII of EBCDIC tekenset. Stel je declareert een variabele van dit type van maximaal 20 tekens groot:

```
SL:             tekst,  a20
C:              char tekst[21];
BASIC:          tekst AS STRING * 20
```

Je vult deze variabele vervolgens met de tekens "HALLO". Wanneer je dan met een programma dat het geheugen kan weergeven op de plaatsen kan kijken die variabele TEKST toegewezen heeft gekregen, zie je het volgende:

De numerieke representatie van de variabele TEKST met inhoud HALLO is in:

ASCII	decimaal	binair	octaal	hexadecimaal
H	72	01001000	110	48
A	65	01000001	101	41
L	76	01001100	114	4A
L	76	01001100	114	4A
O	79	01001111	117	4F

Als je bij de waarde van het eerste teken in bovenstaande variabele de decimale waarde 10 optelt, wordt de decimale waarde 82. Dit staat in de ASCII tabel voor de letter 'R'. Plaats de

string op het scherm en er staat 'RALLO'. Het bestaat uit niets en heeft geen enkele tastbare emotionele waarde. Dit hele boek is op de computer uitgewerkt en bestaat binnen die virtuele wereld uitsluitend uit gemagnetiseerde of gebrande delen van een schijf die, door de hardware, als één grote aaneengesloten reeks microscopisch kleine aan/uit schakelaartjes worden behandeld. Door allerlei internationale afspraken kan door software uiteindelijk een poging worden gedaan deze gegevens naar voor ons leesbare tekens om te vormen en op een beeldscherm of printer af te beelden. Voor een computerkunstenaar het ultieme digitale feest omdat ingetikte tekst als bitreeks een kakafonie aan geluid kan produceren als je die bits naar de luidsprekers stuurt of een prachtige ruis kunnen veroorzaken op het beeldscherm. De huidige muziekafspeelprogramma's, zoals UltraPlayer, hebben bijvoorbeeld allemaal een *visualizer* ingebouwd die de volume- en toonwisselingen gebruiken om hypnotiserende beelden te genereren.

Waardebereik

De typen variabelen waarmee je kan rekenen zijn integer, long integer en float. En alleen bij het type float kan met een drijvende komma (*floating point*) gerekend worden, bij de rest moet je de kommapositie bij de definitie specificeren. En als dat niet kan bij de programmeertaal die je gebruikt, dan vallen de decimalen er helaas af of moet je de decimalen in het veld opnemen en daar met de berekeningen rekening mee houden. Bij het type float moet echter niet gerekend worden op een superieure accuratesse, want de gemiddelde processor voor zakelijk en thuisgebruik is daar niet op gebouwd. Alleen de hele dure supercomputers kunnen goed omgaan met de drijvende komma. Daarom werken de meeste HLL's niet met drijvende komma's maar met integers met een vast te definiëren kommapositie. En dat niet alleen vanwege de acuratesse, integers zijn door de processoren ook nog eens sneller te verwerken dan een drijvende komma getal. Een bedragveld dat tien miljoen (8 cijfers voor de komma) moet kunnen bevatten maar ook vijf cent (2 posities na de komma) wordt dan als volgt gedefinieerd: 8 + 2 = 10.2 waarbij 10 de gehele lengte van het veld aangeeft en .2 het geintegreerde aantal decimalen. Er kan ook gekozen worden voor gewoon 10, want ook dan bevat het veld het bedrag in centen. Er moet dan echter rekening gehouden worden in de programmatuur dat het getal een factor 100 te groot is, wat normaliter door de rekenregels van de taal wordt afgehandeld:

```
Resultaat = 0           ' 10.2 veld
Bedrag1 = 100           ' 10.2 veld : 10000     (  100,00)
Bedrag2 = 10000         ' 10 veld   : 10000     (10000   )

Resultaat = Bedrag1     ' Inhoud wordt 100      (  100,00)
Resultaat = Bedrag2     ' Inhoud wordt 10000    (10000,00)
```

De HLL’s stellen variabelen, ook numerieke, samen uit bytes. Daardoor is een getal niet gebonden aan de grootte van een integer of long. Een long beslaat normaliter 16 tot 32 bytes, meer niet. In een HLL echter, kan een decimaal veld worden gedefiniëerd van 72 bytes en daar ook mee rekenen. Daar zorgt de compiler voor.

Dan heb je bij het type integer en type long integer nog een extra type aanduiding, namelijk *unsigned*. Even uitleggen... in deze typen variabelen kan je ook negatieve waarden opslaan. Maar zoals we de afgelopen hoofdstukken wel hebben gemerkt, is het onmogelijk

een negatieve waarde uit te drukken met aan/uit schakelaars of enen en nullen. Dat het toch mogelijk lijkt, komt doordat er internationaal is afgesproken dat er bij *signed* variabelen, en ook bij *zoned decimals*, de meest linkse bit voor het + of - teken wordt gebruikt. Daardoor heb je wel één bit minder ter beschikking. Geef je het type echter het voorvoegsel 'unsigned' mee, dan geef je aan dat er geen teken (+ of -) mag worden gebruikt bij deze variabele. En daardoor wordt het bereik van de positieve waarden vergroot tot het maximaal haalbare binnen het gekozen type variabele en zijn negatieve waarden niet mogelijk.

Laten we eens wat gaan spelen met een integer. We stoppen er ruwweg de omtrek van de aarde in: 40000. Dat kan echter alleen in een integer met een breedte van minimaal 16 bits! Nu willen we de binaire representatie weten van dit getal. Het makkelijkst is het om deze decimale waarde eerst om te rekenen naar octaal of hexadecimaal, want elk cijfertje van deze talstelsels staat voor een paar bits die enkel aan elkaar geplakt hoeven te worden. Omrekenen van decimaal naar een ander talstelsel gaat als volgt: steeds blijven delen door de stelselgrootte totdat het quotient kleiner is dan de stelselgrootte. Alle resten, van rechts naar links aan elkaar geplakt, vormen tesamen het getal in het nieuwe stelsel. Willen we dus omrekenen naar het achtallige stelsel, dan moeten we volgens bovenstaande regel blijven delen door 8, totdat het quotient kleiner is dan 8:

```
40000 / 8 = 5000, rest = 0
```

Deze rest is het eerste, meest rechtse, cijfer: 0. Dan moet je nu verder gaan met het quotient (5000):

```
5000 / 8 = 625, rest = 0
```

Deze 0 is het tweede cijfer van rechts: 00. Nu 625 delen door 8:

```
625 / 8 = 78, rest = 1
```

Deze 1 is het derde cijfer: 100. Nu 78:

```
78 / 8 = 9, rest = 6
```

Nu hebben we het vierde cijfer: 6100. Nu 9 nog:

```
9 / 8 = 1, rest 1
```

En dat zijn de laatste twee cijfers. De rest eerst, en dan het quotient. Er staat dan, octaal, 116100. Dezelfde regel gaat op voor het hexadecimale stelsel, dus blijven delen door 16:

```
40000 / 16 = 2500, rest = 0
```

Deze rest is het eerste, meest rechtse, cijfer: 0. Dan moet je nu verder gaan met de 2500:

```
2500 / 16 = 156, rest = 4
```

Deze 4 is het tweede cijfer: 40. Nu 156 delen:

```
156 / 16 = 9, rest = C
```

En dat zijn de laatste twee cijfers. De rest eerst, en dan het quotient. Er staat dan, hexadecimaal, 9C40. Ter controle rekenen we de beide resultaten terug via de machten van de positie van elk cijfer:

octaal		hexadecimaal	
$1 * 8^5 =$	32768	$9 * 16^3 =$	36864
$1 * 8^4 =$	4096	$C * 16^2 =$	3072
$6 * 8^3 =$	3072	$4 * 16^1 =$	64
$1 * 8^2 =$	64	$0 * 16^0 =$	0
$0 * 8^1 =$	0		40000
$0 * 8^0 =$	0		
	40000		

Maar we wilden in eerste instantie de binaire representatie zien van 40000. We nemen daarom het octale getal, da's makkelijker omdat elk cijfertje slechts 3 bits representeert, en dus een maximale waarde van 7. Het octale getal 116100 moet je als volgt uitsplitsen:

```
  1      1      6      1      0      0
001    001    110    001    000    000
```

Er ontstaat een binair getal van 18 bits (6 x 3). Omdat de bits 18 en 17 de waarde 0 bevatten, vallen ze af en houden we 16 bits over. Dat past dus precies in een unsigned integer van 16 bits. Zou je per ongeluk het decimale getal 40000 in een signed integer stoppen van 16 bits, dan krijg je geen 40000 terug omdat de eerste bit niet meer als waarde (32768) wordt meegteld maar aangeeft dat er zich in de volgende 15 bits een negatief getal bevindt met de waarde 40000 - 32768 = -1 * 7232.

Kortom, wanneer er vreemde dingen gebeuren in je programma en je raakt gigantische bedragen kwijt of alles wordt opeens negatief, kijk dan eens naar het gebruik van signed en unsigned variabelen. Een voordeel van de hogere programmeertalen zoals COBOL, RPG, SL, BASIC, etc. is dat standaard alles signed is. Een nadeel is dan dat je voor grote positieve getallen meer geheugen nodig hebt omdat de variabele groter moet zijn vanwege die sign-bit

die mee gaat. Vroeger gaf dat nog toen je 1000,= voor één megabyte moest neertellen. Maar tegenwoordig....

Een laatste grapje met deze integer. We schuiven deze 16 bits, dus 2 bytes, over de eerste twee bytes van de variabele TEKST uit een eerder voorbeeld. Er stond ondertussen al 'RALLO' in variabele TEKST, maar na deze operatie:

oude binaire waarde:

```
01010010    01000001    01001100    01001100    01001111
    R           A           L           L           O
```

nieuwe binaire waarde (zonder voorloopnullen):

```
10011100    01000000    01001100    01001100    01001111
    ú           @           L           L           O           (Macintosh)
    œ           @           L           L           O           (Windows)
```

32-tallige Stelsel

Ach, het stelt allemaal niet zoveel voor. Nu de 32-, 64- en 128-bits componenten in opmars zijn, is het weer eens tijd voor een talstelsel dat die lange reeksen bits nog wat extra verkort weer kan geven, want zelfs de hexadecimale getallen worden nu lang. Daarom creëren we nu een herhalingsvariatie met 0 en 1 van de 5^e klasse: een 32-cijferig talstelsel (2^5). Cijfers van dit talstelsel worden 0 t/m 9 en A t/m V. V heeft de volgende decimale waarde:

```
stelselgrootte - 1 = 32 - 1 = 31
```

Hoeveel is V + 1?

```
10              (1 * 32^1 + 0 * 32^0 = 32 decimaal)
```

Dit talstelsel moet ook een naam krijgen. Historisch is Grieks de taal van de wiskunde en dus noemen we het stelsel het ditriacontale stelsel. Als we nou die vijf bytes van het woord 'HALLO' weer eens nemen en we splitsen domweg de binaire waarden op naar talstelsel, dan zie je dat voor het 32-talstelsel je vijf bits per cijfer nodig hebt:

```
binair:             2^1 bits =  2 cijfers (0 t/m 1; 1 = binair 1).
octaal:             2^3 bits =  8 cijfers (0 t/m 7; 7 = binair 111).
hexadecimaal:       2^4 bits = 16 cijfers (0 t/m F; F = binair 1111).
ditriacontaal:      2^5 bits = 32 cijfers (0 t/m V; V = binair 11111).
```

Hier volgt de uitsplitsing van 'HALLO' in binaire vorm naar de diverse talstelsels:

Oude binaire waarde:

```
01001000    01000001    01001100    01001100    01001111
    H           A           L           L           O
```

Octaal (klasse 3, dus 3 bits per cijfer):

```
0 100 100 001 000 001 010 011 000 100 110 001 001 111
0  4   4   1   0   1   2   3   0   4   6   1   1   7
```

Hexadecimaal (klasse 4, dus 4 bits per cijfer):

```
0100 1000 0100 0001 0100 1100 0100 1100 0100 1111
 4    8    4    1    4    C    4    C    4    F
```

Ditriacontaal (klasse 5, dus 5 bits per cijfer):

```
01001 00001 00000 10100 11000 10011 00010 01111
  9     1     0     K     O     J     2     F
```

Zoals voor alle andere talstelsels geldt ook hier dat de positie bepalend is voor de waarde van een cijfer. Als dit ditriacontale getal wordt omgerekend naar het decimale stelsel, gaat dat op precies dezelfde wijze als bij de andere positionele stelsels:

```
9 * 32^7 =     9 * 34359738368  =   309237645312 +
1 * 32^6 =     1 * 1073741824   =     1073741824 +
0 * 32^5 =     0                =              0 +
K * 32^4 =     20 * 1048576     =       20971520 +
O * 32^3 =     24 * 32768       =         786432 +
J * 32^2 =     19 * 1024        =          19456 +
2 * 32^1 =     2 * 32           =             64 +
F * 32^0 =     15 * 1           =             15 +
                                  ---------------
                                  310333164623
```

In Appendix A vindt je een programma, in HTML en een beetje JavaScript, om een positieve decimale waarde naar de vier, dus inclusief het nieuwe, talstelsels om te rekenen. Neem een eenvoudige tekstverwerker, Simpeltekst, Simpletext, Kladblok of Notepad, toets of scan het programma in en bewaar het met de extensie .html. Sleep het op je internet-browser icoon (een versie die JavaScript ondersteund en aan heeft staan), geef een decimaal getal in en klik op 'Reken om'.

5. CREATIE, BEHEER EN ONDERHOUD

Het moment is aangebroken om het eens te gaan hebben over de aanpak van een automatiseringstraject. Of het nu over één (extra) subroutine gaat of over een geheel nieuw te bouwen software pakket, de aanpak verschilt namelijk niet zoveel, slechts de omvang en de details. Het belangrijkste van alles om een traject tot een goed einde te brengen is dat je als softwaremaker de probleemstelling *begrijpt* en kan *overzien*. Een klant van mij gebruikt de term 'helicopter-view'.

Luisteren, aanvoelen en flexibiliteit

Je moet heel goed kunnen luisteren, je kunnen inleven in de situatie van de opdrachtgever en gelijk kunnen 'zien', of aanvoelen, wat de bedoeling is. Wanneer je dit kan, kan je als automatiseerder tijdens een gesprek of vergadering al gauw wat verder vooruit kijken en daardoor aan zien komen waar in het softwarepakket en waar in de organisatie mogelijke problemen op kunnen duiken. Vervolgens kun je de betrokkenen hiermee confronteren en daaraan zien ze dat je met hen meedenkt, wat vaak een geruststelling is voor de opdrachtgever. Hij krijgt zo niet het gevoel dat hij iets probeert uit te leggen wat door de uitvoerende kant niet wordt begrepen.

Mee- en vooruit denken is ook belangrijk omdat problemen en bepaalde details nooit voor de volle 100% aan je worden voorgelegd of aan je worden verteld. Een heel groot deel van het automatiseringstraject kan door niet-automatiseerders niet worden overzien en je kan de broodnodige informatie er alleen uit krijgen door de juiste vragen te stellen. En om de juiste vragen te kunnen stellen, moet je inzicht hebben of krijgen in de problematiek waarmee de opdrachtgever worstelt. Door de interactie tussen jou en de opdrachtgever kom je vanzelf tot een voor dat moment ideale oplossing. Ik zeg 'voor dat moment', want ook jij zult niet van te voren kunnen bedenken wat er binnen een probleemstelling voor mogelijkheden zijn om een probleem aan te pakken. Tijdens het verder uitwerken van het automatiseringstraject merk je dat er steeds nieuwe uitdagingen opduiken die om een oplossing vragen. Daarom is het heel gewoon dat een aan het begin bedachte oplossing er aan het eind héél anders uit is gaan zien. Wanneer dat zonder overleg met de betrokkenen plaatsvindt, heb je ten alle tijde een groot probleem, omdat je de mensen niet geeft wat ze verwachten terwijl ze zich al wel op de komst van de software hebben ingesteld; sterker nog, zeer waarschijnlijk zijn er al allerlei organisatorische zaken in beweging gekomen ter voorbereiding van de komst van jouw software. Om deze beschamende situatie te voorkomen zal je alles waar je ook maar een klein vraagje over hebt uit moeten werken met de gebruikers, zodat die meegroeien met de ontwikkelingen en niet voor verrassingen komen te staan wanneer jij plotseling op komt draven met je programma’s.

En dan is er nog het aanpassingsvermogen dat je nodig hebt. Wanneer je in een vreemde omgeving terecht komt waar je met andere programmeurs gaat samenwerken, kan je natuurlijk niet zomaar op je eigen vertrouwde wijze verder programmeren, tenzij ze het daar ook zo doen als jij. Je moet je eerst richten op de daar geldende regels en de manier waarop de software in die omgeving wordt gemaakt. Ga vooral niet gelijk programmeren 'om je te bewijzen', dat werkt gewoonweg niet; eerst lezen en leren, dan pas doen. Verdiep je in de aanwezige documentatie, het liefst de meest recente. Je kan altijd nog oude documentatie

doorwrochten wanneer je zover bent dat je de achtergronden van de bestaande software eens nader zou willen bekijken. Om snel in te burgeren bekijk je de sources, start je de programma's die uit die sources zijn ontstaan en laat je de manier van programmeren die wordt gebruikt op je inwerken zodat het al gauw lijkt alsof je nooit anders hebt gedaan. Wanneer je zonder documentatie aan een software pakket wordt gezet, is het van nog meer belang van te voren eerst alles goed te aanschouwen. Zo zal je vanzelf herhalingen vinden in de sources die duiden op bepaalde gehanteerde 'standaards'. Veldnamen, namen van functies en subroutines, etc.. Vergewis jezelf ervan of die herhalingen inderdaad de gebruikelijke manier van werken illustreren door bij je collega's navraag te doen. Het is beter om veel te vragen dan dat men je steeds moet corrigeren. Je leert er altijd weer wat van en, als je zelf al over ruime ervaring beschikt, kan je anderen zelfs wat van jou kennen en kunnen laten meegenieten.

Een andere, heel belangrijke, eigenschap is flexibiliteit. Stel je werkt bij een organisatie die vandaag rechtdoor walst en morgen linksaf zou kunnen slaan. Dat kan echter ook rechtsaf zijn. Daardoor worden er constant nieuwe eisen aan de software gesteld, zoals een net afgerond ontwerp dat nu compleet moet worden herzien omdat het bedrijf toevallig net gisteren een andere koers heeft genomen. Uiteraard wisten sommigen binnen het bedrijf wel dat het ging gebeuren, ze hebben alleen verzuimd het jou op tijd te vertellen. Als je te boek wilt staan als een automatiseerder 'die je moet hebben', dan moet je dit soort situaties heel goed op kunnen vangen. Je kan niet eigenwijs tóch rechtdoor willen gaan. Bedrijven moeten nou eenmaal actief op de marktwerking reageren en zullen altijd proberen hun concurrentiepositie te verstevigen. Overnames zijn geen onbekende factoren en zorgen voor een zware druk op de automatiseerders die zich plotseling in een geheel andere en grotere organisatie geplaatst zien: nieuwe collega's, andere bedrijfs- en overlegculturen en eventueel andere hard- en software. Software krijgt bij deze processen een steeds zwaardere ondersteunende werking. Blijf daarom met beide benen op de grond staan en zorg dat je een goed contact houdt met het management.

Contacten en kruiwagens

In je eentje kom je niet ver. Nu zullen een heleboel zeggen dat dat niet zo is, maar zonder zeepfabrikant stonk je nu nog een uur in de wind. En zonder werkgever of klanten krijg je ook geen, of onvoldoende, geld. Dus inderdaad, in je eentje kom je niet ver. Een goede zaak om na te streven is om binnen de organisatie waarvoor je programma's schrijft, één of twee mensen te vinden op management niveau, of hoger, waarmee je een goed persoonlijk contact kunt opbouwen en, nog belangrijker, die volledig van de bedoelde functionaliteit en toepassing binnen de organisatie van het softwarepakket op de hoogte zijn en blijven.

De hoofdreden om zulke contactpersonen te vinden en daarmee een goede relatie op te bouwen is dat die personen vanuit hun positie een overzicht over de organisatie hebben en kunnen behouden. Zij kunnen daardoor beter vertellen waarom bepaalde nieuwbouw of beppaalde aanpassingen noodzakelijk zijn, dan iemand die zich op een lager niveau beweegt. Dat komt onder andere door het simpele feit dat het lagere personeel nou eenmaal niet van alles wat zich er op een hoger niveau afspeelt op de hoogte is.

Je contactpersonen moeten je voor de volle 100% willen ondersteunen bij kennisgebrek aan jouw kant en ook concreet probleemoplossend kunnen zijn. Ze moeten daarom affiniteit

met automatisering hebben anders zullen ze je nooit echt begrijpen en zal jij hun bedoelingen niet verstaan. Een andere eis waaraan deze contactpersonen moeten voldoen is dat ze een ruime kennis hebben van de werking van de software op het functionele vlak om te kunnen beoordelen of gewenste functionaliteit al in het pakket aanwezig is of niet, zonder steeds jou te moeten vragen. De combinatie van technische en functionele personen die elkaar begrijpen is ijzersterk en levert daarom een ijzersterk softwarepakket op.

Een moeilijker te beheersen thema is een softwarepakket dat door een internationale organisatie wordt gebruikt en ook internationaal, meertalig, wordt ingezet. Elke cultuur heeft zo zijn wetten, werkwijzen en taalgebruik. De beste aanpak is dat er op hoog niveau een club wordt opgericht waarin grensoverschrijdend management zitting heeft dat voldoet aan de eerder gestelde eisen èn waarin ook jouw persoon zitting heeft. Alle benodigde aanpassingen en nieuwbouw dienen dan door de club goedgekeurd te worden voordat ze worden geprogrammeerd.

Het komt regelmatig voor dat binnen een organisatie bepaalde mensen hun eigen softwareaanpassingen door proberen te drukken. Pas daarvoor op en overleg deze wensen altijd met je contacten in het management. Als ze weten dat je hun wensen altijd toch eerst aan je contacten voorlegt, zullen ze proberen, uit jaloezie, de band die jullie hebben te verbreken. Wanneer je zulke dingen merkt, schroom dan vooral niet het je contactpersonen te vertellen, want alleen zij bevinden zich in de positie deze collegae weer op het rechte pad te brengen. Het probleem met dit soort 'collegae' is dat ze lager in hiërarchie zitten, toch een soort leidende functie hebben en niet verder kunnen kijken dan hun eigen afdeling en werkzaamheden. Zou je je door deze mensen laten verleiden en domweg voor ze gaan programmeren, dan ga je de mist in. Reken daar maar op. De één wil dit, de ander dat en van weer een ander moet dat eigenlijk dit zijn en dat moet anders dan dit was en ook een beetje meer naar rechts, maar alleen als je tijd hebt, hoor... En zeker hebben ze ook wel eens gelijk, maar het beste is ze er op te wijzen dat ze hun wensen niet meer bij jou neerleggen maar direct met jouw management contacten bespreken. Die hebben de 'helicopter-view' en kunnen bepalen of het de investering van tijd en geld waard is en jij bent de last kwijt.

Tijd

Als je eenmaal lekker bezig bent, hèb je helaas geen tijd meer over. Wanneer je wilt overleggen over programma-aanpassingen of wanneer er vanuit de organisatie nieuwe wensen kenbaar worden gemaakt, heeft het alleen zin die uit te diepen als er vóóraf een korte probleemstelling aan de betrokkenen wordt voorgelegd ter kennisname en er vervolgens door alle betrokkenen op een bepaalde dag tijd voor wordt vrijgemaakt, maar dan ook ruim tijd en niet 'effe tussendoor', om de nieuwigheden te bespreken.

Wanneer zo'n uitgebreid overleg ten einde is moet, voor iedereen het probleem duidelijk zijn en moeten de ingewikkelde zaken die ter sprake gekomen zijn of die voortvloeien uit de nieuwe wegen die worden ingeslagen, beschreven zijn op een wijze die twee weken later nog steeds even begrijpelijk voor je is. Tevens moet het zeker zijn dat het probleem (grotendeels) opgelost kán worden en moet het ook voor iedereen duidelijk zijn hóe het opgelost gaat worden. Dat jij en/of je team het softwarematige deel van het vraagstuk oplost, daar mag nooit enige twijfel over ontstaan. Dat je ook mee adviseert en meedenkt met de aanschaf van hardware staat buiten kijf. Je behoort te weten waarop en waarmee je werkt of moet gaan

werken, maar je hoeft geen monteur te zijn.

Welke programma's er moeten komen en wat voor bestanden of databases er moeten worden gebouwd is deels aan jou en komen vanzelf naar voren uit een gedetailleerde uitwerking van de oplossing voor het probleem. Wat ook na de vergadering bekend moet zijn is op welke platforms de oplossing wordt gerealiseerd (AS/400, Unix, Windows, MacOS, etc.) en met welke ontwikkelomgevingen. Er kan dan namelijk al rekening worden gehouden met de aanschaf van hardware en eventuele software licenties. Ook zijn er bij sommige vraagstukken meerdere afdelingen binnen een bedrijf betrokken die er nieuwe werkzaamheden bij krijgen of waar er handelingen wegvallen. Ook die veranderingen moeten worden gedocumenteerd, om later te kunnen kijken of die doelen werkelijk zijn bereikt. Dat heeft misschien niet direct meer iets met jouw vakgebied van doen, maar het hoort er wel bij dat je van zulke bedrijfsmatige veranderingen op de hoogte bent. Zo realiseer je je tenminste weer eens wat voor impact jouw software pakket op een bedrijf, de klanten en de werknemers kan hebben.

Waar je ook vooral tijd voor vrij moet maken is je vakliteratuur en het volgen van nieuwe ontwikkelingen die je misschien nog niet eens raken. In grote organisaties ben je echt niet de enige die op de hoogte is van het nieuwste en zullen er eerder vragen rijzen waarom 'we nog niet zus of zo doen'. Dus blijf bij de tijd, want voor je het weet vragen ze het jou en sta dan maar eens met een mond vol tanden...

Een ander belangrijk punt is het reserveren van of zoeken naar programmeurs. Als bekend is hoe een vraagstuk opgelost gaat worden, is ook bekend welk soort programma's er gemaakt moeten worden. En uiteraard zijn er programma's bij die veel tijd kosten om te maken maar niet ingewikkeld zijn. Daar kan je misschien een goedkopere programmeur op zetten zodat je je kan concentreren op de ingewikkelde zaken die voor een buitenstaander níet snel op te pikken zijn.

Wanneer je bekend bent met de gewenste aanpassing of nieuwbouw, is het zaak er grip op te krijgen en te behouden. Niet alleen voor jezelf, maar ook voor je collega's en je eventuele opvolgers. Waarschijnlijk denk je eerder 'ach, ik onthoud het wel'. Vast wel. Maar niet voor lang, omdat je eenvoudigweg wel meer aan je hoofd hebt en je eigenlijk nooit gelijk kan beginnen met programmeren. Daardoor ga je de kleine dingetjes vergeten en dat worden er langzamerhand zoveel dat je niet meer precies weet hoe het ook al weer zat en ben je wéér extra tijd kwijt om het allemaal op te rakelen. Wat moet je doen?

Registreren

Neem vooral de tijd om je gedachten en notities te ordenen. Zoek bijvoorbeeld een rustige werkplek op waar de kans op verstoring erg klein is. Na de eerste inventarisatieronde blijken er vaak toch nog onduidelijkheden en niet gestelde vragen op te duiken. Noteer deze ook en probeer er een antwoord op te krijgen. Hoe minder vragen er overblijven, des te duidelijker de ingeslagen weg voor iedereen wordt. Geef vraagstuk vervolgens een nummer en een thema en noteer hoe het bij je terecht is gekomen, van wie het afkomstig is, wanneer het gemeld is, welke prioriteit het heeft gekregen, etc.. Beschrijf het vraagstuk, het beoogde doel en globaal de taken die na elkaar uitgevoerd moeten worden. Als dat is gelukt probeer je het traject in stukken de hakken die onafhankelijk van elkaar gebouwd kunnen worden. Geef deze

subprojecten ook een nummer en verwijs naar het hoofdnummer. Pas dan kun je rustig verder gaan met een implementatieplan en een diepere analyse van elk subproject over de aanpassingen aan bestanden en programma's die moeten plaatsvinden, welke impact dit heeft op het systeem, op externe systemen en de bedrijfsvoering. Dit hoeven echt geen ellenlange teksten te zijn die er indrukwekkend uit moeten zien, tenzij dat de bedrijfscultuur is waarin je je beweegt; het mogen ook kladjes zijn die na verwerking in het ronde archief verdwijnen. Het is echter in je eigen belang om bij opdrachten die meerdere mensen betreffen geen kladjes maar een netjes uitgewerkt document te publiceren. Print deze documenten naar een PDF (*Adobe's Portable Document Format)* of HTML (*Hyper Text Markup Language*) bestand en stel deze beschikbaar op het intranet van het bedrijf waarvoor je programmeert. Waar het om gaat is dat de doelen, de werking en de wijzigingen beschreven zijn, aan de hand van het document zijn uitgevoerd en voor nieuwe medewerkers als referentie beschikbaar zijn. Een uitgewerkt plan kan je stap na stap afvinken, waardoor je zelf het inzicht, overzicht en een status behoudt over de voortgang van het subproject. Op deze manier werk je eigenlijk een project uit naar een boomstructuur en deze structuur werk je van onder naar boven af.

Wanneer alles op bovenstaande wijze in een database wordt bijgehouden, is het mogelijk deze gegevens in de vorm van een heldere rapportage uit te delen aan, en via internet raadpleegbaar te maken voor, de betrokkenen.

Wanneer dan de zaken zijn aangepast of veranderd, willen de betrokkenen, die natuurlijk altijd weer gespannen afwachten waarmee ze vandaag worden geconfronteerd, graag geïnformeerd worden of de mogelijkheid hebben zichzelf vooraf te informeren. Denk daarbij aan een pagina op het intranet waarop per datum de belangrijkste wijzigingen worden vermeld. Daarmee is die pagina gelijk een soort tijdbalk waarin te vinden is wanneer bepaalde wijzigingen hebben plaatsgevonden.

Voorbereidingen

Tachtig procent van de programma's die je zal maken zien er hetzelfde uit, van binnen en van buiten. Er is natuurlijk wel wat variatie, maar niet veel. Zo zijn er eenvoudige programma's als 'Onderhoud Landen', 'Onderhoud Valuta', 'Onderhoud Branches', etc. Deze hebben allemaal dezelfde eenvoudige structuur. De iets uitgebreidere versies zoals 'Onderhoud Klanten', 'Onderhoud Artikelen', 'Onderhoud Boeken en Strips' hebben ook alle ongeveer dezelfde structuur. De overige 20% van de programma's zijn echt specifiek voor een taak gemaakt. Dat zijn de verwerkingsprogramma’s die gebruik maken van de gegevens die met de 80% worden onderhouden. Voor die 80% van de programma's is het van belang er niet te lang over te doen als je eens een nieuwe moet maken. Wat vaak gebeurt is dat er een copietje van een bestaand programma wordt gemaakt, wat dan wordt aangepast. Bij heel simpele programmaatjes lukt dat ook wel zonder ernstige problemen. Bij de iets uitgebreidere programma’s ben je toch al gauw wat langer bezig met de oude code te vervangen door nieuwe. Een ander nadeel van copiëren is dat je een nieuw programma baseert op oude code. Om dat nou te voorkomen, moet je een skelet maken voor je programma's welke je als *template* voor toekomstige programma's hanteert. Alles wat je tijdens het programmeren er bij leert aan handigheidjes neem je in dit skelet op. Dan zal een nieuw programma altijd gebaseerd zijn op je meest recente kennisniveau en zou je, wanneer je toevallig eens in de oude programma's bezig bent, een programma met oudere code kunnen opwaarderen naar de nieuwste stand.

Door gebruik te maken van deze templates heb je snel de beoogde functionaliteit en gebruikersinterface ingebouwd zonder steeds weer over taalelementen en structuren je hersens te moeten pijnigen. De gebruikersinterface blijft bij elk nieuw programma zeer herkenbaar, wat zorgt voor een vermindering van stress bij de gebruiker. Daarvan heeft deze namelijk al genoeg last door alle veranderingen in de software die wekelijks op hem af komen. Een ander, zeer belangrijk, aspect is dat de foutkans zeer gering is en dat fouten sneller zijn opgespoord omdat het niet aan de standaard source kan liggen. Doet het dat in dat ene uitzonderlijke geval nu net wèl, dan heb je tenminste in alle andere programma's die op deze standaard zijn gebaseerd ook de fout gevonden èn, als je geluk hebt, opgelost nog voordat iemand anders hem ontdekt.

Men past deze werkwijze natuurlijk al wereldwijd toe door gebruik te maken van herbruikbare objecten, klassen, controls, shared libraries, dynamic link libraries (DLL), etc.. Op het locale niveau waar jij je beweegt, waar jij gebruik maakt van al die objecten van anderen, is het echter net zo belangrijk dat jij je ook zo opstelt voor je eigengebouwde software. Het scheelt echt verschrikkelijk veel tijd, want de kern die de oplossing is voor een probleem is vaak redelijk snel geprogrammeerd. Maar de gebruikersinterface..... daar ben je wel even zoet mee en die kost de meeste tijd en hoofdbrekens.

Wanneer je eens weer wat nieuws hebt ontdekt dat een programma qua gebruikersvriendelijkheid verbeterd, voeg je dat natuurlijk aan je skelet toe. Als je niet de enige bent die gebruik maakt van deze sources, moet je van te voren overleg plegen met je collega's of die dit niet ook al hadden ontdekt en het misschien op een betere manier hebben opgelost. Sta vooral open voor de oplossingen van anderen. Je zal wel merken dat veel programmeurs niet zo open staan voor kritiek en snel jaloers zijn op iemand die een betere oplossing heeft, wat natuurlijk helemaal FOUT is.

Waarmee je ook je probleem oplost, met elk ontwikkelgereedschap kan je een basisstructuur opbouwen. Voor elk probleem dat middels een computer kan worden opgelost is wel een programma te maken en niet voor elk soort probleem heeft het zin een skelet te maken. Programmeer je hardware, dan is het practisch onmogelijk om standaard sources en objecten te hanteren. Ik ga echter uit van de beginnende programmeur en die beweegt zich als eerste vaak op het administratieve vlak.

Een tip: probeer zo min mogelijk afdrukprogramma's te maken. Slimme managers kunnen de gegevens zelf wel via ODBC of een andere interface in een spreadsheet of grafiekenprogramma binnenhalen en de mooiste lijsten produceren.

Verandering aan bestaande programmatuur

Sleutelen aan bestaande, werkende en in gebruik zijnde programma's en bestanden brengt grote risico's met zich mee. Een voorbeeld is het verdwijnen van ongedocumenteerde functionaliteit door een grote aanpassing in het programma. Dat dit kan gebeuren en dat je alles moet doen om het voorkomen, daarvan moet je je dan ook altijd bewust zijn. Om de grootste problemen te voorkomen is het bittere noodzaak een zéér recente backup bij de hand te hebben. De backup bevat elke dag alle gebruikersprogramma's, alle bestanden en alle sources. Het tweede wat je moet doen is de voorgenomen wijziging eerst goed doorspreken

met je contactpersoon. Is het de investering van tijd en geld wel waard? Zijn er bedrijfsmatige veranderingen te verwachten waardoor de gewenste aanpassing niet nodig zou zijn? Is het wel ècht nodig? Kan het niet met een standaard formuliertje, in MS-Word gemaakt of zo, worden opgelost? Als besloten wordt een wijziging tóch uit te voeren, is het van belang inzicht te krijgen in de moeilijkheden die kunnen ontstaan en dus vermeden of opgelost dienen te worden.

Kleine wijzigingen zoals een tekstje aanpassen of een schermveld iets verplaatsen kunnen vaak zonder overleg worden uitgevoerd. Zomaar een veld aan een bestand toevoegen of uit een bestand verwijderen kan echter vervelende gevolgen hebben. Wanneer er met databases en SQL verbindingen wordt gewerkt, is de impact van een tabelaanpassing niet zo groot, omdat databases op veldnaam worden benaderd en niet, zoals bij 3GL, het hele record lezen om dan een veld aan te kunnen spreken. Dat is het voordeel van dit databasesystemen. Als er dus een veld bijkomt, wijzigt of afvalt, is dat alleen van invloed op de programma's die dit veld gebruiken:

```
Select BANAM2, BASTAD
From 'Bestand A'
Where ...
```

In 3GL ontwikkelomgevingen wordt de layout van een record volledig in het programma opgenomen op het moment van compilatie. Zolang er dan niet aan de lengte van de velden of de plaats in het record van de velden wordt gesleuteld, is er niets aan de hand. Bij talen die met recordlayouts werken (C, Basic, SL) is het verstandig wanneer je een veld verwijderd, de vrij gekomen plaats niet te verwijderen dan verschuiven de gegevens binnen het record niet. Wordt er echter een veld tussengevoegd, dan verschuiven er wel gegevens binnen het record en veranderd de recordlengte ook. Dan moeten alle 3GL programma's die dat bestand benaderen opnieuw worden gecompileerd, zodat ze beschikking krijgen over de nieuwe bestandsindeling. Hieronder volgt een voorbeeld ter ondersteuning van wat wordt bedoeld:

```
R BSTA
  BANAM1          25 A
  BANAM2          25 A
  BASTRT          25 A
  BASTAD          25 A
                  -----
                  100 A
```

Voorbeeld 63: Oorspronkelijke record indeling

```
R BSTA
  BANAM1          25 A
  XXXXX1          25 A
  BASTRT          25 A
  BASTAD          25 A
                  -----
                  100 A
```

Voorbeeld 64: Record indeling met verwijdering door middel van hernoemen

In bovenstaand voorbeeld is op de plaats van het veld BANAM2 een ander veld gekomen

met een naam die aangeeft dat die plek vrij is om voor wat anders te gebruiken. In deze situatie hoeft je alleen de programma's aan te passen die het oude veld gebruiken.

```
R BSTA
  BANAM1          25 A
  BASTRT          25 A
  BASTAD          25 A
                  -----
                  75 A
```

Voorbeeld 65: Record indeling met verwijderde velden

In dit voorbeeld is het veld echt weggehaald, daardoor is de recordlengte veranderd en zijn de velden BASTRT en BASTAD 25 posities verschoven. Wanneer je nu niets aan programma's zou doen die het verwijderde veld niet gebruiken maar het bestand wel lezen en er naar schrijven, dan houden die programma's de oude record layout vast en krijgen de inhoud van BASTAD op de nieuwe positie in het veld BASTRT op de oude positie. Gelukkig genereren deze programma's een fout bij het *schrijven* naar dit bestand: onjuiste recordlengte (*invalid recordsize*). Het *lezen* van records uit het bestand kan onder sommige omstandigheden foutloos plaatsvinden omdat het in te lezen record korter is dan de in het programma opgenomen recordlayout. OS/400 heeft dit probleem ondervangen door op het moment van compilatie een controlegetal uit het bestand in het programma-object op te nemen. Dit controlegetal wordt gemaakt op het moment dat het bestand wordt gemaakt. Wanneer er dan iets cruciaals aan een bestand veranderd, veranderd het controlegetal mee en geven alle niet opnieuw gecompileerde programma's waarbij het ingesloten controlegetal op het moment van openen van het bestand niet meer overeenkomt met het nieuwe controlegetal, een foutmelding. Wanneer je dus een programma bent vergeten te compileren of aan te passen, kom je er vanzelf achter. Het getuigt echter niet van kwaliteit als je zo'n afwachtende houding aanneemt. Daarom moet je bij een verandering aan een bestand altijd echt alle routines en alle programma's die dit bestand benaderen opnieuw compileren en linken. Tja, en welke dat dan allemaal zijn?

Aanpak

Een advies: begin het aanpassen van achter naar voren. Daarmee wil ik zeggen dat je het software pakket eerst moet voorbereiden op de gevolgen die de komende voorgenomen aanpassing met zich mee zal gaan brengen. Vaak is het zo dat er zoveel moet worden aangepast dat er geen goed overzicht meer te behouden is. En om dat te voorkomen moet je eerst alles wat problemen kan geven of duidt op onzinnig gebruik, aanpassen en weer werkend krijgen zodat het dagelijkse gebruik ervan er niet onder lijdt. Vervolgens is het zaak de aanpassing te analyseren en uit te werken zodat er zichtbare brokken ontstaan die losstaand van elkaar, doch wel ná elkaar, kunnen worden uitgevoerd èn worden geactiveerd. Voordeel is dat je goed in de gaten kan houden of het hele softwarepakket nog steeds optimaal functioneert, dat de aanpassingen op zich steeds overzichtelijker worden omdat je er steeds een klein stukje van kunt schrappen, en dat de gebruikers langzaam aan de eventuele nieuwe functionaliteit kunnen wennen.

Zoeken

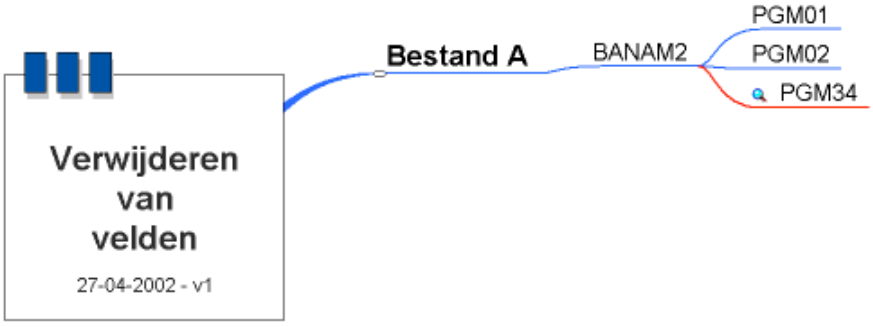
Het meest onmisbare gereedschap voor de programmeur is de zoekfunctie van de ontwikkelomgeving, of soms nog beter, die van het besturingssysteem waaronder geprogrammeerd wordt. Vele programmeurs maken gebruik van ontwikkelgereedschappen die cross-reference lijsten maken en andere lijsten over het gebruik van je objecten produceren. Bij al deze gereedschappen ben je natuurlijk afhankelijk van de werkwijze en uitvoer die het gereedschap voorschrijft en van de programmeurs die het gereedschap hebben gemaakt. Echter, ze zijn vaak onhandig in gebruik en geven een overdaad aan informatie waarom je niet hebt gevraagd. Een doodsimpele zoekfunctie die naar tekst ín bestanden kan zoeken, kan je ondersteunen bij het zoeken naar het gebruik van bestanden, tabellen, velden en programma's. De resultaten van een zoekactie moeten natuurlijk kunnen worden opgeslagen als platte tekst, voor latere verwerking door conversieprogramma's, bijvoorbeeld.

De aangewezen programma's voor het, ook voor anderen begrijpelijk, documenteren van software of aanpassingstrajecten, vooral voor het weergeven van allerlei verbindingen tussen objecten onderling, zijn de *outliners*. De uitvoer die een outliner produceert is ook door niet-automatisserders te volgen wegens de eenvoudige schematechniek. Je kan de *outline view* van MS-Word gebruiken of een andere goede outliner zoals MindManager, welke te prefereren is boven MS-Word wegens de grafische weergave van de onderlinge relaties en vertakkingen.

Hieronder volgen voorbeelden van een MS-Word Outline en een MindManager Map ter voorbereiding van het verwijderen van het veld BANAM2. Alle programma's zijn doorzocht naar het gebruik van dit veld en daar waar het gebruik nader onderzoek behoeft zijn de programma's voorzien van een symbool of kleur:

- **Verwijderen van velden**
  - ***Bestand A***
    - **BANAM2**
      - **PGM01**
      - **PGM02**
      - **PGM34**

Voorbeeld 66: Documenteren van een uit te voeren wijziging met MS-Word



Vinden

Wat moet je allemaal doen om een verandering foutloos en voor de gebruiker nauwelijks merkbaar uit te voeren? Doorzoek alle sources naar het gebruik van te wijzigen objecten. Door goed te zoeken, en vóór alles weten wáár je moet zoeken, vindt je echt alles: programma'a, schermen, formulieren, hulpvelden, bestanden, etc.. Het kan een leuke training zijn voor je probleemoplossende capaciteiten.

Het zoeken naar het ene kan leiden tot het vinden van het andere. Voor elk andere dat je vindt, moet je de zoek-cyclus herhalen en wel net zolang tot er geen nieuwe objecten meer opduiken. Een project waarbij deze aanpak zeer waardevol is gebleken is de overgang van 1999 naar 2000, waarvoor de uitbreiding van datumvelden nodig was zodat de eeuw er in kon passen. In heel veel softwarepakketten waren de datumvelden toentertijd 6 posities en bevatten een datum als jaar-maand-dag (JJMMDD) of of dag-maand-jaar (DDMMJJ). Ze moesten echter 8 poities worden om de volledige datum, dus met eeuw, als JJJJMMDD of DDMMJJJJ te kunnen bevatten. Met deze grootschalige aanpassing vindt je routines voor datumcontrole die hun eigen hulpvelden gebruiken, alfanumerieke velden waarin data geformatteerd worden met een streepje er tussen, periode velden die enkel uit een jaar en maand bestaan, etc., etc.. Al deze velden moesten met twee posities worden uitgebreid. Echter, je moet ook practisch denken als programmeur. Daarom besluit je om in sommige programma's de wijzging níet door te voeren omdat dat het traject te veel ophoudt en het voor de bedrijfsvoering niet storend is als je het achterwege laat.

Wanneer je alle, maar dan ook echt alle, bronnen van het softwarepakket hebt doorzocht naar het gebruik van een object, moet je als eerste de bestanden aanpassen, als die er zijn, en in een testdirectory of bibliotheek klaarzetten. Om gegevens uit een oud bestand in een nieuw bestand te krijgen, moet je meestal een conversieprogramma schrijven. Op de AS/400 kan je met CPYF FMTOPT(*MAP) al vaak uit de voeten om de oude data naar de nieuwe vorm te converteren, zodat er geen conversieprogramma nodig is.

In een niet-database omgeving waar met record-includes wordt gewerkt, zoals RMS op de VAX en de platte bestanden onder DOS en UNIX, is het altijd verstandig bij een bestandsaanpassing gelijk wat extra ruimte te reserveren in het bestand, voor toekomstige aanpassingen. Deze lege ruimte in een bestand noemt men een *filler* (vuller) waarmee je voorlopig het probleem van een veranderende recordlengte kan voorkomen. Neem niet zuinig 10 tekens extra, nee, reserveer gelijk 100 of 200 tekens extra. De extra ruimte is zó verbruikt als je adres- of e-mailvelden toe moet voegen, want die vragen redelijk wat ruimte. Nog een tip: een recordlayout kan gebruikt worden voor meerdere bestanden! Vergeet dit niet te controleren en vergeet vooral niet die andere bestanden ook te converteren.

Aanpassen

Vervolgens komt de grote klus: het aanpassen van de programma's, schermen en afdrukken. Wanneer je alles hebt doorzocht, houdt je een lijst over met daarin alle aan te passen objecten. En al denk je uit je hoofd te weten dat bepaalde programma's op de lijst

eigenlijk 'niets met het object van doen hebben', is het toch verstandiger ze niet van de lijst te schrappen. Je vergeet de werking van dingen vaak sneller dan je denkt. Als je lijst compleet is, sorteer hem op naam van de aan te passen objecten, print hem uit, leg hem naast je en vink steeds het object af dat je hebt aangepast. Het nadeel van deze manier van werken is dat je je, als het om veel objecten gaat, al gauw een soort lopendeband werker voelt. Echter wegen de voordelen op tegen dit nadeel, want je bent tijdens het aanpassen met al je gedachten bij het thema, bij één object en de werking ervan, en dat, als je verderop in de lijst weer in hetzelfde objekt moet zijn, je het al een beetje in je hoofd hebt zitten en gelijk ziet of je op een eerder moment niet wat bent vergeten aan te passen. Of misschien wel verkeerd hebt aangepast. Je kan er namelijk van op aan dat wanneer je in één object gelijk álles probeert aan te passen, je steeds weer terug moet keren naar de eerste aanpassing om te kijken of je die wel goed hebt gedaan. Vervolgens schieten er allerlei andere 'oh jee, heb ik daar wel aan gedacht?' gedachten door je hoofd met als resultaat dat je er zeker twee keer langer over doet.

De besproken methode werkt nog mooier als je een grote aanpassing moet uitvoeren, waarbij het veel objecten betreft, en dat je die met collega's samen doet. Ieder van je collega's krijgt dezelfde lijst met aan te passen objecten, echter ieder begint op een andere plek in de lijst en daardoor met een ander object. Het grote voordeel is natuurlijk dat het werk is verdeeld over meerdere mensen die allen aan hetzelfde werken en dat het daardoor sneller af kan zijn. Echter, de kracht van de werkwijze ligt hem hierin dat anderen jouw aanpassingen zien als ze voor hun object in een source moeten zijn waar jij al in bezig bent geweest. En jij ziet hun aanpassingen als zij eerder in de source zijn geweest dan jij. Mijn ervaring met dit soort klussen is, omdat je zo alert bent in zo'n periode, dat je zelfs andere fouten in de software ontdekt!

Wijk tijdens de sessie nooit af door te veel zijdelingse aanpassingen te gaan doen die je ook later nog kan doen, want voor je het weet ga je te ver en kan je niet meer terug zonder steeds maar meer te moeten aanpassen. En dan kan je later niet meer achterhalen wat de oorzaak is van een eventueel optredende fout. Maak er, in plaats van het aan te gaan passen, een melding van in je foutenregistratiesysteem, dan komt het vanzelf op de agenda. Dus: eerst het ene project afronden, dan pas met het volgende beginnen.

Iets wat er al is compleet vervangen

Het vervangen van bestaande software door nieuwe kan een hele module van een softwarepakket betreffen, maar ook een enkel programma. Uiteraard is de hier te volgen werkwijze niet veel anders ten opzichte van de eerder besproken situaties. Waar je hier vooral op moet letten, is wat er van de reeds bestaande functionaliteit moet worden bewaard en, eventueel vernieuwd, moet worden ingebed in de nieuw te bouwen software. In deze situatie is het daarom ook heel belangrijk mensen ter beschikking te hebben die de bestaande software door en door kennen en weten hoe die binnen het bedrijf wordt toegepast. Er is niet altijd documentatie aanwezig over de werking van bepaalde programma's. Wanneer je deze dingen mist, moet je zelf op ontdekking gaan en binnen het bedrijf net zolang door blijven vragen tot je er zeker van bent dat je weet wat mag vervallen en wat overgenomen moet worden. Ga vooral niet na één keer geen antwoord te hebben gehad gewoon beginnen met programmeren en zien waar het schip strand. Dat wordt zeer zeker niet gewaardeerd! Zoek ook in de sources naar commentaar dat vorige programmeurs hebben achtergelaten. Het zal je maar gebeuren dat je oude software vervangt door iets nieuws waar je weken aan hebt gewerkt en het halve

bedrijf valt over je heen omdat er iets is verdwenen waarvan iedereen voetstoots aannam dat het ook in de nieuwe software weer aanwezig zou zijn.

Iets maken wat er nog helemaal niet is

Over nieuwbouw kan ik kort zijn: heerlijk! Zorg voor een goed, werkbaar ontwerp of, nog beter, vele kleine ontwerpen waar je zó mee kunt starten. Vele kleine ontwerpen krijg je door een groot ontwerp, welke het geheel dekt, op te splitsen in deeltaken. Elke deeltaak moet zo afgebakend zijn dat het, desnoods door extra werkzaamheden, geen open einde heeft. Zo kan je dan al delen van het geheel activeren voor de gebruikers. Ze kunnen zo volgen wat je maakt en tijdig aan de bel trekken als ze tot nieuwe inzichten zijn gekomen, wat juist vaak gebeurt door het nieuwe waar ze al mee aan de slag hebben kunnen gaan.

Een ander, zeer groot, voordeel van op zichzelf staande deeltaken is dat je als drukbezette IT-er eigenlijk niet eens tijd hebt om aan één stuk door aan een groot nieuwbouw project te werken, maar wel nèt genoeg tijd hebt om steeds een klein stuk af te kunnen ronden. Het zijn de 'tussendoortjes' die je gedachten van je project af leiden. Onder de 'tussendoortjes' worden de gebruikelijke vragen van gebruikers en familie, de collega's en klanten verstaan. Door kleine ontwerpen van korte duur blijft het voor jezelf overzichtelijk en kan je ook makkelijker de draad weer oppakken als je er een week of zo tussenuit bent geweest.

Een ander probleem waarvoor de meeste IT-ers zich gesteld zien is de druk van de oplevertermijn. Het moet altijd allemaal het liefst eergisteren klaar. Wees echter standvastig en laat je niet opjagen: eerst nadenken over iets nieuws. Hoe ga je het oppakken, waar zitten de haken en ogen, hoe en waar moet het aan de bestaande software worden gekoppeld. Zitten er ingewikkelde processen in en wat is de meeste efficiënte methode om ze te programmeren. Het kan best wel een paar weken duren voor je de juiste oplossing of werkwijze voor ogen hebt. Men zegt niet voor niets dat beter een paar nachtjes over nieuwe ideeën kunt slapen! Laat het project door je hoofd spelen terwijl je je aan de dagelijkse beslommeringen wijdt. Er komt toch wat 'belangrijkers' tussendoor, waardoor de start van andere projecten opschuift. Maak je dus nooit heel erg druk over opleverdata, het blijkt dat mensen vaak langer zonder iets kunnen dan ze in eerste instantie dachten. Lever liever programmatuur af die gelijk of minstens binnen een dag na oplevering voor de volle honderd procent draait dan dat je moet gaan haasten omdat er iemand zit te piepen en dat je daardoor dingen maakt die maar half af zijn, waardoor er zéker klachten komen van de zijde van de opdrachtgever en je daardoor jezelf in een slecht daglicht stelt.

Kortom, wees die koele IT-er die als een rots in de branding alles in alle rust aanpakt wat er voor zijn voeten wordt geworpen, hoe hectisch de wereld om hem heen ook reageert, en nooit zegt 'dat kan niet'. Dan, en alleen als blijkt dát je ook voor alles een hele goede oplossing hebt, alleen dan dwing je respect af en krijg je het ook.

APPENDIX A – SOURCES

Schilderijen ophangen – deuren.c

```
#include "c:\applic~1\ms\c\qc\inc\stdio.h"

int hup;
int nivo;

main() {
        hup = 0;
        nivo = 0;

        open_deur(zoek_deuren());
}

open_deur(deuren)
int deuren;
{
        nivo = nivo + 1;

        while (deuren) {
                open_deur(zoek_deuren());
                deuren = deuren - 1;
        }

        nivo = nivo - 1;
        hang_schilderij();

        return;
}

int zoek_deuren () {
        int deuren;

        hup = hup + 1;

        switch (hup) {
                case 1:
                        deuren = 2;
                        break;
                case 3:
                        deuren = 3;
                        break;
                default:
                        deuren = 0;
                        break;
        }

        return (deuren);
}

hang_schilderij() {
        printf ("\nSCHILDERIJ op nivo %d.", nivo);
        return;
}
```

Schilderijen ophangen – deuren.sl

```
.main   DEUREN
```

```
external function
        zoek_deuren,    d

common
        hup,    d1
        nivo,   d1

.proc
        open (1, i, 'tt:')

        hup = 0
        nivo = 0

        xcall open_deur(%zoek_deuren)

        close 1
.end



.function zoek_deuren

record
        deuren, d2

common
        hup,    d1

.proc
        hup = hup + 1

        using (hup) select
        (1),    deuren = 2
        (3),    deuren = 3
        (),     deuren = 0
        endusing

        freturn (deuren)
.end



.subroutine hang_schilderij

common
        nivo,   d1

.proc
        display (1, 10,13, "SCHILDERIJ op nivo ", %string(nivo), ".")
        xreturn
.end



.subroutine open_deur, reentrant

deuren, d       ; parameter

external function
        zoek_deuren,    d

common
        nivo,   d1

.proc
        nivo = nivo + 1
```

```
        while (deuren) begin
                xcall open_deur(%zoek_deuren)
                deuren = deuren - 1
        end

        nivo = nivo - 1
        xcall hang_schilderij

        xreturn
.end
```

Schrikkeljaartest – leapyear.html

```
<html>
<head>
        <title>Schrikkeljaar test</title>
        <script language="JavaScript">

        function compute(form) {
                jjjj   = form.year.value;

                if (((parseInt(jjjj / 100) * 100) != jjjj) &&
                    ((parseInt(jjjj /   4) *   4) == jjjj))
                        form.result.value = "SCHRIKKELJAAR!";
                else if ((parseInt(jjjj / 400) * 400) == jjjj)
                        form.result.value = "SCHRIKKELJAAR!";
                else    form.result.value = "GEEN SCHRIKKELJAAR!";
        }

        </script>
</head>

<body bgcolor="#FFFFCC">

<table border=2 width=400>
<tr><td>
        <center>
        <h1>Schrikkeljaar test</h1>
        <FORM>
                Toets een jaartal in (1900, 1996, ...):
                <INPUT TYPE="text" NAME="year" SIZE=4>
                <p>
                <INPUT TYPE="button" VALUE="Bereken"
                        ONCLICK="compute(this.form)">
                <br><br><hr><br>
                Resultaat:
                <INPUT TYPE="text" NAME="result" SIZE=25 ><BR>
        </FORM>
        </center>
</td></tr>
</table>

</body>
</html>
```

Talstelsels – talstelsels.html

```
<html>
<head>
        <title>Talstelsels</title>
        <script language="JavaScript">
```

```
function naarradix(getal) {
        teken = "";

        if (getal < 10)
                teken = getal;
        else if (getal == 10)
                teken = "A";
        else if (getal == 11)
                teken = "B";
        else if (getal == 12)
                teken = "C";
        else if (getal == 13)
                teken = "D";
        else if (getal == 14)
                teken = "E";
        else if (getal == 15)
                teken = "F";
        else if (getal == 16)
                teken = "G";
        else if (getal == 17)
                teken = "H";
        else if (getal == 18)
                teken = "I";
        else if (getal == 19)
                teken = "J";
        else if (getal == 20)
                teken = "K";
        else if (getal == 21)
                teken = "L";
        else if (getal == 22)
                teken = "M";
        else if (getal == 23)
                teken = "N";
        else if (getal == 24)
                teken = "O";
        else if (getal == 25)
                teken = "P";
        else if (getal == 26)
                teken = "Q";
        else if (getal == 27)
                teken = "R";
        else if (getal == 28)
                teken = "S";
        else if (getal == 29)
                teken = "T";
        else if (getal == 30)
                teken = "U";
        else if (getal == 31)
                teken = "V";

        return(teken);
}

function rekenen(dec, radix) {
        txt = "";

        while (dec >= radix) {
                nxt  = parseInt(dec / radix);
                rest = dec - (nxt * radix);
                txt  = naarradix(rest) + txt;
                dec  = nxt;
        }

        txt = naarradix(dec) + txt;
        return(txt);
}
```

```
        function rekenom(form) {
                /* Hexadecimaal */
                form.hexa.value = rekenen(form.dec.value, 16);

                /* Octaal */
                form.octa.value = rekenen(form.dec.value, 8);

                /* Binair */
                form.bina.value = rekenen(form.dec.value, 2);

                /* Ditriacontaal */
                form.ditr.value = rekenen(form.dec.value, 32);
        }

        </script>
</head>

<body bgcolor="#FFFFCC">

<table border=2 width=400>
<tr><td>
        <center>
        <h1>Talstelsels</h1>
        <FORM>
                Toets een decimaal getal in:
                <INPUT TYPE="text" NAME="dec" SIZE=10>
                <p>
                <INPUT TYPE="button" VALUE="Reken om"
                        ONCLICK="rekenom(this.form)">

                <br><br><hr><br>

                <table border=0>
                <tr><td>
                  Hexadecimaal:
                </td><td>
                  <INPUT TYPE="text" NAME="hexa" SIZE=25 ><BR>
                </td></tr>
                <tr><td>
                  Octaal:
                </td><td>
                  <INPUT TYPE="text" NAME="octa" SIZE=25 ><BR>
                </td></tr>
                <tr><td>
                  Binair:
                </td><td>
                  <INPUT TYPE="text" NAME="bina" SIZE=25 ><BR>
                <tr><td>
                <tr><td>
                  Ditriacontaal:
                </td><td>
                  <INPUT TYPE="text" NAME="ditr" SIZE=25 ><BR>
                <tr><td>
                </table>
        </FORM>
        </center>
</td></tr>
</table>

</body>
</html>
```