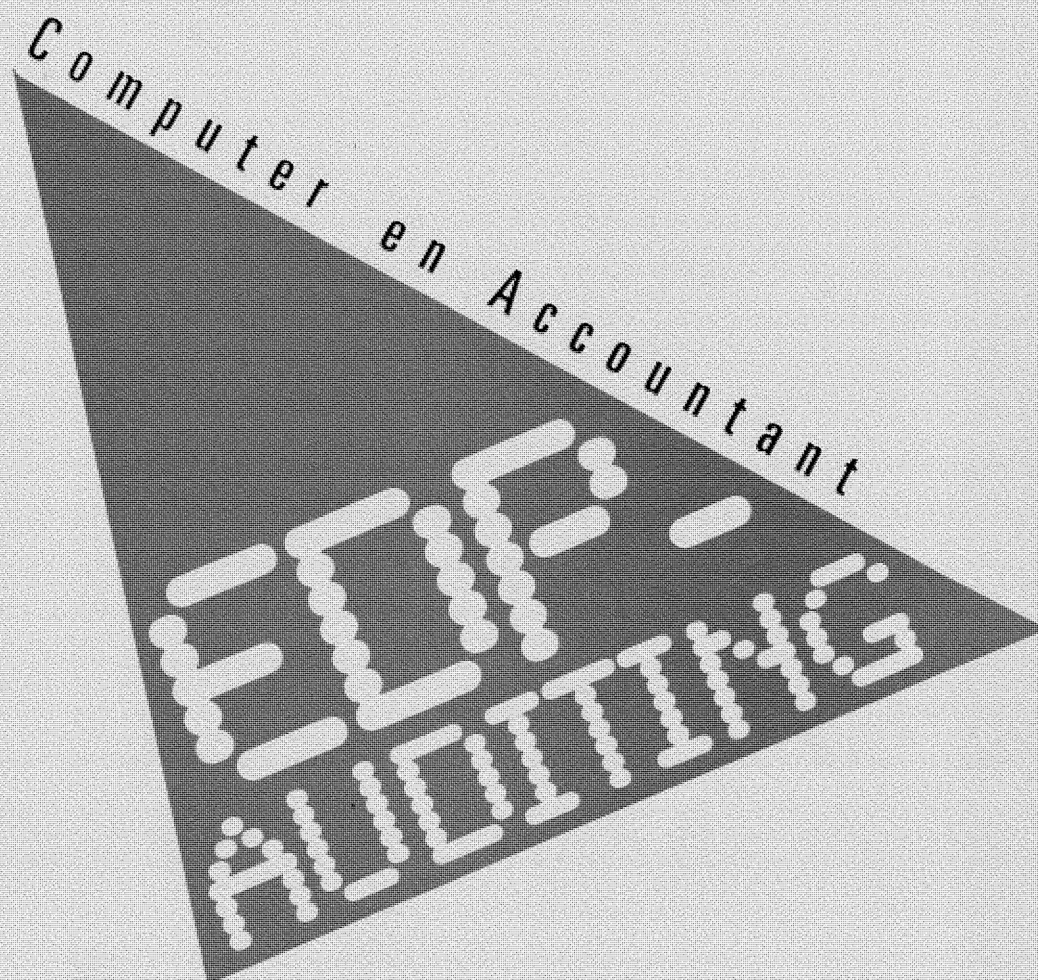
KPMG Klynveld EDP Audit Services

88/2
lente/zomer 1988

COMPACT



Special

van de sectie

Software Engineering

Computer en Accountant

Inhoudsopgave

- Van de redactie 1
- De sectie Software Engineering, een inleiding 5
  H. Veenman Chartered Engineer.
- Software Engineering 10
  H. Veenman Chartered Engineer en Ing. L.J.M.W. Gielen
- Het testen van software 19
  O. Kluyt
- UNIX 26
  Ing. A. van der Vlist en Ing. J.C. van Winkel RI
- Computervirussen 34
  Ing. J.C. van Winkel RI
- Objects 41
  Ing. L.J.M.W. Gielen
- HyperCard 51
  J. Schalk
- Programmeertheorie 56
  J. Schalk
- Het AppleTalk netwerk, een beschouwing 61
  J.L. Ramos Najera
- PS/2 - OS/2 69
  Ir. J. de Graaff en Drs. D.J.P. Witte
- Elektronisch betalen, de betaalpas 78
  Ing. J. Rotteveel
- Boeken 83

Nummer 46
ISSN 0920-1645

lente/zomer 1988

Van de Redactie

Als U de inhoudsopgave van deze Compact leest, bekruipt U ongetwijfeld de vraag: "Wat doet de accountant hier nu eigenlijk mee?".
Het ontwikkelen van software in de breedste zin van het woord, dát is de boodschap die de sectie Software Engineering ons duidelijk maakt.

Voor U ligt het werk van de sectie Software Engineering, onderdeel van Klynveld EDP Audit. Wij nodigen U, geachte lezer, uit kennis te nemen van dit speciale themanummer.

Per artikel zal door middel van een waarderings-'raampje' aangegeven worden hoe actueel, diepgaand en/of educatief een artikel is.

De sectie Software Engineering, een inleiding
H. Veenman Ch. Eng.

laag			hoog	
			X	actueel
	X			diepgaand
			X	educatief

Begonnen als een handvol enthousiaste programmeurs en studenten is de sectie Software Engineering in vijf jaar uitgegroeid tot een professioneel team van software-bouwers en onderzoekers.
In dit artikel leest u, als inleiding op de volgende artikelen in deze speciale Compact, over deze evolutie en over de activiteiten die nu door de sectie worden uitgevoerd.

Software Engineering
H. Veenman Ch. Eng. en Ing. L. J.M.W. Gielen

laag			hoog	
		X		actueel
	X			diepgaand
		X		educatief

In deze Compact special mag een artikel over software engineering niet ontbreken. Dit artikel verklaart de term software engineering, geeft een kort overzicht van de geschiedenis en belicht een aantal belangrijke aspecten van de discipline.

Het testen van software
O. Kluyt

laag			hoog	
		X		actueel
		X		diepgaand
		X		educatief

Dit artikel wil een overzicht geven van een aantal strategieën en methodieken, die kunnen worden gehanteerd bij het testen van software. Het testen dient een integraal onderdeel te zijn van het ontwerp en de ontwikkeling van een systeem. In de ontwikkeling van een systeem kan, onafhankelijk van de gevolgde methode, een aantal fasen worden onderscheiden. Tijdens elke fase dient getest te worden of het systeem de gewenste functionaliteit bevat en of het systeem aan de overige eisen voldoet. In dit artikel zal de nadruk liggen op het testen van de implementatie; het testen van de specificaties en het conceptueel ontwerp zullen slechts kort worden behandeld.

UNIX
Ing. A. van der Vlist en Ing. J.C. van Winkel RI

laag		hoog		
		X		actueel
			X	diepgaand
			X	educatief

Langzaam maar zeker is UNIX[1] volgroeid tot een volwassen besturingssysteem. Via universiteiten en later ook het bedrijfsleven heeft dit tamelijk eigenzinnige besturingssysteem zijn weg niet alleen naar grote technische systemen maar ook naar administratieve omgevingen en personal computers gevonden. Dit artikel wil ingaan op het ontstaan van UNIX, globaal aangeven hoe het besturingssysteem in elkaar zit en hoe de beveiliging is geregeld. Ook wordt aandacht geschonken aan onderwerpen als standaardisering, derivaten, communicatie en 'hacking'.

Computervirussen
Ing. J.C. van Winkel RI

laag		hoog		
			X	actueel
		X		diepgaand
			X	educatief

In dit artikel zal een uiteenzetting worden gegeven van de eigenschappen van computervirussen; hoe ze werken, hoe computers besmet worden met een virus en hoe dit voorkomen danwel genezen kan worden.

Objects
Ing. L.J.M.W. Gielen

laag		hoog		
		X		actueel
			X	diepgaand
		X		educatief

De ontwikkeling van programmeertalen en -technieken staat niet stil. Met de regelmaat van een langzaam tikkende klok wordt melding gemaakt van nieuwe ontwikkelingen. Sommige daarvan zijn gebaseerd op het gebruik van 'objects'. Het werken met objects, het object georiënteerd programmeren, is hard op weg zich een belangrijke plaats te veroveren in de informatie-technologie. In dit artikel over object georiënteerd programmeren wordt ingegaan op de algemene begrippen en de daarbij gebruikte terminologie. Voor meer diepgaande beschrijvingen wordt verwezen naar de literatuurlijst ter afsluiting van dit artikel.

HyperCard
J. Schalk

laag		hoog		
			X	actueel
	X			diepgaand
			X	educatief

"HyperCard is a software erector set", Bill Atkinson.
Bovenstaand citaat van de maker van HyperCard (HC) geeft aan hoe men dit pakket het best kan beoordelen: het is een constructieset voor Apple Macintosh-applicaties. Het met iedere Macintosh gratis meegeleverde pakket geeft aan de gebruiker de mogelijkheid om zijn eigen toepassingen te ontwikkelen. In dit artikel wordt gepoogd een indruk te geven van de mogelijkheden van HC. Hierbij wordt extra aandacht besteed aan voor de accountant belangrijke gebieden zoals protectie van HC-applicaties.

---

1 *UNIX is een geregistreerd handelsmerk van Bell Labs (Bell Labs is het research center van AT&T).*

Programmeertheorie
J. Schalk

laag			hoog	
		X		actueel
			X	diepgaand
	X			educatief

Programmeren wordt door velen nog gezien als een kunst die niet in logische of mathematische wetten te vangen valt. Dit artikel probeert hierin verandering te brengen. Het geeft aan hoe men ook op formele wijze tegen het oplossen van programmeringsproblemen kan aankijken.

Het AppleTalk-netwerk, een beschouwing
J.L. Ramos Najera

laag			hoog	
			X	actueel
		X		diepgaand
		X		educatief

Nu de Apple Macintosh zijn intrede heeft gedaan als hulp in de accountantscontrole van KPMG, zal het gebruik van het AppleTalk-netwerk een steeds vaker voorkomend fenomeen zijn. Het begint met een tamelijk eenvoudige opstelling van Macintoshes en laserprinters en groeit dan in vele gevallen uit tot complete netwerken. Gezien het relatief transparant gebruik van het AppleTalk-netwerk zouden we gemakkelijk kunnen vergeten dat hier een doordachte netwerkarchitectuur achter staat met een zeer groot aantal protocollen. Een beschouwing waard.

PS/2 - OS/2
Ir. J. de Graaff en drs. D.J.P. Witte

laag			hoog	
			X	actueel
	X			diepgaand
		X		educatief

IBM is in 1987 met vier nieuwe modellen, onder de naam Personal System 2™ (PS/2) op de personal computermarkt gekomen. Daarnaast is in samenwerking met Microsoft een nieuw besturingssysteem ontwikkeld dat in staat is de hardware-faciliteiten van PS/2 ten volle te benutten. Dit systeem heeft men Operating System 2™, of kortweg OS/2 genoemd. In dit artikel zal een overzicht worden gegeven van de meest in het oog springende eigenschappen van zowel het besturingssysteem als de hardware. Aan het eind wordt tevens een overzicht geboden van de toekomstige ontwikkelingen op dit gebied.

Elektronisch betalen, de betaalpas
Ing. J. Rotteveel

laag			hoog	
			X	actueel
		X		diepgaand
			X	educatief

Er heerst nogal wat beroering op het gebied van het elektronisch betalen in Nederland. Het gebruik van zowel magneetkaarten als betaalpassen zou niet betrouwbaar zijn. Gelijktijdig wordt de chip-kaart als oplossing voor deze problematiek gepresenteerd. In dit artikel zal kort worden ingegaan op de in de media geschetste problematiek van het elektronisch betalen en zullen mogelijke oplossingen besproken worden.

Tot slot treft u in de rubriek "Boeken" een drietal referenties aan.

Compact (R) is een uitgave van
KPMG Klynveld EDP Audit

Deze informatie is in de eerste plaats bestemd voor diegenen, die in de algemene controlepraktijk werkzaam zijn van KPMG Klynveld Kraayenhof & Co. De in dit tijdschrift weergegeven meningen mogen niet altijd gezien worden als officiële zienswijzen van KPMG Klynveld EDP Audit. De in rubrieken besproken tijdschriften, boeken en artikelen worden soms geheel opgenomen of verkort aangehaald, tevens als regel voorzien van commentaar.

Redactie:
A.W. Neisingh
Prof. D. Steeman
H. Weerd
H.J.M. van der Wielen (secr.).

Special Redactie:
Ing. A. van der Vlist
O. Kluyt
Drs. D.J.P. Witte.

Kopij kunt u inleveren bij de
secretaris van de redactie.

Adres:
World Trade Center Amsterdam
Strawinskylaan 1257
Toren D 11e etage
1077 XX Amsterdam
Postadres:
Postbus 72001
1007 TB Amsterdam

© 1988 KPMG Klynveld EDP Audit

Nadruk van deze uitgave is toegestaan mits met bronvermelding.
Van overgenomen artikelen uit andere bladen blijven de rechten berusten bij hun uitgever/auteur. Wij verwijzen steeds naar de vindplaatsen.
ISSN 0920-1645

Indien u belangstelling heeft voor meerdere exemplaren kunt u deze aanvragen bij de secretaris van de redactie, evenwel zolang de voorraad strekt (telefoon 020 - 5461912).

De sectie Software Engineering, een inleiding

door: H. Veenman Ch. Eng.

Begonnen als een hand vol enthousiaste programmeurs en studenten is de sectie Software Engineering in vijf jaar tijd uitgegroeid tot een professioneel team van software bouwers en onderzoekers.
In dit artikel leest u, als inleiding op de volgende artikelen in deze speciale Compact, over deze evolutie en over de activiteiten die nu door de sectie worden uitgevoerd.

1. Historisch overzicht

1982. Zowel binnen de Nederlandse KKC-organisatie als internationaal in KMG-verband vraagt men zich af wat de aanstormende microcomputer voor de accountantspraktijk kan betekenen. Twee studenten van de Eindhovense HTS Informatica krijgen de opdracht om een bijdrage te leveren tot een antwoord op deze vraag. Als stagiairs zijn zij gedurende dat jaar in dienst bij de toenmalige Automatisering & Controle-groep van KKC. Terwijl één student zich specifiek bezighoudt met de problematiek rond de wijze waarop een microcomputer zich aan de accountant-gebruiker zou moeten presenteren, bestudeert de ander de mogelijkheden om relevante cliëntinformatie, zoals grootboek, voorraad en journaalposten beschikbaar te krijgen voor de microcomputer van de accountant, die al snel tot 'audit-micro' wordt gedoopt.

In februari 1983 wordt door de leiding van de Automatisering & Controle-groep besloten om, mede door de veelbelovende uitkomst van de afstudeerrapporten [SPAP83] [SALE83], een aparte groep op te richten, die zich bezighoudt met de ontwikkeling van software voor de audit-micro. De microgroep is geboren.
Nog in hetzelfde jaar wordt de allereerste versie van het KKC Audit Package ontwikkeld. Het pakket draait op een Altos multi-user computer en biedt de mogelijkheid om steekproeven, selecties en tellingen uit te voeren. Dit pakket is nu bekend als het File Analysis Tool.
In het najaar van 1983 wordt door de internationale KMG-organisatie geadviseerd om binnen de organisatie zoveel mogelijk op IBM Personal Computers of compatibles te standaardiseren, om de uitwisselbaarheid van gegevens en programma's te verhogen. Een simpel overzetten van de bestaande Basic-programmatuur naar de MS/DOS-omgeving van de personal computers blijkt niet mogelijk door verschillen tussen de Basic-dialecten. Omdat ook de prestaties van de in deze taal ontwikkelde programma's te wensen overlaten, geeft deze verplichte ommezwaai de gelegenheid om opnieuw de te gebruiken programmeertaal te overwegen.
Gekozen wordt uiteindelijk voor C, een taal die in een eerdere fase is afgewezen in verband met de slechte leesbaarheid en de grote mate van vrijheid, die de taal biedt. Door strakke programmeringsprocedures [SCHA86] [SCHA87] worden deze nadelen echter weggenomen. Andere talen, die in het onderzoek zijn meegenomen zijn Pascal, Fortran, Cobol en Basic.

Tegelijk met de uitreiking van de eerste 35 draagbare microcomputers aan accountants van KMG Klynveld Kraayenhof & Co. wordt de eerste officiële versie van het File Analysis Tool op 19 juni 1984 gepresenteerd.
Naast FAT worden ook andere pakketten voor gebruik door de accountant ontwikkeld. Te noemen zijn het File Match Tool (FMT), het File Conversion Tool (FCT) en PALET (Process AnaLysing and Editing Tool) [VAKT86].

Al snel na de oprichting van de toenmalige microgroep wordt duidelijk dat door het ontwikkelen van software kennis wordt vergaard en mensen worden aangetrokken, die uniek zijn voor een accountantsorganisatie. Nieuwe gebieden als computer graphics, datacommunicatie, micro-processoren, software-portabiliteit en vele anderen gaan tot de discipline behoren.

Gedurende de jaren 1985 en 1986 wordt geconsolideerd. Bestaande pakketten worden uitgebreid, programmabibliotheken verbeterd en procedures bijgewerkt.
De naam van de sectie wordt gewijzigd in Software Engineering, omdat dat de activiteiten beter weergeeft.
Door een aantal personen wordt meegewerkt aan EDP Audit-opdrachten, die de diepgaande technische kennis van leden van de sectie vereisen. Op deze activiteiten wordt later in deze inleiding ingegaan.

In februari 1987 wordt in het kader van de fusie van KMG met Peat Marwick International besloten om in het vervolg de Apple Macintosh als audit-micro te gaan gebruiken. Er wordt gekozen voor een overgangsfase, waarin de bestaande MS/DOS-pakketten snel worden overgezet naar de Macintosh, zonder rekening te houden met de voor deze machine geldende richtlijnen ten aanzien van programmeertechniek en gebruikers-interface.

In de loop van 1988 zullen alle pakketten op de Macintosh beschikbaar komen; reeds nu is een aantal ervan als testversie voor gebruik beschikbaar.
Dat de sectie de MS/DOS-wereld niet uit haar gezichtsveld heeft verloren, mag blijken uit het artikel over OS/2 - PS/2, elders in deze Compact.

2. Software-ontwikkeling

2.1 Ontwikkelomgeving

Naast de keuze van een programmeertaal is tijdens de heroverweging in 1983 gezocht naar een adequate ontwikkelomgeving. Met ontwikkelomgeving wordt het geheel bedoeld van apparatuur, programmatuur en procedures, dat tijdens de systeemontwikkeling wordt toegepast. Aan de keuze van C als programmeertaal was als voorwaarde verbonden dat strakke procedures ten aanzien van programmering zouden worden ontwikkeld, en dat geautomatiseerde gereedschappen beschikbaar moesten zijn voor het beheersen van de ontwikkelde programmacode. Het besturingssysteem UNIX bleek het meest geschikt. Het biedt veel hulpmiddelen op het gebied van software-ontwikkeling, waaronder editors, compilers, debuggers, syntax- en semantische checkers. Daarnaast is er het Source Code Control System (SCCS) met behulp waarvan de verschillende generaties van programmamodules kunnen worden beheerd. SCCS voorkomt tevens het door meer ontwikkelaars tegelijkertijd wijzigen van een zelfde module. De beschikbaarheid van een scala aan simpele filters en de mogelijkheid om zelf scripts (commando-procedures) te schrijven, maakt UNIX een complete omgeving voor het professioneel ontwikkelen van software.
Het artikel over UNIX in deze uitgave gaat dieper in op de mogelijkheden van dit besturingssysteem.

2.2 Overdraagbaarheid

Er wordt tijdens de ontwikkeling van software veel waarde gehecht aan de scheiding tussen machine-afhankelijke en machine-onafhankelijke programma-onderdelen. De modules worden in aparte bibliotheken beheerd.
In het algemeen kan worden gesteld dat routines, die rechtstreeks communiceren met perifere computeronderdelen, zoals beeldscherm, externe schijf of toetsenbord, afhankelijk zijn van de karakteristieken van dat onderdeel en dus moeten worden aangepast bij vervanging door een ander type. Deze routines worden non-portable genoemd. Rekenroutines en functies, die datastrings manipuleren, zijn daarentegen onafhankelijk van de configuratie en kunnen als portable worden beschouwd.

Voor de tijdelijke conversie van de MS/DOS-pakketten naar de Macintosh is gekozen om ook nu nog deze scheiding aan te houden. Een "non-portable" bibliotheek voor de Mac is daar het gevolg van. Zonder deze aanpak was het niet mogelijk geweest de MS/DOS-programmatuur in één jaar tijd over te zetten.
Een bijkomend voordeel is, dat door deze aanpak de MS/DOS-programmatuur blijft bestaan. Door simpelweg een pakket "aan te linken" met de MS/DOS non-portable library, wordt een MS/DOS-versie aangemaakt. Op deze wijze zullen beide machinelijnen nog een aantal jaren kunnen worden onderhouden, met slechts geringe extra inspanning.

2.3 De programmeertaal

De vroegere programmeringsprocedures van de sectie schreven voor dat niet in standaard C geprogrammeerd mocht worden, zoals dat beschreven is door Kernighan en Ritchie in [KERN78], maar in KKC C. De keuze voor een afwijkende taaldefinitie was gebaseerd op literatuurstudie en enkele praktische onderzoeken.
Door keywords in hoofdletters te schrijven, door strakke indenteringsregels te hanteren en cryptische commando's door meer mnemonische te vervangen, zou beter leesbare en dus beter onderhoudbare code worden geschreven.
Tijdens het gebruik werd langzaam aan duidelijk dat ook nadelen zijn verbonden aan een eigen definitie van de taal. Routines in C, die door leveranciers, tijdschriften en universiteiten worden aangeboden zijn geschreven in standaard C, en dienen dus te worden aangepast alvorens in KPMG-programmatuur te kunnen worden opgenomen.
De op de markt beschikbare gereedschappen voor het opschonen van C-programmatuur en het verhogen van de leesbaarheid (C beautifiers) zijn geënt op standaard C en werken niet voor KKC C. Op veel scholen wordt tegenwoordig C gebruikt voor programmeerpractica, zodat nieuwe software engineers en tijdelijke krachten gewend zijn aan deze taal, en moeite hebben met het lezen en schrijven van KKC C.
Omdat inmiddels ook programma-editors zijn verbeterd, met onder andere automatische correcte indentering, is in het najaar van 1987 besloten om terug te keren tot standaard C.
De officiële standaard van ANSI (American National Standard Institute) wordt gehanteerd, die is afgeleid van de oudere beschrijving van K&R [KERN78] , en door het merendeel van de C compilers als uitgangspunt wordt gehanteerd. Hoewel de standaard taaldefinitie wordt gehanteerd, gelden nog steeds strakke richtlijnen t.a.v. programmeerstijl, -lay-out en documentatie. De toegankelijkheid en de

onderhoudbaarheid van de code is hierdoor gewaarborgd.

2.4 Standaards
Naast de genoemde voorschriften ten aanzien van het programmeren zijn in de beginjaren van SE, ook richtlijnen ontwikkeld voor de gebruikers-interface van de te ontwikkelen programmatuur. Zo is vastgelegd op welke wijze de menugestuurde interactie met de gebruiker moet plaatsvinden en hoe de dialoog tussen mens en computer moet worden gerealiseerd. Zo veel als mogelijk wordt het gebruik van deze richtlijnen gestimuleerd door de beschikbaarheid van standaard routines, die zich geheel volgens deze standaards gedragen. Het gebruik van deze routines in plaats van nieuw te ontwikkelen programmatuur betekent vaak een substantiële verkorting van het programmeertraject.

2.5 Kwaliteitsbewaking

2.5.1 Bibliotheken
In de programmatuur worden momenteel ten aanzien van de kwaliteitsbewaking twee soorten routines onderscheiden, te weten bibliotheekroutines en applicatieroutines. De applicatieroutines zijn specifiek voor één applicatie ontwikkeld. Bibliotheekroutines daarentegen zijn meer algemeen bruikbaar. Een voorbeeld van een bibliotheekroutine is de routine die zorgt voor de presentatie van een keuzemenu op het beeldscherm, volgens de user interface-standaards.

Daar een bibliotheekroutine door meer applicaties gebruikt kan worden, zijn de mogelijke gevolgen van een fout in zo'n routine ook groter dan wanneer het een applicatieroutine betreft. Er is om deze reden een aparte programmabewaarfunctie gecreëerd, die - naast de andere taken als software engineer - de verschillende bibliotheken beheert.

Wanneer wordt besloten dat een bibliotheekroutine moet worden geschreven of aangepast, dan wordt direct iemand aangewezen, die de programmering voor zijn rekening neemt, en een ander, die de routine test.
Het gehele traject van invoering c.q. aanpassing van een bibliotheekroutine wordt begeleid door een formulier, waarop de betrokkenen hun opmerkingen en hun paraaf zetten. Deze begeleidingsformulieren worden centraal bij bibliotheekbeheer bewaard.

Voor de kwaliteitsbewaking van een applicatieroutine is de projectleider van een project verantwoordelijk. In principe vervult hij een zelfde functie als bibliotheekbeheer, echter nu voor de routines die specifiek tot zijn project behoren. Ook deze routines worden onder SCCS beheerd, in een separate applicatie-bibliotheek.

2.5.2 Testen
Nadat uit de verschillende applicatie- en bibliotheekroutines het gewenste systeem is opgebouwd en getest door de leden van het project, worden alle routines met behulp van SCCS "bevroren". Uit deze set routines wordt het systeem door compilatie en linken opnieuw gegenereerd. Deze min of meer definitieve vorm wordt vervolgens binnen Software Engineering getest. Indien de planning dit toelaat zal het systeem ook worden getest door een aantal engineers die niet direct bij het project betrokken zijn. Deze test heet de *alpha-test.*

Wanneer het systeem de alphatest naar tevredenheid heeft doorlopen, krijgt het de status van *betatest*-versie. In deze staat kan het produkt door een beperkte groep gebruikers in de praktijk worden toegepast. Dit dient twee doelen. Op de eerste plaats dient het uitzetten van betatest-exemplaren voor het testen van het pakket in de praktijkomgeving. Een gebruiker test vaak op een andere manier, dan een ontwikkelaar. Daarnaast wordt door het beschikbaar stellen van de betatest-versie vaak een eerste behoefte aan het pakket bevredigd. Door deze combinatie van factoren wordt de proef serieus gedaan.
De betatest-versie wordt tegelijkertijd - of kort daarop - ter acceptatie aangeboden aan de sectie Support & Programming van EDP Audit, die vervolgens uit naam van het Directoraat Vaktechniek een serie *acceptatietests* uitvoert. Nadat ook deze serie met goed gevolg is doorlopen, wordt het pakket officieel vrijgegeven voor gebruik bij de accountantswerkzaamheden.
Mochten tijdens de acceptatiefase fouten worden geconstateerd, die tot verkeerde resultaten kunnen leiden of op een andere wijze storend zijn, dan wordt het pakket teruggestuurd naar de projectleider van Software Engineering.
Hier zal vervolgens diagnose worden gesteld en zullen de fouten worden gelokaliseerd en opgelost. Na opnieuw het systeem getest te hebben, wordt vervolgens een alphatest gedaan, alvorens het pakket opnieuw ter acceptatie wordt aangeboden.
De problematiek van het testen en de kwaliteitsbewaking wordt in de artikelen "Het testen van software" en "Software engineering" verder uitgediept.

3. Overige activiteiten

Software Engineering is een Research and Development (R&D) afdeling. Naast het ontwikkelen van pakketten wordt veel aandacht besteed aan het vergaren en onderhouden van kennis op diverse gebieden.
Hieronder een greep uit de tot nu toe uitgevoerde onderzoeksopdrachten:

- software-portabiliteit;
- computervirussen (zie artikel "Computervirussen");
- diskette-formaten;
- microprocessoren;
- compilers;
- programmeertheorie (zie artikel "Programmeertheorie");
- padentesters;
- datacommunicatie (zie artikel "Het AppleTalk-netwerk, een beschouwing");
- standaards;
- telecommunicatie;
- besturingssystemen;
- elektronische geld/betaalsystemen (zie artikel "Elektronisch betalen, de magneetkaart");
- object-georiënteerd programmeren (zie artikelen "Objects" en "HyperCard");
- kunstmatige intelligentie.

Een deel van de research-activiteiten heeft betrekking op de microcomputeromgeving. Door de diepgaande kennis over de MS/DOS en de Apple hardware en software, wordt door Software Engineering regelmatig nieuwe apparatuur en programmatuur getest. Ook wordt SE gebruikt als proeftuin voor netwerkfaciliteiten, zoals AppleTalk, Ethernet, modem en file servers, en gateways naar andere systemen. Ze speelt hierbij een belangrijke rol bij de kantoorautomatiseringsactiviteiten van KPMG Klynveld.

Daarnaast wordt de door onderzoeksactiviteiten verkregen kennis aangewend bij het uitvoeren van EDP-audit-opdrachten.
Soms zullen opdrachten worden geleid door een lid van de EDP Audit-sectie, soms ook zullen meer technische opdrachten volledig door SE-leden worden uitgevoerd.
Software Engineering zal bij EDP audits worden ingeschakeld wanneer behoefte bestaat aan technische, praktische kennis en vaardigheden.
Voorbeelden zijn:

- het beoordelen (en eventueel certificeren) van machinecode in beveiligings-IC's (Integrated Circuits);
- als professionele hacker de toegangsbeveiliging van een computernetwerk aan de tand voelen;
- het bouwen of bedienen van technische hulpmiddelen voor het uitvoeren van opdrachten.

Daarnaast worden leden van SE ingezet bij EDP audits wanneer dit past in het opleidings- en carrièreplan van de persoon.

4. Waarom Software Engineering?

De audit-micro wordt door KPMG gezien als strategisch hulpmiddel, dat tot doel heeft de effectiviteit en efficiëntie van de accountantscontrole te verhogen. Om deze reden is besloten om de programmatuur, die specifiek is voor de controle-aanpak van KPMG, in eigen beheer te ontwikkelen. Daartoe zijn twee ontwikkelcentra aangewezen, te weten de SEACAS (System Evaluation Approach - Computerized Audit Support) groep te Montvale (New Jersey, USA) en de sectie Software Engineering te Amsterdam.
Een deel van de door SE uitgevoerde research gebeurt dan ook om het niveau van software-ontwikkeling hoog te houden en indien van toepassing nieuwe(re) technieken en methoden aan te wenden.
Een tweede bestaansreden voor de sectie Software Engineering is te vinden in de snelle technologische ontwikkelingen in het bedrijfsleven. Steeds meer is de cliënt omgeven door en afhankelijk van informatietechnologie, waardoor ook de EDP auditor er steeds vaker mee wordt geconfronteerd. De behoefte aan technisch geschoolden neemt dus ook in dit vakgebied snel toe.

Door middel van deze inleiding en de hierna volgende artikelen hoopt de sectie Software Engineering een indruk te geven van de werkzaamheden die door haar worden verricht.

5. Literatuur

[KERN78] Kernighan, BW, Ritchie, DM: The C Programming Language. New Jersey, Prentice Hall inc.
[MYER78] Myers, GJ: Composite/Structured design.
[SALE83] Salemans, P: Audit-micro, de gebruikers-interface.
[SCHA86] Schalk, J: The KPMG Klynveld programming standard.
[SCHA87] Schalk, J: The KPMG Klynveld C standard.
[SPAP83] Spape, H: Audit-micro, de datacommunicatieverbinding en de keuze van een programmeertaal.
[VAKT86] Micro Special. Vaktualiteiten september 1986.

Software Engineering

door: H. Veenman Ch. Eng.
Ing. L.J.M.W. Gielen

In deze Compact special mag een artikel over software engineering niet ontbreken. Dit artikel verklaart de term software engineering, geeft een kort overzicht van de geschiedenis en belicht een aantal belangrijke aspecten van de discipline.

1 Inleiding

De afgelopen decennia is de aandacht voor de discipline van het ontwikkelen van software sterk toegenomen. Door de enorme groei van computersystemen in aantal en complexiteit èn doordat deze systemen zich steeds vaster wortelen in onze samenleving, neemt ook de behoefte aan systematische benadering van software-ontwikkeling en onderhoud toe.
Software engineering is de leer, die zich bezighoudt met deze activiteiten.

In dit artikel worden in vogelvlucht de verschillende aspecten van software engineering belicht, gelardeerd met een aantal praktische vuistregels. Nadruk hierbij zal liggen op de kwaliteitsbeheersing; hoe een hoge mate van kwaliteit wordt nagestreefd en op welke gebieden verbeteringen kunnen worden aangebracht.

Alvorens inhoudelijk in te gaan op de belangrijkste aspecten van het ontwikkelen van softwareprodukten wordt eerst een historisch overzicht gegeven. Tevens zullen enkele definities de revue passeren en zal de context van het artikel beschreven worden.

1.1 Historie

De behoefte aan een systematische aanpak van software-ontwikkeling en onderhoud stamt uit de jaren zestig. Gedurende die periode kwamen derde-generatie computers beschikbaar, als ondersteuning voor multiprogramming en timesharing-faciliteiten. Vele projecten werden opgestart voor de ontwikkeling van een verscheidenheid aan systemen, waaronder vluchtreserveringssystemen, militaire controlesystemen en procesbesturingssystemen. Slechts enkele daarvan werden opgeleverd, vele werden voortijdig gestopt. Het merendeel van de projecten was onderhevig aan forse overschrijding van de beschikbare budgetten, late oplevering, inefficiency, en gebrek aan betrouwbaarheid en gebruikersacceptatie.

Al snel werd duidelijk dat de behoefte aan betrouwbare software-pakketten sneller groeide dan de mogelijkheid om ze te produceren en te onderhouden.

Eind zestiger jaren werd de term **software engineering** geïntroduceerd als de leer om aan deze chaos een einde te maken.
Onder auspiciën van de NATO werden in 1968 en 1969 twee conferenties aan dit onderwerp gewijd. De term software engineering werd enigszins provocerend op de volgende wijze ingevoerd: het moet toch mogelijk zijn om software te bouwen zoals men huizen en bruggen bouwt, dat wil zeggen uitgaande van een theoretische basis en praktische disciplines, zoals dit gebruikelijk is in de ingenieurswetenschappen.
Sinds die tijd is de complexiteit en het belang van software verder toegenomen, waardoor de noodzaak van ijzeren discipline tijdens het ontwikkelen en onderhouden van software een feit is geworden.

1.2 Definities

Het Institute of Electrical and Electronics Engineers definieert software engineering als volgt [IEEE83]:
Software engineering is the systematic approach to the development, operation, maintenance, and retirement of software.

In deze definitie wordt onder 'software' het volgende verstaan:
Computer programs, procedures, rules, and possibly associated documentation and data pertaining to the operation of a computer system.

Hier wordt slechts gesproken over de benadering van de verschillende fasen in de levenscyclus van een pakket. Fairley gaat in zijn boek over dit onderwerp [FAIR85] iets verder door ook het planningsaspect in de definitie te betrekken:
Software engineering is the technological and managerial discipline concerned with systematic production and maintenance of software products that are developed and modified on time and within cost estimates.

Een meer pragmatische benadering werd indertijd gekozen door Van Vliet [VLIE84] met de volgende omschrijving van het begrip software engineering:
Software engineering is het geheel aan methoden en technieken dat toegepast wordt bij de constructie van grote programma's.

1.3 Context

Voor een beschrijving van de context van dit artikel is de omschrijving van Van Vliet in de voorafgaande paragraaf uitermate geschikt.
Van Vliet doelt met de uitdrukking 'grote programma's' op het criterium, dat meerdere ontwikkelaars, gebruikers en onderhoudsmensen bij het produkt betrokken zijn. Gevolg hiervan is de behoefte aan afspraken, regels en procedures over werkverdeling, wijze van communicatie, verantwoordelijkheden, etc.

Centraal staat ook het beheersen van de complexiteit. Vaak zijn systemen zo complex, dat ze in hun geheel niet meer overzien kunnen worden. Splitsen in onderdelen is dan noodzakelijk, waarbij de communicatie tussen deze onderdelen tot een minimum moet worden beperkt.

Het doel van software engineering-activiteiten is het opleveren van een software-pakket. Er worden echter naast de functionele eisen ook eisen gesteld ten aanzien van de onderhoudbaarheid van het systeem. Dit is van belang omdat een pakket na oplevering aan aanpassing onderhevig is.

Van de vele aspecten, die aan software engineering zijn verbonden, zullen in het kader van dit artikel de volgende in het kort worden behandeld:

- afbakenen van het ontwikkelproces;
- activiteiten in de ontwikkeling;
- planning en begroting;
- kwaliteitsbewaking;
- onderhoud.

Deze aspecten zijn niet geheel los van elkaar te behandelen. Zo is kwaliteitsbewaking sterk afhankelijk van de fasering van het ontwikkelproces.

2 Afbakenen van het ontwikkelproces

In dit hoofdstuk wordt het belang van modularisering van het ontwikkeltraject nader beschreven.

2.1 Noodzaak en doel

Alle ontwikkelmethodieken schrijven een fasering van het ontwikkelproces voor. Door deze fasering kan op gezette tijden duidelijkheid verkregen worden over de beschikbaarheid èn kwaliteit van de (tussen)resultaten.

De meeste programmatuur wordt ontwikkeld door personen die de produkten niet zullen gebruiken. De ontwikkelaars ontvangen een opdracht die onder impliciete controle van de opdrachtgever(s) uitgevoerd wordt. Het feit dat de produkten door anderen dan de ontwikkelaars gebruikt (en eventueel door wéér anderen onderhouden) zullen worden alsmede het feit dat meerdere personen bij de ontwikkeling betrokken zijn, maakt de communicatie onontbeerlijk.
Logisch volgt hieruit de behoefte aan een meer systematische aanpak.
Om een software-pakket te kunnen ontwikkelen, dienen de eisen en wensen van toekomstige gebruikers expliciet te worden vastgelegd. Ontwikkelaars, gebruikers en onderhoudsmensen moeten het eens worden over het ontwerp van het produkt. Programmacode dient zorgvuldig geschreven en terdege getest te worden. Ondersteunende documentatie moet worden ontwikkeld zoals bedieningsinstructies, gebruikershandleidingen, opleidingsbescheiden en onderhoudsdocumentatie.

Een dergelijk complex van eisen, wensen en doelstellingen kan beter worden beheerst wanneer op van tevoren gedefinieerde meetpunten afstemming plaatsvindt. Hiertoe wordt een ontwikkeltraject in fasen opgedeeld.

Fasering van het ontwikkeltraject heeft tot doel de doelstellingen, die bij de start zijn gesteld, bewuster na te streven. Daartoe zal iedere fase worden afgerond met een bepaald produkt, vaak bestaande uit een rapport of een document. Aan de hand van dit produkt kan worden bepaald of ook na uitvoering van de zojuist beëindigde fase dezelfde doelstellingen worden nagestreefd en of de activiteiten volgens planning en begroting zijn verlopen. Afhankelijk van deze terugkoppeling zal bijsturing plaatsvinden van het geplande vervolgtraject.

2.2 Richtlijnen

Bij een goede fasering worden in een zo vroeg mogelijk stadium de volgende vragen beantwoord:

- welke eisen worden gesteld aan het te ontwikkelen produkt?
- wanneer moeten welke (deel)resultaten beschikbaar zijn en gecontroleerd worden?
- wie zijn verantwoordelijk voor de ontwikkeling en de kwaliteitsbewaking?
- hoe is de overgang naar een volgende fase geregeld?

Daarnaast moet voor een goede opzet van het ontwikkelproces vooraf worden bepaald welke methoden en technieken gehanteerd zullen worden en aan welke standaarden de produkten moeten voldoen.

2.3 Voordelen

In de voorafgaande paragrafen is reeds een belangrijk voordeel van een goede fasering van het ontwikkelproces aangegeven. Een betere beheersing van de ontwikkelactiviteiten is mogelijk omdat op van tevoren bepaalde meetpunten, duidelijkheid wordt verkregen over de beschikbaarheid èn kwaliteit van de (tussen)resultaten en omdat een afstemming met de gebruikers en/of opdrachtgevers kan plaatsvinden.
De activiteiten worden opgedeeld in een aantal tijdvakken. Dit verhoogt de mogelijkheden voor een goede planning.
Daarnaast levert de fasering een impliciete partitionering van het probleem. Hierop zal in de volgende paragraaf nader worden ingegaan.

2.4 Wat versus hoe

De fasering van een ontwikkelproces dient hoofdzakelijk om een moeilijk karwei hanteerbaar te maken. De resultaten van een eerste fase

zijn globaal/abstract; de resultaten van iedere volgende fase worden steeds meer precies. Getracht wordt om alle produkten van een bepaalde fase van een zelfde abstractiegraad te krijgen. Hierbij treedt een probleem op wat wel aangeduid wordt met het "wat versus hoe" dilemma.
Alle produkten van een fase moeten de vraag "wat?" beantwoorden en dienen als het startpunt voor het beantwoorden van de vraag "hoe?" van een volgende fase.

Zo geven de gebruikerswensen aan **wat** er moet komen. De daaruit voortvloeiende produktbeschrijving geeft een antwoord op de vraag **hoe** aan deze gebruikerswensen kan worden voldaan en **wat** de functionele specificaties moeten verduidelijken. De functionele specificaties geven weer antwoord op de vraag **hoe** aan de produktomschrijving kan worden voldaan.

Deze afwisseling van de vragen "wat?" en "hoe?" gaat door tot tijdens het testen een antwoord wordt verkregen op de vraag **hoe** het uiteindelijke produkt **(wat)** werkt.
Dit dilemma levert in de praktijk grote problemen op. Veel ontwerpers kunnen moeilijk inschatten wat in een globaal ontwerp beschreven moet worden en wat in een gedetailleerd ontwerp. Een vuistregel zoals *het globaal ontwerp beschrijft wat er gerealiseerd moet worden en het gedetailleerd ontwerp geeft aan hoe dit moet gebeuren* is te simplistisch.
Ook maakt dit dilemma het inschatten van de diepgang van een tussenprodukt moeilijk.

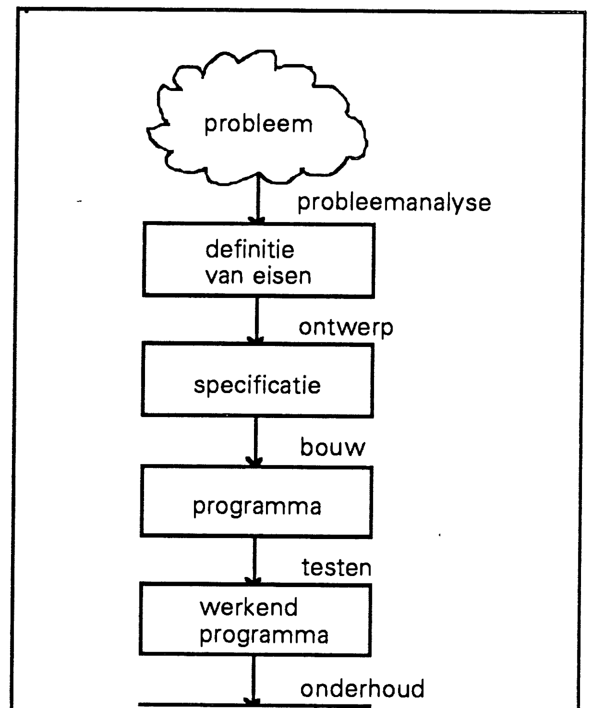
2.5 Mogelijke faseringen
Zoals reeds aangeduid zijn verschillende modellen voor de fasering van een ontwikkelproces mogelijk. Een bekend model is het 'watervalmodel', waarin de kwaliteitsbewaking voor elke fase expliciet naar voren komt. In dit model zien we de interim-produkten van een afgesloten fase soepel overlopen naar de volgende fase [FAIR85].
Andere modellen besteden expliciet aandacht aan prototyping-activiteiten.

Grofweg kan in het traject van software-ontwikkeling de volgende fasering worden onderkend: probleemanalyse, ontwerp, bouw, testen, evaluatie en onderhoud. (Zie figuur 1.)

De meeste ontwikkelmethoden, die in de praktijk worden gebruikt, houden een soortgelijke fasering aan, zij het in meer of minder gedetailleerde vorm, of gebruik makend van andere benamingen voor de verschillende deeltrajecten.

Na afsluiting van elke fase wordt in overleg met de gebruiker aan de hand van de behaalde resultaten van die fase het vervolgtraject bepaald. Dit kan uitvoering zijn van de volgende fase (na goedkeuring van de resultaten door de gebruiker), (gedeeltelijke) heruitvoering van de afgesloten fase of beëindiging van het project (beide na afkeuring van de resultaten door de gebruiker).

2.6 Documentatie
Om de resultaten van een fase vast te leggen en om de voortgang en de kwaliteit te kunnen meten, worden gedurende het gehele ontwikkeltraject documenten geproduceerd. Voorbeelden hiervan zijn definitiestudie, functioneel ontwerp, technisch ontwerp, systeem- en programmadocumentatie, testverslagen, gebruikershandleiding, etc. Te zamen vormen zij de bij het systeem behorende systeemdocumentatie. Aangevuld met voortgangs- en planningsrapporten vormen zij de projectdocumentatie. Documentatie is derhalve niet als aparte fase genoemd; het is een integrale activiteit die gedurende het gehele ontwikkeltraject aandacht verdient.
Op zich kan iedere fase weer worden opgedeeld in een reeks van activiteiten, die na elkaar of parallel moeten worden uitgevoerd.



Figuur 1. Fasering van het software engineering-traject

2.7 Nadelen en gevaren
Een ontwikkelmethode, waarin een bepaalde mate van fasering is onderkend, heeft naast de reeds genoemde voordelen ook een aantal nadelen. Maar al te vaak worden faseringen gebruikt als blindengeleidehond. Zonder verder inzicht worden de voorgestelde fasen stuk voor stuk doorlopen, terwijl men zich niet afvraagt of de gekozen methode wel de juiste is.

Een ontwikkelmethode dient slechts als referentiekader te worden gebruikt, als checklist om na te gaan of tijdens de ontwikkeling niets over het hoofd wordt gezien. Het gevaar van een dergelijk 'inslapen' neemt toe naarmate de detaillering van een fasering in voorgekauwde stappen groter is.

3 Activiteiten in de ontwikkeling

Vasthoudend aan de in paragraaf 2.5 gegeven fasering, volgt hieronder een korte omschrijving van de basisactiviteiten van een ontwikkelproces.

3.1 Probleemanalyse

Doel van de probleemanalyse is te komen tot een zo volledig mogelijke beschrijving van het op te lossen probleem. Opdracht tot uitvoering van deze fase wordt door de (toekomstige) gebruiker verleend. In de beschrijving van het probleem zullen de volgende elementen zijn opgenomen:

- de gewenste functionaliteit;
- de gewenste flexibiliteit (aanpassingen, uitbreidingen);
- de gewenste documentatie;
- de gewenste prestaties.

Aan de hand hiervan zal een haalbaarheidsonderzoek worden uitgevoerd om te bepalen of economisch en technisch tot een oplossing kan worden gekomen.
Tijdens deze fase dienen, in overleg met de gebruiker, ook de eisen te worden geformuleerd, die aan de omgeving worden gesteld. Als eindprodukt van deze fase worden de functionele specificaties opgeleverd.

Deze functionele specificaties dienen zo nauwkeurig mogelijk te omschrijven aan welke eisen het op te leveren systeem moet voldoen.
Op basis van de functionele specificaties worden planning en budget opgesteld of bijgesteld. Naarmate het project vordert kunnen deze schattingen worden bijgesteld.

3.2 Ontwerp

Aansluitend op de beschrijving van het op te lossen probleem wordt in deze fase een beschrijvend model gemaakt van de oplossing ('het systeem').
Genoodzaakt door de complexiteit zal het geheel dikwijls in hanteerbare delen worden opgesplitst. Het is van groot belang om de functie(s) van deze 'modules' zo nauwkeurig mogelijk te beschrijven, evenals de verbindingen tussen de modules. Vaak wordt aan deze fase te weinig aandacht besteed, omdat de volgende fase, het bouwen, de voorkeur heeft van ontwikkelaars. Deze houding heeft echter een zeer negatieve invloed op de kwaliteit van het uiteindelijke pakket. Eindprodukt van de ontwerpfase is een volledige beschrijving van het te implementeren systeem: het ontwerp.
De functionele specificaties en het ontwerp worden in de praktijk vaak gehanteerd als contract tussen de opdrachtgever en de software engineer. Een eenduidige formulering van deze documenten is dus van groot belang.

3.3 Bouw

Uitgaande van het ontwerp worden in deze fase de verschillende modulen gebouwd, geprogrammeerd. Door een strakke modulaire aanpak is het nu ook mogelijk om verschillende modules door verschillende software engineers te laten schrijven. Van belang is op te merken dat het doel van het bouwen is een goed gedocumenteerd, overzichtelijk, betrouwbaar, leesbaar, flexibel programma op te leveren. Slechts bij hoge uitzondering mogen programmeringstrucs worden toegelaten, die de efficiëntie zouden verhogen.

3.4 Testen

Na de bouw wordt iedere module getest. Een module wordt veelal getest door de ontwikkelaar zelf. Nagegaan wordt of de module voldoet aan de in de specificaties gestelde eisen ten aanzien van functionaliteit, representatie, snelheid, e.d.
Vervolgens zullen de modules worden samengesmolten tot het uiteindelijke systeem. Tijdens deze integratie worden continu 'integratietests' uitgevoerd. Nadruk bij dit testen ligt onder meer op de werking van de verbindingen tussen de verschillende modulen. Na voltooiing van de integratie zal het systeem vervolgens de 'systeemtest' en 'acceptatietest' ondergaan. Hierbij wordt de werking van het gehele systeem getest, telkens weer afgezet tegen de eisen, zoals die in de specificatie zijn vastgelegd.
De systeemtest wordt een 'white box test' genoemd, omdat hierbij gebruik gemaakt wordt van de inhoudelijke kennis van het systeem. De acceptatietest daarentegen geldt als 'black box test', omdat de tester (namens de gebruiker) uitgaat van de functionele specificaties van het systeem.
Voor alle uit te voeren tests dient tevoren een testplan te zijn opgesteld. De resultaten van een test worden vastgelegd in een testrapport.

3.5 Evaluatie

Evaluatie van het project, hier als aparte fase onderscheiden, is een vaak vergeten deeltraject. Het belang ervan wordt schromelijk onderschat, maar juist door een zorgvuldige evaluatie is het mogelijk om lering te trekken uit de gang van zaken. Zaken, die deze keer niet naar wens verliepen, kunnen worden verbeterd. Activiteiten, die geheel naar wens verliepen kunnen in een volgend project op soortgelijke wijze worden aangepakt.
Met de afsluiting van deze fase wordt een vervolgtraject opgestart: het onderhoudstraject.

3.6 Onderhoud
Als eindprodukt van de testfase wordt het software-pakket opgeleverd, voorzien van alle benodigde documentatie. Vanaf dat moment is het eigenlijke ontwikkeltraject afgelopen. In vervolg hierop vindt echter nog training en opleiding van de aspirant-gebruikers plaats, waardoor de gebruikers in staat worden gesteld hun produkt ook daadwerkelijk te gaan gebruiken.
Zodra de oplevering van het software-pakket heeft plaatsgevonden worden meestal nog fouten gevonden. Dit wordt veroorzaakt door het feit dat nu anderen het produkt gaan gebruiken, maar is tevens inherent aan het feit dat testen nimmer zal leiden tot een uitspraak als: het is 100% zeker dat dit programma geen fouten bevat. Sterker nog, er kan bij grote programma's *met een gerust hart* van worden uitgegaan dat er nog fouten in zijn achtergebleven. De kans is groot dat deze fouten slechts na lange tijd en bij toeval worden ontdekt, omdat ze verstopt zitten in delen van het programma die maar zelden worden gebruikt, of slechts onder zeer bijzondere omstandigheden worden veroorzaakt.

Onder onderhoud van de pakketten valt onder meer:
- het analyseren van wijzigingsaanvragen;
- herontwerp en modificatie van programmacode;
- testen van gemodificeerde code;
- bijwerken van documenten en documentatie;
- distributie naar de verschillende gebruikers.

In een volgend hoofdstuk zal dieper worden ingegaan op het onderhoudstraject.

3.7 De gebruiker
Als uiteindelijke opdrachtgever is de gebruiker een uiterst belangrijke schakel in het ontwikkelproces. De gebruiker beslist of een fase mag worden opgestart; bij afsluiting van de verschillende fasen wordt telkens door de gebruiker afgestemd of afspraken ten aanzien van functionaliteit, planning e.d. zijn nagekomen. De projectorganisatie moet derhalve zodanig zijn, dat de gebruiker in staat is de planningen en resultaten van iedere fase te beoordelen.

4 Planning en begroting
Parallel aan de beschreven fasen dient een planning en begroting van het project te worden opgesteld en bijgehouden. Dit alles is vervat in het projectplan.
Hierin worden de volgende aspecten belicht:
- een beschrijving van het levenscyclusmodel van het te ontwikkelen produkt; in dit model wordt de te hanteren terminologie vastgelegd, worden mijlpalen in het ontwikkeltraject bepaald en wordt de op te leveren projectdocumentatie beschreven;
- de organisatorische structuur;
- voorlopige schattingen van benodigde bemanning en middelen;
- een voorlopige kostenraming;
- een beschrijving van de te hanteren projectbeheersingsmethoden en -technieken (zoals voor ontwerp en programmering).

Bij de kostenraming zijn twee fundamenteel verschillende benaderingen te onderscheiden, te weten de top-down- en de bottom-up-benadering. Bij de eerste aanpak wordt primair gekeken naar de kosten, die nodig zijn voor de uitvoering van de verschillende functionele activiteiten van het project. Bij de bottom-up-benadering daarentegen wordt bij de kostenschatting met name gekeken naar de ontwikkeling van elk van de te onderscheiden software-componenten. In de praktijk zal vaak een mix van deze benaderingen worden gehanteerd.
Er zijn verschillende technieken en modellen ontwikkeld, die als ondersteuning kunnen dienen bij deze activiteit.
Gedurende het project zal de planning voortdurend kritisch worden bekeken. Indien nodig zal ze na iedere fase worden bijgesteld.

5 Kwaliteitsbewaking
Gedurende het gehele ontwikkeltraject (en veelal ook tijdens het gebruik van het produkt) dient de kwaliteit van het uiteindelijke produkt te worden gewaarborgd door verificatie en evaluatie van de tussenliggende resultaten. Telkens zullen de opgeleverde documenten, zoals specificaties, moeten worden afgezet tegen de tevoren uitgesproken verwachtingen. Slechts door stringente controle op de kwaliteit van de tussenresultaten kan een kwalitatief goed produkt worden verkregen. De rapportage inzake de kwaliteitsbewaking moet zodanig zijn dat deze door de gebruiker kan worden beoordeeld.

Testen leidt per definitie niet tot foutloze pakketten. Met behulp van testen kan slechts de kans dat een fout in een systeem achterblijft, worden verkleind.
Als regel wordt in dit verband vaak gehanteerd, dat het aantal fouten, dat in een pakket onontdekt achterblijft, in dezelfde orde van grootte ligt als het aantal fouten dat tijdens het testen is gevonden.

In dit hoofdstuk zullen achtereenvolgens beschreven worden:
- het belang van kwaliteitsbewaking voor de kwaliteit van een goed gedefinieerd ontwikkelproces;
- enkele technieken die toegepast kunnen worden;
- het plannen van kwaliteitsbewaking.

Ook zal nader worden ingegaan op de bedreigingen ten aanzien van kwaliteit en op het belang van goede procedures voor het geval dat gevonden fouten hersteld moeten worden.

5.1 Afbakenen van het ontwikkelproces
Teneinde een goede kwaliteitszorg mogelijk te maken, moet op gezette tijden duidelijkheid verkregen worden over de kwaliteit van de (tussen)resultaten. Periodiek moeten resultaten beschikbaar zijn voor controle en moeten personen aangewezen worden om die controle uit te voeren. Dit op zijn beurt impliceert het vooraf plannen van de kwaliteitszorg.

Om kwaliteitscontrole mogelijk te maken, is het noodzakelijk dat criteria worden gedefinieerd waaraan de in elke fase geproduceerde produkten kunnen worden getoetst.
De doorlooptijd en de hoeveelheid te verrichten werk van een fase kan van dien aard zijn, dat het controleren van de kwaliteit bij het beëindigen van een fase niet acceptabel is. Daarom is het veelal nuttig een fase op te splitsen in sub-fasen en de resultaten van elke sub-fase te controleren. Door deze opsplitsing van het proces in een aantal sequentieel uit te voeren deelprocessen, wordt bewerkstelligd dat fouten zo vroeg mogelijk in het project naar voren komen en de kosten van reparatie minimaal blijven.

Samengevat is voor een goede bewaking van de kwaliteit vereist dat:
- het proces wordt opgedeeld in kleinere, controleerbare, deelprocessen (hierbij oppassen voor het gevaar van 'inslapen' (zie 2.7)) en dat
- tijdens en bij beëindiging van elk deelproces kwaliteitsbewaking wordt uitgevoerd.

Ten aanzien van kwaliteitszorg kunnen voor elk deelproces vervolgens onder meer de volgende afgeleide eisen worden geformuleerd:
- wat de entry- en exit-criteria zijn;
- welke technieken gebruikt moeten worden voor eindmeting van exit-criteria;
- welke technieken gebruikt moeten worden voor in-process meting van de exit- criteria en
- hoe de resultaten van de metingen gebruikt zullen worden.

Daarnaast is het sterk aan te bevelen een continue bewaking van de kwaliteit in te voeren. Dit kan door de betrokkenen te wijzen op de verwachte kwaliteit en ze aan te moedigen om bijvoorbeeld walk-throughs toe te passen [KLUY88].

5.2 Technieken voor kwaliteitsbewaking
Drie technieken voor het bewaken van de kwaliteit, die op de resultaten/produkten van vrijwel elke (sub)fase kunnen worden toegepast, zijn:
- Inspectie.
  Dit is een formele evaluatietechniek, waarin tijdens een vergadersessie alle delen van een (deel)produkt in detail worden onderzocht of gemeten door een groep van personen.
- Walk-through.
  Dit is een meer informeel overleg tussen twee of meer ontwikkelaars, waarbij een deel van de werkzaamheden van één van de deelnemers wordt doorgenomen. De walk-through heeft in vergelijking met de inspectie een meer continu karakter.
- Testen.
  Het testen kan omschreven worden als het gebruiken van de produkten teneinde fouten of onvolkomenheden op te sporen. Alle produkten van een ontwikkeling komen in aanmerking voor testen.

5.3 Planning
Het inschakelen van personeel en materiaal ten behoeve van kwaliteitszorg maakt een goede planning noodzakelijk. Zo vroeg mogelijk moet deze planning opgesteld worden. Er moet terdege rekening mee gehouden worden dat deze planning synchroon moet lopen met de ontwikkelplanning.
Ten behoeve van de planning zijn de volgende activiteiten noodzakelijk:
- bepalen van de uit te voeren activiteiten;
- bepalen van tussentijdse activiteiten (bijvoorbeeld het detailleren van een testplan);
- selecteren van uitvoerend personeel;
- definiëren van benodigd materiaal teneinde activiteiten te kunnen uitvoeren en het zo nodig bijstellen van de planning;
- het implementeren van activiteiten in de ontwikkelplanning;
- het plannen van bestelling en/of bouw van benodigd materiaal;
- overleg met de ontwikkelgroep.

Indien de organisatie beschikt over een specifiek orgaan voor de bewaking van de kwaliteit, zoals een EDP Audit-afdeling of zelfs een afdeling Kwaliteitsbewaking, kan hiervan gebruik worden gemaakt.
Overleg met betrokken medewerkers met betrekking tot de planning is essentieel. De produktie van de Ontwikkelafdeling moet niet of zeer weinig vertragen als gevolg van het uitvoeren van kwaliteitsbewakingsactiviteiten.

Eén van de beslissingen die genomen moet worden bij het plannen van de activiteiten ten behoeve van kwaliteitsbewaking is, het opstellen van stopcriteria. Deze criteria moeten in samenwerking met de gebruiker worden opgesteld. Voor elke activiteit moeten de stopcriteria expliciet worden vermeld. Veelal zullen deze criteria een goede planning bemoeilijken.

5.4 Bedreigingen
De kwaliteit van een te produceren softwarepakket wordt negatief beïnvloed door een groot aantal factoren. Om effectief de kwaliteit te kunnen bewaken moeten deze bedreigingen onderkend worden en passende maatregelen genomen worden. In deze paragraaf zal een aantal bedreigingen aangestipt worden.

Wellicht de bedreiging met de grootste invloed is het gebrek aan kennis en ervaring van de ontwikkelaars in verhouding tot de voor het project noodzakelijke niveau. Hierbij moet niet alleen gedacht worden aan de algemene vakkennis van de betrokkenen, maar ook aan specifieke ervaringen met soortgelijke projecten/applicaties. Veelal zal het ontbreken van een goed opleidingsplan hieraan debet zijn.

Een andere belangrijke bedreiging, het ontbreken van een goede systematische aanpak, is in het voorafgaande al aan de orde gekomen. Een systematische aanpak bevordert en stuurt de communicatie tussen ontwikkelaars. Het belang hiervan wordt onderstreept door het feit dat ongeveer 40% van de ontwikkeltijd besteed wordt aan communicatie.
Een goede communicatie met de gebruikers is eveneens een vereiste. Hierdoor wordt vermeden dat complexe gebruikerswensen slecht gedefinieerd worden. Te vaak bestaat bij ontwikkelaars onvoldoende begrip voor het op te lossen probleem.

Het ontbreken van een goede planning kan als een volgende bedreiging gezien worden. Soms wordt de beschikbare tijd bepaald door externe factoren (bijvoorbeeld wettelijke regelingen). Hiermee gerelateerd is de bedreiging dat gebruikers of opdrachtgevers maar al te vaak snel resultaten willen zien. Een te optimistische planning is het gevolg. Het gezegde *haastige spoed is zelden goed* hebben veel ontwikkelaars aan den lijve ondervonden. Pogingen om in onvoldoende tijd een produkt op te leveren zullen ten koste gaan van de kwaliteit.

Wellicht meer nog dan het eerder genoemde niveau van kennis en ervaring is een goede motivatie noodzakelijk. Onvoldoende motivatie en/of verantwoordelijkheidsgevoel voor de voortgang en de kwaliteit bij de uitvoerende personen, zal al te vaak resulteren in het te laat of niet ontdekken van fouten.

Het is vrijwel niet mogelijk om in een specifieke ontwikkelomgeving alle bedreigingen te elimineren. Wel kan getracht worden de bedreigingen met de grootste invloed te elimineren dan wel grotendeels te ontkrachten.

Een aantal van de bedreigingen kan aan het begin van het ontwikkelproces worden onderschat. Door het bijhouden van een goede documentatie van tijdens het proces geconstateerde fouten, kan worden vastgesteld of een deel van deze fouten chronisch van aard is. Passende acties moeten dan ondernomen worden.

5.5 Correctieprocedures
Met het vinden van onvolkomenheden en fouten is de 'taak' van kwaliteitsbewaking niet ten einde. Ook moet erop worden toegezien dat deze fouten volgens een correcte procedure worden hersteld, omdat anders het gevaar gelopen wordt dat fouten op verkeerde tijdstippen en/of in verkeerde versies van de programmatuur worden hersteld.

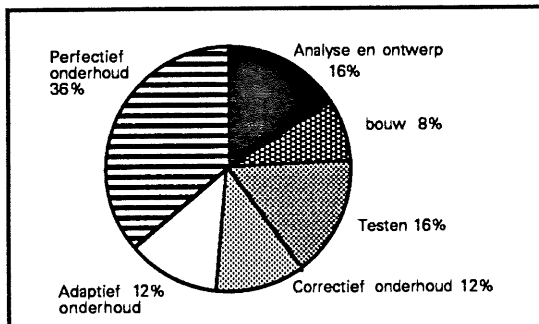
Bij het corrigeren van een fout bestaat het gevaar van **bad fix injection**. Bad fix injection is de introductie van een nieuwe fout bij het herstellen van een bestaande fout. Om inzicht te verkrijgen betreffende zowel detection efficiency als repair efficiency, is geschreven rapportage een vereiste.

6 Onderhoud
Zoals in paragraaf 3.6 is gesteld begint direct na de oplevering van het produkt het traject van onderhoud, dat aanhoudt totdat het pakket buiten gebruik wordt gesteld. Deze fase wordt veelal negatief beschouwd. Redenen hiervoor zijn:
- weinig creativiteit wordt verondersteld, omdat het een aanpassing aan een bestaand produkt betreft;
- de angst die bij de programmeur leeft voor een zgn. bad fix injection;
- de produktiviteit is moeilijk te meten;
- de noodzaak aan specialistische kennis wordt vaak onderschat.

Dit terwijl het belang van onderhoud in de literatuur regelmatig wordt onderstreept.
Ter illustratie: een software-pakket is gemiddeld 1 tot 3 jaar in ontwikkeling, terwijl een dergelijk pakket gemiddeld 15 jaar in gebruik is en dus onderhouden moet worden. Uitgedrukt in inspanning kan worden gesteld dat de verhouding tussen ontwikkeling en onderhoud varieert van 40/60 tot 10/90 (!).
In de literatuur (b.v. [SWAN76]) worden vaak drie soorten onderhoud onderscheiden:
- *correctief onderhoud* - het repareren van bugs;
- *adaptief onderhoud* - het aanpassen van het software-pakket aan in de omgeving opgetreden veranderingen;
- *perfectief onderhoud* - het aanpassen van het pakket aan de wensen van de gebruiker.

Correctief onderhoud zal in alle gevallen als daadwerkelijk onderhoud kunnen worden opgevat. Bij de overige twee categorieën zal echter door de mate van aanpassing worden bepaald of het nog als onderhoud kan worden gezien, of dat het produkt zodanig zal moeten worden aangepast of uitgebreid, dat feitelijk van

een nieuw produkt - dus een nieuw project - sprake is. Dit wordt ook wel 'vernieuw bouw' genoemd.
De verdeling van inspanning over deze categorieën is ongeveer 20 : 20 : 60. Als we daarnaast de inspanningsverhoudingen in ogenschouw nemen van de verschillende stadia van het ontwikkeltraject, te weten 40 procent voor analyse en ontwerp, 20 procent voor bouwen en 40 procent voor integratie en acceptatietest, komen we tot een totale distributie van inspanning over de levenscyclus van een pakket. In figuur 2. is deze distributie visueel weergegeven, uitgaande van een verhouding in inspanning tussen ontwikkeling en onderhoud van 40 : 60.



Figuur 2 : Inspanningsverhoudingen.

7. Theorie versus praktijk

In de voorafgaande hoofdstukken is een aantal aspecten van software engineering aan de orde gekomen. Deze aspecten zijn vanuit een meer theoretisch oogpunt beschreven. Hoe zit het nu eigenlijk met de praktijk? Is het waar dat het maken van software vergelijkbaar is met het bouwen van een brug?

Om een antwoord te geven op deze vraag moet eerst een aantal resultaten van studies van Belady en Lehman [BELA79][LEHM80] nader bekeken worden. Ze komen op basis van hun studies naar de ontwikkeling van grote pakketten tot vijf wetten voor programma-evolutie. De twee belangrijkste wetten zijn:

- De wet van de continue verandering. Een programma wordt continu aangepast of het wordt steeds minder bruikbaar. Dit proces zet zich voort tot het qua kosten effectief wordt het programma te vervangen door een opnieuw ontwikkelde versie.
- Toenemende complexiteit.
  De complexiteit van een programma, neemt toe wanneer het wordt aangepast, tenzij actief wordt gewerkt aan het reduceren van deze complexiteit.

Daarnaast noemt Van Vliet [VLIE84] een in dit kader belangrijke eigenschap van software: software is niet continu.

Beide wetten en de bovengenoemde eigenschap geven een aantal belangrijke aanknopingspunten. Zo zal aan een programma voortdurend functioneel gesleuteld worden. Een programma is nooit 'af'. Tijdens het 'onderhoud' bestaat het gevaar dat een kleine aanpassing in het programma zeer grote gevolgen heeft (het programma voert bepaalde taken anders of in het geheel niet uit).
Het verschil tussen software engineering en andere disciplines wordt daarnaast ook vaak verklaard uit het verschil in leeftijd. Software engineering wordt pas gedurende enkele decennia bedreven, terwijl de bouwkunde al eeuwen bestaat.

Tot slot is er het feit dat een stuk software niet 'zichtbaar' is. Software heeft géén fysieke eigenschappen; de executie van software kan bestudeerd en geëvalueerd worden, de software zelf niet. Omdat software niet 'zichtbaar' is, is voortgang in een ontwikkeltraject moeilijk vast te stellen. Door middel van rapporten en tests moet bepaald worden in hoeverre het gewenste produkt aanwezig is.

8 Toekomstverwachtingen

Eind jaren zestig werd de noodzaak van een gefaseerde en gecontroleerde ontwikkeling onderkend. Sindsdien heeft een evolutie plaatsgevonden ten aanzien van ontwikkelmethoden en -technieken. Tot nu toe heeft deze evolutie geleid tot betere produkten en beter beheersbare projecten, maar tevens tot een verdere detaillering van taken.
Nu zien we een evolutie starten in tegenovergestelde richting. Nieuwe programmeertechnieken zoals object-georiënteerd programmeren, laten reeds een integratie van ontwerp en bouw zien. Voordeel hiervan is een directere betrokkenheid van de gebruiker bij de ontwikkeling.
Door het beschikbaar komen van meer geavanceerde hulpmiddelen ter ondersteuning van het specificatietraject, zoals PSL/PSA van het ISDOS project [VLIE84], wordt de benodigde inspanning voor het werkelijke bouwtraject steeds verder teruggebracht.
Deze ontwikkeling zal zich kunnen voortzetten totdat slechts een gedetailleerde specificatie nodig is voor het ontwikkelen van een oplossing. De leer van software engineering zal zich telkens aanpassen aan deze en toekomstige ontwikkelingen, zodat de kwaliteit van de uiteindelijke toepassingen, zijnde de mate van aansluiting op de eisen van de gebruiker, zich verder in positieve zin zal ontwikkelen.

9 Personalia

Hans Veenman is sedert 1978 in dienst bij KPMG Klynveld EDP Audit. Hij heeft sinds de oprichting

in 1983 leiding gegeven aan de sectie Software Engineering. Zijn vooropleiding ligt op het gebied van de elektrotechniek. Daarnaast heeft hij de AMBI-opleiding voltooid en is chartered engineer in the field of Telecommunications.
Sinds begin 1988 heeft hij zijn werkterrein verlegd naar de telematica audit, de nieuwe discipline die zich richt op de beheersbaarheid van telecommunicatiesystemen.

Laurent Gielen is sinds juli 1985 als Software Engineer werkzaam bij KPMG Klynveld EDP Audit. Hij is in 1985 afgestudeerd aan de HIO te Eindhoven. Zijn interessegebieden zijn User Interfaces, programmeertalen en -technieken en artificial intelligence.

10 Literatuurreferenties

[BELA79] Belady, L, Lehman, M: The characteristics of large systems. Research Directions in Software Technology, MIT Press, Cambridge, Mass., 1979.

[FAIR85] Fairley, R: Software engineering concepts. McGraw Hill, 1985.

[IEEE83] IEEE Standard Glossary of Software Engineering Terminology, IEEE Standaard 729-1983.

[KLUY88] Kluyt, O: Het testen van software. Elders in deze Compact.

[LEHM80] Lehman, M: On Understanding Laws, Evolution, and Conservation in the Large-Program Life Cycle. The Journal of Systems and Software, vol. 1, no. 3, 1980.

[MYER78] Myers, GJ: Composite/structured design.

[SCHA86] Schalk, J: The KPMG Klynveld programming standard. Versie April 1988.

[SWAN76] Swanson, EB: The dimensions of maintenance, Proceedings 2nd International Conference on Software Engineering, IEEE (1976) pp 492-497.

[VLIE84] Vliet, JC van: Software engineering. Stenfert Kroese, 1984.

Het testen van software

door: O. Kluyt

1 Inleiding

Dit artikel wil een overzicht geven van een aantal strategieën en methodieken, die kunnen worden gehanteerd bij het testen van software. Het testen dient een integraal onderdeel te zijn van het ontwerp en de ontwikkeling van een systeem.
In de ontwikkeling van een systeem kan, onafhankelijk van de gevolgde methode, een aantal fasen worden onderscheiden. Tijdens elke fase dient getest te worden of het systeem de gewenste functionaliteit bevat en of het systeem aan de overige eisen voldoet.
In dit artikel zal de nadruk liggen op het testen van de implementatie; het testen van de specificaties en het conceptueel ontwerp zullen slechts kort worden behandeld.
Tijdens de bouw van een systeem zijn de volgende stappen te onderkennen waarin het testen plaatsvindt: de functietest, moduletest, integratietest, systeemtest en de acceptatietest.
Daarnaast zal in het kort worden ingegaan op de problematiek ten aanzien van het testen van modificaties.
Het programma of systeem dat wordt getest, wordt aangeduid als probleemprogramma (PP).

Het testen van software dient twee doelen: voldoet het systeem aan de specificaties en is de ergonomie van het systeem juist in relatie tot de toekomstige gebruikers. Bij het eerste doel moet niet alleen worden vastgesteld of de geboden functionaliteit volledig en juist is, maar tevens of het systeem niet méér doet.
Het testen kan, uitgaande van deze twee doelen, ook gebruikt worden ingeval van prototyping (is dit wat en hoe de gebruiker het wil).

In dit artikel zal eerst een vergelijking worden gemaakt tussen handmatig (interactief) testen en automatisch testen. Hierbij kan gebruik gemaakt worden van een aantal hulpmiddelen. Van elk zal kort het doel en de toepassingsmogelijkheden worden behandeld.
Vervolgens zullen de verschillende fasen, waarin het testen van het PP valt onder te verdelen, worden behandeld alsmede een vergelijking tussen incrementeel en niet-incrementeel testen.
Ten slotte zal enige aandacht worden besteed aan de ergonomische aspecten van een systeem.

2 Handmatig versus automatisch testen

Een test kan handmatig of automatisch worden uitgevoerd. Dat wil zeggen, de tester kan gebruik maken van geautomatiseerde hulpmiddelen die de test uitvoeren, of zelf het PP interactief testen.
Voor beide geldt een aantal voor- en nadelen.

Voordelen van handmatig testen:
- inspiratie van het moment;
- kan door gebruiker worden uitgevoerd;
- geen speciale automatiseringskennis vereist;
- niet tijd-, plaats- of omgeving-gebonden.

Nadelen van handmatig testen:
- slechte herhaalbaarheid;
- afhankelijk van kennis en ervaring van de tester;
- slecht beheersbaar;
- traag.

Voordelen automatisch testen:
- goede herhaalbaarheid;
- beheersbaar;
- onafhankelijk van kennis en ervaring van de tester;
- volledigheid.

Nadelen automatisch testen:
- vereist veel voorbereiding;
- speciale automatiseringskennis vereist;
- moeilijke realiseerbaarheid;
- onderhoud van de test;
- de test kan het PP beïnvloeden.

Onder herhaalbaarheid wordt verstaan, de mogelijkheid tot heruitvoering van een test. Naspeelbaarheid houdt in, het kunnen herhalen van bepaalde resultaten van een test voor die versie van het PP.

Hoewel automatisch testen meer zekerheid geeft omtrent de volledigheid en beheersbaarheid van de test, zal in de praktijk zelden uitsluitend automatisch worden getest. Veelal zal er voor een combinatie van handmatig en automatisch testen worden gekozen.

3 Hulpmiddelen bij testen

In de voorgaande paragraaf werd gesproken over geautomatiseerde hulpmiddelen. Het doel van deze hulpmiddelen is de tester in staat te stellen de naspeelbaarheid en herhaalbaarheid van de tests te vergroten en zo de kwaliteit van de tests (direct) als de kwaliteit van het PP (indirect) te verbeteren.
Mogelijke hulpmiddelen zijn I/O-capture, testdatageneratoren, padentesters en call charts.

3.1 I/O-capture

I/O-capture-hulpmiddelen vangen de input (zoals toetsaanslagen) van de tester voor het PP en de output van het PP af en slaan deze op in een file. Deze informatie kan, al dan niet geformatteerd door het hulpmiddel, worden afgedrukt voor bestudering en rapportering. Veelal kan de file als input voor het PP dienen m.b.v. het hulpmiddel, zodat de sessie van de tester wordt nagespeeld. I/O-capture kan dan worden gebruikt om een au-

tomatische test op te bouwen. Nadeel is wel dat als de I/O-interface verandert (nieuwe versie van het PP), de opgenomen sessie onbruikbaar is.
Er zijn twee methodes voor I/O-capture binnen of buiten het PP. Binnen het PP heeft als voordeel dat de capturing "tailor made" is; alleen datgene wordt opgeslagen wat van belang is voor het naspelen van een sessie. Automatisch naspelen is eenvoudig te realiseren. Als nadeel kan worden genoemd dat het PP zelf wordt aangepast waardoor het gedrag anders kan zijn.
Buiten het PP heeft als voordeel dat het transparant voor het PP kan worden uitgevoerd. De capturing kan voor meerdere programma's worden gebruikt. Automatisch naspelen en selectief opslaan is een probleem daar programma's specifieke eisen kunnen stellen aan de presentatie van de invoer. Het probleem bij selectief opslaan is in dit geval, dat het hulpmiddel moeilijk in staat is vast te stellen welke in- of uitvoer van belang is voor de evaluatie van de test of het naspelen van de test.
Een algemeen probleem van I/O-capture is de benodigde opslagruimte op disk. Tijdens de test kan de opslagruimte vol raken en specifieke problemen als het vol raken van opslagruimte kan niet (of heel lastig) worden getest. Ook kan I/O-capture het PP vertragen, waardoor de werking van het PP wordt ontregeld.

3.2 Testdata-generatoren
Testdata-generatoren genereren willekeurige testdata, overeenkomstig opgegeven eisen. Hiermee kan een heel scala van testdata (zowel volgens het PP juiste als onjuiste waarden) worden gegenereerd, waartegen het PP kan worden getest.
Het gebruik van testdata-generatoren kan voorkomen dat het PP voortdurend. tegen dezelfde data wordt getest, waardoor specifieke situaties niet worden getest.

3.3 Padentesters
Padentesters geven een overzicht van de kwaliteit van de test. Het hulpmiddel verdeelt de programmatuur in paden. Een pad is een groep van aaneengesloten statements zonder controlstatements (zoals if, while enz.). Elk pad krijgt een identificatie en een teller welke aangeeft hoe vaak het pad is doorlopen. Deze waarden kunnen na afloop van de test in een overzicht worden weergegeven, zodat van elk pad is af te lezen in welke functie of module het pad zich bevindt, de nestingsdiepte van het pad en hoe vaak het pad door de test is geraakt.
Het is belangrijk dat de padentester vanuit een historische situatie kan werken, zodat in het overzicht de resultaten van meerdere afzonderlijke tests kunnen worden gecombineerd. Paden die door geen van de tests zijn geraakt, dienen aan een nader onderzoek te worden onderworpen. Indien mogelijk moeten er tests worden ontworpen die ook deze paden dekken. Als dit niet mogelijk is, dient de reden hiervoor te worden vastgesteld, omdat niet geraakte paden op overbodige of zelfs foute code kunnen duiden.
Ook bij padentesters geldt als nadeel, dat het programma zelf wordt aangepast. Dit is bij het testen ongunstig, omdat het programma kan worden beïnvloed door de padentester, waardoor het gedrag van het programma anders kan zijn. In sommige omgevingen is het denkbaar, dat het programma met padentester niet kan worden opgestart of zelfs worden gecompileerd, vanwege compilerbegrenzingen of beperkt intern geheugen.
Padentesters zijn in de praktijk lastig in het gebruik voor complete systemen vanwege de hoeveelheid geproduceerde output. Echter voor kleinere tests (functie- of moduleniveau) zijn deze hulpmiddelen relatief eenvoudig te ontwikkelen en blijft de hoeveelheid output handelbaar.

3.4 Call-charts
Call-charts geven een overzicht van de samenhang van het PP. Weergegeven wordt de aanroepstructuur van het PP, welke functies andere functies aanroepen en omgekeerd.
Het nut van zo'n overzicht is dat de tester inzicht krijgt waar eventuele problemen liggen. Functies die veel andere functies aanroepen - en dus veel activiteiten ontplooien - of functies die vaak worden aangeroepen, zijn potentiële probleempunten. Tevens kan de tester zien, welke functies door de test-case geraakt kunnen worden.
De meeste call-charts gebruiken de source-code van het PP, ingeval van een groot systeem kan het samenstellen van een volledige call-chart daarom vrij lastig zijn.

Uit het bovenstaande mag blijken, dat deze hulpmiddelen ook uitstekend bij automatisch testen kunnen worden gebruikt.

4 Testfasen
In dit artikel zullen de volgende fasen onderscheiden worden:
- functietest;
- moduletest;
- integratietest;
- (sub)systeemtest;
- acceptatietest;
- modificatietest.

De nadruk zal liggen op de acceptatietest, de overige zullen meer globaal behandeld worden.
De eerste drie fasen zijn in veel gevallen moeilijk te onderscheiden. Het hangt af van de grootte en complexiteit van een project of deze drie fasen ook daadwerkelijk zullen bestaan. Zo zullen de functie-testfase en de module-testfase enerzijds, de module-testfase en de integratie-testfase anderzijds vaak samenvallen.

Onderstaande figuur geeft aan door welke personen de verschillende fasen worden uitgevoerd.

Fase \ Tester	binnen ontwikkelomgeving		buiten ontwikkelomgeving
	bouwer	beheerder	
functie	X		
module	X		
integratie	X	X	
systeem	X	X	X
acceptatie			X

Figuur 1: Wie test welke fase.

4.1 Functietest

De functietest wordt vrijwel altijd uitgevoerd door de bouwer van die functie (de programmeur). Op dit niveau moet er naar worden gestreefd, het testen automatisch te verrichten, omdat de test over het algemeen elementair is maar toch relatief veel tijd kost. Tevens worden dan minder snel test-items vergeten en kan de test meegroeien met de aanpassingen aan de functie, zodat vroegere foutsituaties altijd worden getest. Dit werkt een betere modificatiecontrole in de hand.

De meest geëigende testmethode voor deze fase is het white box testen. White box testen houdt in het in detail bestuderen van de code om de juiste testmethode te bepalen. Dit heeft als nadeel dat problemen voortkomend uit weggelaten functies niet worden ontdekt. De testmethode wordt afgeleid uit de programmastructuur en is dus afhankelijk van de "code logica".

4.2 Moduletest

De moduletest is in wezen gelijk aan de functietest. Vaak zullen modules onder een beheerfunctie vallen, waardoor de bouwer of aanpasser van de module moet aangeven dat de module naar behoren is getest en correct werkt. Dit kan mede aan de hand van padentesters. Zoals eerder gesteld geeft een padentester een overzicht waaruit kan worden afgelezen hoe de dekking van de tests is. Deze zijn voor complete applicaties vaak onbruikbaar, maar voor geïsoleerde tests als deze zijn ze goed toepasbaar en relatief eenvoudig te bouwen. Ook kan een vaste set testdata worden doorlopen. De tester dient tevens na te gaan of de aanpassing wel door deze set wordt gedekt.

Er dient procedureel te worden vastgelegd welke gegevens moeten worden overhandigd en aan wie. Hierbij moet worden aangetekend, dat om enige modificatiecontrole te hebben en dus de beheersing te vergemakkelijken, de test, indien mogelijk, geautomatiseerd moet worden uitgevoerd.

In deze fase vindt de source review zijn plaats. Een ander dan de bouwer of aanpasser van de module doorloopt de code. De reviewer controleert of aan de programmeervoorschriften is voldaan, of de code begrijpelijk en onderhoudbaar is en voldoende gedocumenteerd (commentaar in de source of anderszins).

Deze laatste stap heeft een aantal alternatieven als code inspections en walkthroughs. In deze gevallen evalueert een groep van drie tot vier mensen (inclusief de auteur) de module in een formele zitting volgens duidelijk afgesproken regels.

Bij een code inspection wordt de code stap voor stap doorgenomen. Tijdens deze zitting legt de programmeur per stap de logica van de module uit. De discussie die hierop volgt kan tot ontdekking van fouten leiden. Daarnaast wordt de code geanalyseerd aan de hand van een lijst met veel voorkomende fouten (zie voor zo'n lijst [MYER79]).

Bij een walkthrough evalueert het team de code aan de hand van testgegevens. Deze testgegevens dienen simpel van aard te zijn en functioneren meer als äanzet tot discussie dan serieuze testinvoer. Ook hier kan de discussie tot de ontdekking van fouten leiden.

Deze twee methoden dienen niet alleen om fouten op te sporen, zij kunnen daarnaast leiden tot kennisvergaring voor de programmeurs op het gebied van algoritmen en programmeerstijlen en -technieken.

4.3 Integratietest

In deze fase worden de verschillende modules van het PP samengevoegd en getest. Dit is op twee manieren te realiseren; incrementeel of niet-incrementeel.

4.3.1 Incrementeel testen

Incrementeel testen werkt als volgt:

Ontwerp, codeer en test een module afzonderlijk, voeg daarna een andere module toe en test deze combinatie. Deze stappen worden herhaald totdat alle modules van een compleet (sub)systeem zijn geïntegreerd. Deze benadering kan worden gesplitst in top-down en bottom-up.

<u>Top-down</u>

Bij top-down wordt eerst de hiërarchisch hoogste module getest en vervolgens worden hiërarchisch lagere modules toegevoegd en wordt deze combinatie getest en zo steeds verder. Bij het testen van in ontwikkeling zijnde systemen zullen nog niet alle modules bestaan. De plaats van niet-bestaande modules wordt ingenomen door zgn. stomp-modules, om het testen van het PP in een vroeg stadium mogelijk te maken. Stomp-modules verrichten hooguit enkele elementaire taken van de module die zij vervangen, meestal bestaan zij uit een enkele return-opdracht.

Voordelen top-down:
- Voordelig als deficiënties (gebreken van het PP) voornamelijk rond de "top" van het systeem optreden. Met de top-down-benadering zullen deze dan snel worden gevonden.
- Als eenmaal de I/O-functies zijn toegevoegd, is de creatie van test-cases eenvoudiger.
- Sluit aan bij een prototyping-achtige ontwikkeling.

Nadelen top-down:
- Stomp-modules moeten worden ontwikkeld, welke vaak complexer blijken te zijn dan oorspronkelijk was aangenomen.
- Voordat de I/O-functies zijn toegevoegd, kan de inpassing van de test-cases in de stomp-modules lastig zijn. Dit is een van de oorzaken van bovenstaand probleem. Het kan nu noodzakelijk zijn, meerdere stomp-modules voor elke test-case te moeten ontwikkelen, vanwege het ontbreken van de I/O-functies.
- Testcondities kunnen moeilijk, zo niet onmogelijk, te creëren zijn.
- De test-output kan slecht worden geïnterpreteerd. Omdat de I/O-modules pas later worden toegevoegd, zal de test-output voornamelijk bestaan uit de gegevens welke aan de toekomstige I/O-modules worden doorgegeven.
- Top-down werkt de misvatting, dat ontwerp en testen kunnen overlappen, in de hand. De completering van de tests voor sommige modules wordt vergeten. Dit komt doordat, door het ontbreken van sommige lagere modules, de te testen module zeer veel test-cases (eventueel verwerkt in stomp-modules) vereist. Het gevaar bestaat dat de tester wacht met het testen van die test-cases, totdat stomp-modules zijn vervangen door de echte modules. Als die modules zijn toegevoegd, wordt vergeten dat hogere modules nog moeten worden getest.

<u>Bottom-up</u>

Bottom-up is de tegenhanger van top-down. Men begint met het testen van elementaire of "bottom-level" modules en voegt daar steeds modules van een hoger niveau aan toe, waarna deze combinatie getest wordt. Deze methode maakt ontwikkeling van test-drivers noodzakelijk om de nog niet toegevoegde modules van een hoger niveau te vervangen. Test-drivers zijn modules die reeds afgeronde modules van een lager niveau aansturen, als de modules van het hoger niveau nog niet gereed zijn.

Voordelen bottom-up:
- Voordelig als deficiënties voornamelijk in de lagere modules optreden.
- Testcondities zijn eenvoudiger te definiëren.
- De test-output is makkelijker te interpreteren, daar de test-output voor een groot deel zal bestaan uit output, die ook door het uiteindelijke systeem wordt gegenereerd.

Nadelen bottom-up:
- Testdrivers moeten worden ontwikkeld.
- Het systeem als entiteit bestaat pas als de laatste module (de top-module) is toegevoegd. Dit is een nadeel, omdat dit de toekomstige gebruiker slecht in staat stelt vast te stellen of het PP aan de gestelde eisen voldoet.

4.3.2 Niet-incrementeel testen

Naast incrementeel kan de integratietest ook niet-incrementeel worden uitgevoerd. Niet-incrementeel wordt ook wel aangeduid als traditioneel testen of "big-bang"-testen.
Traditioneel testen gaat als volgt in zijn werk:
Ontwerp, codeer en test elke module afzonderlijk (unit of moduletest). Vervolgens worden groepen van modules gecombineerd en getest (subsysteemtest). Ten slotte worden alle modules gecombineerd en getest. Zowel ingeval van incrementeel als ingeval van traditioneel testen overlapt de integratiefase de moduletest en de systeemtest voor een deel.

4.3.3 Incrementeel versus traditioneel testen

Zowel aan incrementeel als aan traditioneel testen kleeft een aantal voor- en nadelen.
Als voordelen van incrementeel testen gelden:
Programmafouten betreffende interfaces tussen modules worden sneller ontdekt. Dit komt doordat in een vroeg stadium modules worden gecombineerd, terwijl bij niet-incrementeel testen dit pas aan het eind van het proces gebeurt.
Hieruit voortvloeiend zullen fouten makkelijker te lokaliseren zijn, daar de optredende fouten zeer waarschijnlijk verband houden met de laatst toegevoegde module.
Incrementeel testen zou in grondiger testen kunnen resulteren. Incrementeel testen vervangt steeds de stomp-modules of de test-drivers door reeds geteste modules. Dit heeft tot gevolg, dat de modules "zwaarder belast" worden naarmate de test vordert.
Nadelen van incrementeel testen zijn:
Het samenstellen van testcondities kan een uiterst lastig karwei zijn.
Het is moeilijk te bepalen hoeveel systeemtests nog nodig zijn.
Indien de kwaliteit van de tests afhangt van sommige "sleutel-modules" kan het testproces worden bemoeilijkt, wanneer de ontwikkeling van deze sleutel-modules uitloopt.
Deze voor- en nadelen afwegend, kan men stellen dat incrementeel testen tot betere resultaten kan leiden.

4.4 Systeemtest

De systeemtest dient in wezen als

"acceptatietest"-test. In deze fase wordt getest of het systeem een acceptatietest zal overleven. Daar de testers van deze fase niet (mogen) weten wat (of hoe) er in de acceptatietest wordt getest, is deze fase zo veelomvattend en zwaar mogelijk. Duidelijk is dat hier het documenteren van de tests belangrijk is, zodat naderhand de resultaten van deze fase met die van de acceptatiefase kunnen worden vergeleken.
Dit is zeker van belang wanneer het systeem niet wordt geaccepteerd. Niet alleen moeten dan de onvolkomenheden uit het systeem worden gehaald; ook de systeemtest wordt onder de loep genomen, waarom deze die resultaten niet heeft opgeleverd.
In deze fase, evenals in de volgende, zal naast andere methoden de black box methode gebruikt worden. Black box testen is functioneel testen zonder de code te bestuderen, dat wil zeggen testcriteria opstellen aan de hand van de specificaties. Nadeel daarvan is dat functies of features, die niet in de specificaties voorkomen, niet getest worden. Deze methode is onafhankelijk van de "code logica" en wordt afgeleid uit de specificaties.

4.5 Acceptatietest

Tijdens de acceptatietest zal het PP in het algemeen als geheel worden getest. Dit houdt in dat de test voornamelijk de black box methode zal volgen. Voor de white box methode zal de tester over de afzonderlijke modules moeten beschikken (de object-code en de source-code) waaruit het systeem is opgebouwd. De bouwers van het systeem zullen de source-code van het PP meestal niet willen verschaffen, zeker indien een externe partij de acceptatie verricht. Daarnaast zal de acceptatietester veelal niet over de kennis en/of interesse beschikken om de source te kunnen gebruiken.
Voor zeer kritische systemen echter is het denkbaar, dat de acceptatietest module-gewijs wordt verricht. In dat geval is incrementeel testen, bottom-up of top-down uitgevoerd, heel goed mogelijk.
De acceptatietest wordt uitgevoerd door gebruikers of door vertegenwoordigers van de (toekomstige) gebruikers. Het is belangrijk dat deze testfase door anderen dan de bouwers wordt uitgevoerd. De acceptatietester dient functioneel onafhankelijk van de ontwikkelgroep te zijn. Daarnaast is het nuttig de test door gebruikers te laten uitvoeren, omdat deze belang hebben bij een goed toepasbaar en correct werkend produkt. Meestal wordt de acceptatietest echter door een vertegenwoordigende instantie uitgevoerd, omdat bij de gebruikers veelal de tijd en de kennis ontbreekt om het PP voldoende te kunnen testen.

Wat ten aanzien van testen en documenteren geldt, geldt zeker in het geval van de acceptatietest. Deze test zal door anderen dan de bouwers van het systeem worden uitgevoerd, soms zelfs door een externe instantie. Van groot belang is nu dat de documentatie zodanig is opgesteld, dat deze "self-explanatory" is. Vooraf kan men eisen opstellen waaraan de documentatie moet voldoen.
De documentatie bestaat uit twee delen, het testplan en het testrapport.
Het testplan (pre-testdocumentatie) bevat onder andere:

- beschrijving van het PP;
- te gebruiken hulpmiddelen;
- beschrijving testsets:
  te testen item, te volgen methode(n), te gebruiken hulpmiddel(en), testdata, vereiste resultaat;
- planning.

Het testrapport (post-testdocumentatie) bevat onder andere:

- wat de test betreft, zoals naam en versie of datum van de software, zodat zeker is dat het juiste getest is;
- de gebruikte apparatuur, zoals computer, printers, (grafische) kaarten, etc;
- waarmee is getest, gebruikte hulpmiddelen, zoals padentesters, testdata generators;
- wat er is getest te zamen met resultaten;
- wat er niet is getest.

Voor sommige systemen kunnen andere gegevens ook van belang zijn, zoals wanneer en hoelang getest is.

De beschrijving van wat er niet is getest, is belangrijk voor een eventuele risico-analyse. Uit deze beschrijving kan (in theorie) worden afgeleid, hoe groot de kans is op niet ontdekte fouten. Tevens volgt hieruit hoe volledig het PP is getest.

Een goede documentatie draagt er toe bij dat men inzicht krijgt in de diepgang, doeltreffendheid en volledigheid van de uitgevoerde tests. Een ander uiterst belangrijk punt is, dat goede documentatie herhaalbaarheid en naspeelbaarheid vereenvoudigt.

5 Wanneer testen

Het testen dient zo vroeg mogelijk in het leven van het systeem te beginnen. Uit onderzoeken blijkt, dat het testen meestal pas plaatsvindt na het coderen. Fouten die voortkomen uit een eerdere fase kunnen dan grote problemen geven. Er dient dan ook naar te worden gestreefd reeds in de ontwerpfase te testen, omdat hier de meeste fouten hun oorsprong hebben [HETZ84].
Duidelijk mag zijn dat hoe later een fout wordt ontdekt hoe meer het kost deze te verwijderen.

Het testen van specificaties is echter niet eenvoudig. Het is moeilijk zich in dat stadium een voorstelling te maken, "hoe het er allemaal uit gaat zien".
Er zijn methoden om het insluipen van fouten in de specificaties of in het ontwerp tegen te gaan. De taal waarin of de wijze waarop de specificaties worden geschreven, kan worden gestandaardiseerd. De specificaties moeten eenduidig zijn, ruis en stilte in de tekst moeten worden vermeden [VLIE84].
Ruis is de aanwezigheid in een tekst van een element dat geen relevante informatie bijdraagt tot enig aspect van een probleem.
Stilte is het bestaan van een aspect van een probleem dat in geen enkel punt in de tekst wordt toegelicht.

Voordat met testen kan worden begonnen, moet er een testplan worden opgesteld. Dit testplan dient een beschrijving te bevatten, wat de test betreft, wat er zal worden getest, hoe en waarmee wordt getest.
Het testplan wordt voor een deel opgesteld vanuit de specificaties. Aan de hand van de specificaties wordt een lijst van eisen opgesteld waaraan het systeem moet voldoen. Per eis dient een testset samengesteld te worden. Deze testset bestaat uit een beschrijving hoe de eis wordt getest, eventuele benodigde testdata en de vereiste uitkomst.
Let wel: vereiste uitkomst, een verwachte uitkomst werkt een subjectieve interpretatie van de specificaties in de hand.

Het opstellen van het testplan is een test voor de specificaties; kan het testplan niet opgesteld worden, dan zijn de specificaties niet voldoende.
Hiervoor kan een aantal redenen zijn:

- de specificaties zijn niet eenduidig, er kunnen meerdere conclusies uit worden getrokken;
- de tekst bevat ruis en/of stilte, begrippen en functies worden onduidelijk beschreven;
- de specificaties zijn onvolledig.

De volgende items dienen in het testplan aanwezig te zijn:

- capaciteit ;
- testen van grenswaarden zoals maximum input etc.;
- configuratie:
  - hardware;
  - software;
  - combinaties van deze twee;
- efficiëntie;
- geheugengebruik, zowel intern als extern;
- i/o kanalen;
- interface;
- het testen van de integriteit en timing van interfaces, zoals het koppelen van bestanden of directe communicatielijnen, zowel binnen als tussen systemen;
- prestatie:
  - responstijden;
  - doorlooptijd;
- betrouwbaarheid, omgekeerd evenredig aan de frequentie waarmee fouten optreden.

Een ander aspect van testen, volledigheid, komt hier om de hoek kijken. Het mag duidelijk zijn dat een volledig dekkende testset niet haalbaar is. Dit is niet alleen praktisch gezien onmogelijk ook theoretisch bestaan er barrières tegen "volledig testen", zoals Manna en Waldinger [MANN78] stellen: "We can never be sure that the specifications are correct. No verification system can verify every correct program. We can never be certain that a verification system is correct."
Er zal dan ook een afweging gemaakt moeten worden, wat wel en wat niet getest wordt. Hiervoor kan dan weer gebruik gemaakt worden van risico-analyse.

6 Ergonomie

Ten slotte nog enige aandacht voor de ergonomische aspecten van een systeem in relatie tot het testen.
De tester behoort, naast het correct functioneren van het PP, aandacht te besteden aan de user-interface.
Is deze logisch en eenduidig?
Hoe is de schermindeling, verschijnen boodschappen op een vaste plaats?
Hoe is het gebruik van kleuren, blauw niet als voorgrondkleur vanwege focussering, rood is gevaar, groen is goed/veilig?
Zijn de (fout)boodschappen duidelijk en correct?
Geven de helpfaciliteiten voldoende steun, gerelateerd aan de doelgroep?

De bruikbaarheid van het PP hangt af van de gebruikers waarvoor het PP is bedoeld en hoe goed het PP daarop is afgestemd.
Als gebruikersgroepen kan men onderscheiden:

- onervaren, nieuwe gebruikers;
- regelmatige gebruikers;
- ervaren gebruikers.

Indien het PP voor meerdere groepen is bedoeld, dan dient dat in de user-interface te zijn ondervangen. De helpfaciliteiten moeten begrijpelijk zijn voor die groepen (eventueel door gebruiker in te stellen). Als het PP zowel door onervaren als door ervaren gebruikers wordt gebruikt, dient de interface de onervaren gebruiker goed te begeleiden en de ervaren gebruiker short-cuts te bieden (menu-driven versus command-driven). Tevens dient een ervaren gebruiker de mogelijkheid te hebben commando's af te korten.

7 Personalia

Onno Kluyt is sinds januari 1985 werkzaam bij de sectie Software Engineering van Klynveld EDP

Audit als software engineer. Zijn interessegebieden zijn testen, graphics, user interfaces en datacommunicatie.

8 Literatuur

[CURS86] Cursusmateriaal behorende bij de cursus 'Structured testing'. Frost & Sullivan.

[HETZ84] Hetzel, W: The complete guide to software testing. QED Information Sciences, Inc (ISBN 0-89435-110-9).

[MEYE85] Meyer, B: On formalism in Specifications. IEEE Software vol 2 no 1, Jan. 1985 (ISSN 0740-7459).

[MYER79] Myers, GJ: The art of software testing. Wiley-Interscience (ISBN 0-471-04328-1).

[PEAT] Peat Marwick System Development Manual.

[VLIE84] Vliet, JC.van: Software Engineering. Stenfert Kroese Uitgevers (ISBN 90-207-1298-5).

[MANN78] Manna, Z & Waldinger, R: The logic of computer programming.. IEEE Transactions on Software Engineering, SE-4, 1978.

[BEIZ83] Beizer, B: Software testing techniques. Van Nostrand Reinhold Company (ISBN 0-442-24592-0).

UNIX

door: Ing. A. van der Vlist
Ing. J.C. van Winkel RI

1 Inleiding

Langzaam maar zeker is UNIX[1] volgroeid tot een volwassen besturingssysteem. Via universiteiten en later ook het bedrijfsleven heeft dit tamelijk eigenzinnige besturingssysteem zijn weg niet alleen naar grote technische systemen maar ook naar administratieve omgevingen en personal computers gevonden.

Dit artikel wil ingaan op het ontstaan van UNIX, globaal aangeven hoe het besturingssysteem in elkaar zit en hoe de beveiliging is geregeld. Ook wordt aandacht geschonken aan onderwerpen als standaardisering, derivaten, communicatie en 'hacking'.

2 Historie

Om terug te gaan naar de 'roots' van UNIX, moeten we bijna 25 jaar terug. In 1964 werd door het MIT[2], AT&T Bell Labs en General Electric het ambitieuze MULTICS-project opgestart. MULTICS staat voor Multiplexed Information and Computing Service. Het doel van het project was het bouwen van multi-user, multi-tasking besturingssysteem. Het project mislukte. Vijf jaar later werd voor UNIX de volgende stap gezet, doordat Ken Thompson (Bell Labs) een besturingssysteem voor een GECOS-machine zocht en Bell Labs een vervanging voor het MULTICS-concept wilde hebben. Oorspronkelijk bleven deze initiatieven bij een ontwerp en een simulatie van Thompson op de GECOS-machine maar kort daarna werd er een besturingssysteem, dat hierop gebaseerd was, geïmplementeerd op de PDP-7. Dit systeem was de basis voor het besturingssysteem van de PDP-11. De implementatie werd in 1971 voltooid en Brian Kernighan, eveneens van Bell Labs, bedacht de naam UNICS, van Uniplexed Information and Computing Service, hetgeen later verbasterde tot UNIX. Versie één was geboren en vormde het begin van een lange reeks versies en implementaties. Ken Thompson is hierop nogal trots, hij zegt in zijn artikel 'The UNIX Time-Sharing System' :
'Perhaps the most important achievement of UNIX is to demonstrate that a powerful operating system for interactive use need not be expensive either in equipment or in human effort: it can run on hardware costing as little as $40.000, and less than two man-years were spent on the main system software. We hope, however, that users find that the most important characteristics of the system are its simplicity, elegance, and ease of use. [RITC74]'

De doorbraak voor UNIX kwam in 1975 met versie 6, die gebaseerd was op de taal 'C'. De programmeertaal C had als doel om UNIX portable te maken. Dennis Ritchie, evenals Ken Thompson en Brian Kernighan ook werkzaam bij Bell Labs, ontwikkelde C als een opvolger van de taal 'B' die Ken Thompson ontwikkeld had.

jaar	versie	commentaar
1969	1	PDP-7, PDP-9.
	2	unprotected PDP 11/20
	3	Multiprogramming, PDP-11/34,/40,/45./60,/70.
1971	4	PDP 11/70, Interdata 8/32, ongeveer 600 installaties.
1975	6	Gebaseerd op 'C'.
1979	7	UNIX System III
	8	gebaseerd op v7, BSD 4.0
1989	?	System V 4.0 gebaseerd op BSD 4.2, SunOS, Xenix

Figuur 1: Een aantal versies van UNIX.

Versie 6 was de versie waarin AT&T brood begon te zien. De manier waarop ze hun produkt gingen verkopen was echter, zo blijkt nu althans, zeer uitgekiend. Voor een luttel bedrag van $300 werd UNIX beschikbaar gesteld aan de universitaire wereld. Hier ontstond een snel groeiende enthousiaste club aanhangers die later een sterk uitdragende werking naar het bedrijfsleven had. In 1979 ontstond versie 7. Deze versie is onder de naam UNIX System III[3] uitgebracht op de commerciële markt voor een veel hoger bedrag, $20.000 voor source en object licentie.
Universiteiten en ook het bedrijfsleven gingen later derivaten van UNIX produceren. De meest significante zijn de versies van de universiteit van Berkeley (BSD 2.x voor de PDP-11, BSD 4.x voor de VAX-11) en die van het software house Microsoft (Xenix).
De BSD-versies te zamen met de AT&T-versies itereerden enigszins: de BSD 4.0 was een uitbreiding van AT&T's versie 7, versie 8 van AT&T bevatte weer een aantal zaken van de BSD-versie.

3 Hoe zit UNIX in elkaar?

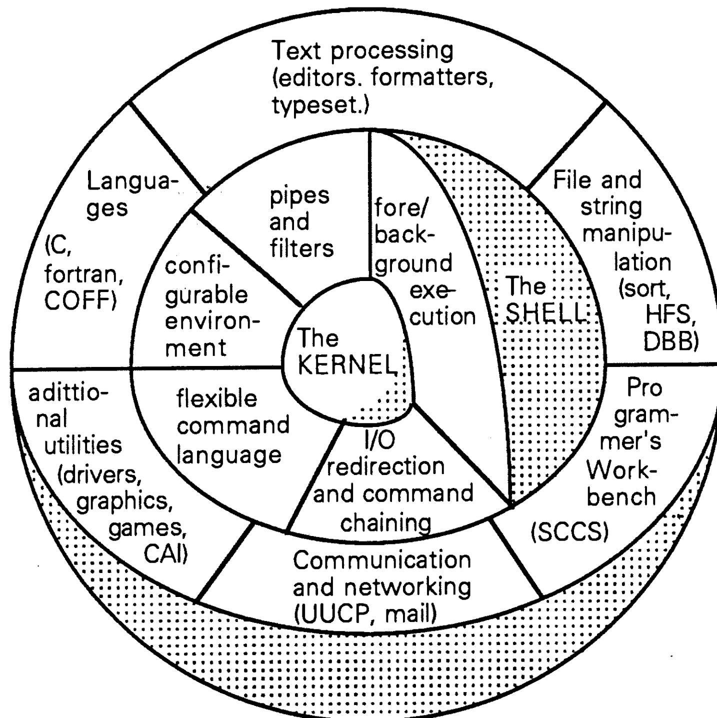
Een aantal begrippen is toonaangevend binnen UNIX. Het is een 'multi-user, multi-tasking' besturingssysteem (ook: 'time-sharing'-systeem) dat

[1] *UNIX is een geregistreerd handelsmerk van Bell Labs (Bell Labs is het research center van AT&T).*

[2] *Massachusetts Institute of Technology.*

[3] *Enigszins verwarrend, versie 7 is identiek aan System III.*

niets anders wil zeggen dan dat meerdere gebruikers (eigenlijk: processen) van één centrale processor unit (CPU) gebruik kunnen maken. Een centrale regelaar verdeelt de CPU-tijd tussen processen die tegelijkertijd 'draaien'. Er zijn overigens ook systemen met meerdere CPU's. De centrale regelaar verdeelt dan de processen over meerdere processoren.



Figuur 2: Het UNIX concept.

3.1 De kernel

UNIX is, zoals de meeste besturingssystemen, opgebouwd uit een aantal schillen. De pit ofwel de 'kernel' vormt het hart van het systeem. De buitenste schil communiceert met de gebruiker of andere omgevingen. Daartussen ligt de 'shell'[4], een systeem dat een gebruikersprogramma de mogelijkheid biedt om te communiceren met de kern. Deze schillen en hun componenten zijn gevisualiseerd in figuur 2.

In de kernel worden de elementaire processen verzorgd zoals het beheer van het geheugen, de processen die in het systeem draaien, communicatie tussen de processen onderling en het filesysteem. Hiervoor is een aantal zogenaamde system calls beschikbaar, zoals het opstarten van een nieuw proces, het aanvragen van meer geheugen, het openen/creëren van files en het lezen/schrijven van/naar files. De kernel zelf is relatief klein, een paar honderd kilobyte.
De kernel verzorgt namelijk alleen het hoogst noodzakelijke (via de system calls), de rest van eventuele beheerszaken wordt door de vele, als gebruikersprogramma's op het systeem draaiende, utilities uitgevoerd. Hier ligt ook de basis voor portabiliteit, als de interface (de system-calls) eenduidig is gedefinieerd, dan houdt dat in dat alle software die daar 'bovenop' draait, portable is. Het is dan ook relatief eenvoudig UNIX aan te passen voor een nieuwe machine, omdat alleen de kernel gedeeltelijk herschreven hoeft te worden.

3.2 De Shell

Bovenop de kernel ligt de 'shell'. De shell vormt de interface tussen de gebruiker en de kernel. Het bevat bijvoorbeeld een uitgebreide commandotaal. Behalve dat met behulp van deze taal simpele commando's ingegeven kunnen worden, is het ook mogelijk om complete programma's in deze (script)taal te schrijven, inclusief variabelen en blok gestructureerde statements als if-then-else-end, while-end, for-end en dergelijke.
Door middel van de shell commando-taal kan de gebruiker processen opstarten, gebruik maken van achtergrond-processing, pipes, filters etc.. Hoe bijvoorbeeld pipes werken in de commandotaal is als volgt: "prep filenaam I sort I uniq I wc". Dit commando 'sluist' de output van het proces via een pipe 'I' steeds naar een volgend proces. In dit voorbeeld wordt een file opgedeeld in woorden (prep), deze lijst wordt gesorteerd (sort), hieruit worden de unieke woorden gehaald (uniq) en ten slotte worden deze woorden geteld (wc).

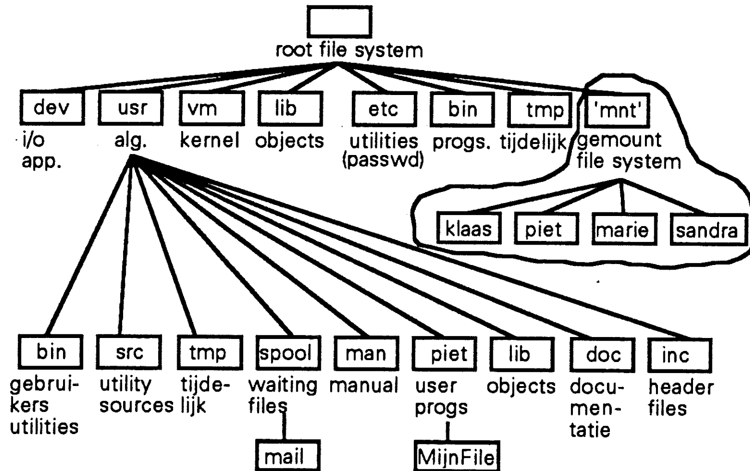
Er zijn momenteel meerdere versies beschikbaar, zoals de C-Shell en de Bourne Shell. De C-Shell is ontwikkeld door de Berkeley Universiteit en de Bourne Shell is genoemd naar de ontwerper, S.R. Bourne. In de praktijk werkt men veelal met de Bourne Shell.

3.3 Het file system

UNIX is een zogenaamd file georiënteerd systeem. Het beschouwt bijna alle entiteiten binnen het systeem als file. Hieronder vallen het geheugen, in- en uitvoer eenheden, terminals enzovoort. Het file system is hiërarchisch opgebouwd. Aan de top zit de 'root', daaronder een aantal directories of files en onder directories weer directories of files. Ook een directory wordt beschouwd als een file.
Dit file system ziet er ongeveer uit als in figuur 3. Een gebruiker heeft een zogenaamde home directory, dat is de directory waar hij binnenkomt bij het inloggen en waar hij werkt. Vanaf die plek (in figuur 3 bijvoorbeeld de directory 'piet') kan de gebruiker zelf een stuk hiërarchie bijmaken voor zijn eigen files.
File- en directory-namen zijn maximaal 14 karakters lang. Om een pad aan te duiden van de root van het file system naar een file toe, worden de namen van de directories en uiteindelijk de file achter elkaar geplakt, met daartussen steeds een

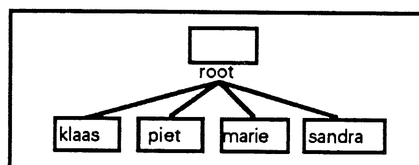
[4] *In dit verslag wordt het woord file voor bestand gebruikt en directory in plaats van directorie alsmede een aantal algemeen bekend geachte Engelse begrippen als i/o, bit etc.*

schuine streep ('/'). De naam van de root directory is leeg, ieder pad vanaf de root directory begint met een '/'. Zo is de volledige padnaam van de file MijnFile in de directory piet in de directory usr onder de root: /usr/piet/MijnFile.



Figuur 3: De UNIX-file-hiërarchie.

Als meerdere file-systemen gebruikt worden, doordat er meerdere schijfeenheden op de machine zijn aangesloten, is het mogelijk om een heel file-systeem (dus de hele boomstructuur van directories en files) onder een directory te 'hangen' van een ander file-systeem. Een file systeem dat op deze manier in een ander file systeem is opgehangen, heet 'gemount'. Een voorbeeld zal een en ander verduidelijken:
In figuur 3 is een directory opgenomen die /mnt heet. Een tweede schijf met de directory-structuur als in figuur 4 kan onder deze directory /mnt gehangen worden. Dit is aangegeven met de omcirkeling. De directory /klaas uit figuur 4 kan dan benaderd worden met de naam /mnt/klaas. Alle paden die beginnen met /mnt zijn dus in feite paden naar files op het gemounte file-systeem.
Het voordeel van deze manier van gebruik van meerdere schijven is dat de gebruiker niet hoeft te weten op welke fysieke schijf zijn gegevens staan, voor de gebruiker is het file systeem opgebouwd als in figuur 3, zonder te weten dat het fysiek gezien eigenlijk twee verschillende file systemen zijn.



Figuur 4: Een mountable file-systeem

Technisch gezien bestaat een UNIX-file-system uit drie delen: het superblok, de i-node-lijst en de datablokken. In het superblok staan gegevens die het file system betreffen: hoe groot het is, wanneer het file system is aangemaakt, hoeveel vrije ruimte er nog is en dergelijke. In de i-node-lijst zijn voor alle files de relevante administratieve gegevens opgenomen, zoals hoe groot de file is, waar deze op de schijf staat, van wie de file is, wie er uit mag lezen, danwel naar mag schrijven en dergelijke. Als laatste zijn in het file system de datablokken opgenomen.
De datablokken bevatten naast de data van de gebruikers ook de directories, omdat directories binnen UNIX gezien worden als (speciale) datafiles. De directory koppelt de naam van een file aan de i-node van de file. Omdat in de i-node van een file alle voor het systeem benodigde gegevens zijn opgenomen, is het niet noodzakelijk om in de directory meer op te nemen dan alleen de naam van de file en het nummer van de bijbehorende i-node.

Door deze manier van koppelen van filenaam en i-node, is het mogelijk om één i-node (en dus de bijbehorende file) aan meer dan een naam te koppelen. Dit heeft als voordeel dat als twee mensen dezelfde file ter raadpleging nodig hebben, ze niet ieder een kopie van de file hoeven te hebben, maar met één en dezelfde file kunnen werken, terwijl toch de file in de directory van beide personen vermeld staat. Omdat alle administratieve gegevens van de file in de i-node staan, is de beveiliging van de file voor beide personen identiek.

3.4 Communicatie tussen UNIX-systemen

Tussen twee UNIX-systemen kunnen met behulp van het UUCP (UNIX to UNIX CoPy) protocol files gekopieerd worden en berichten verstuurd worden. Omdat alle UNIX-systemen hetzelfde protocol gebruiken, is het tussen alle UNIX-systemen mogelijk berichten te versturen en files te kopiëren.
Het kopiëren van files gaat net als met het normale kopieercommando, behalve dat voor de filenamen steeds de naam van de machine waarop de file staat opgenomen moet worden. Zo is naam van de file /usr2/aart/file1 op de machine kpmgnl bekend bij buren van de machine kpmgnl als kpmgnl!/usr2/aart/file1. Als een bepaalde machine geen directe buur is van de machine waarop de file staat, kan net zo vaak de naam van een volgende buurmachine opgenomen worden tot wel de machine in kwestie bereikt is. Bijvoorbeeld: mcvax!botter!ark/user1/jcvw/file2 is de file /user1/jcvw/file2 op de machine ark die een buur is van de machine botter, die een buur is van de machine mcvax, die een buur is van de machine waarop gewerkt wordt.
Op dezelfde manier kan ook mail verstuurd worden aan gebruikers op andere machines. Is op de machine ark een gebruiker jcvw bekend, dan kan die persoon geadresseerd worden als mcvax!botter!ark!jcvw.
Omdat zeer veel UNIX-machines wereldwijd in een op deze wijze opgezet netwerk zijn opgenomen (USENET), is het mogelijk om met

UUCP en mail vrijwel de hele wereld over te komen. Met name universiteiten maken hiervan veelvuldig gebruik in de vorm van bijvoorbeeld bulletin boards voor actuele onderwerpen.
Om de computer tegen ongewild kopiëren of overschrijven te beveiligen, is van systemen en van gebruikers in een tweetal files opgenomen welke commando's ze mogen gebruiken, en welke directories ze mogen benaderen.

Tegenwoordig vindt ook veel communicatie tussen UNIX-systemen onderling en tussen UNIX-machines en niet-UNIX-machines plaats met behulp van Ethernet en het protocol TCP/IP. Dit heeft ten opzichte van UUCP als voordeel dat de snelheid veel hoger ligt. UUCP blijft bij grotere afstanden een veel gebruikt protocol, omdat het bereik van Ethernet altijd beperkt blijft tot enkele kilometers.

4 Beveiliging van UNIX

Het loslaten van UNIX op universiteiten is een goed middel gebleken om het besturingssysteem grondig te testen op zijn (on)veiligheden. Er wordt op de universiteiten enorm veel gekraakt, 'gehackt' en ingebroken. Hierdoor zijn veel lekken en gaten in het systeem aan het licht gebracht en gedicht (Zie voor dergelijke 'hacks': [FILI86] en [REID86]).
De beveiliging van een computersysteem valt naast de organisatie rondom het computersysteem uiteen in hardware-matige beveiliging en software-matige beveiliging. Bij hardware-matige beveiliging kan gedacht worden aan de systeemconsole die afgeschermd moet staan, beveiligde ruimtes etc.. Deze paragraaf wil echter ingaan op de software-matige beveiliging van UNIX.

4.1 De Superuser

De Superuser is de beheerder van het systeem. Hij is hierdoor ook de grootste risicofactor. Hij kan en mag alles. Om een voorbeeld te geven: als de Superuser is ingelogd en hij bevindt zich in de root directory, dan zorgt het commando "rm -f /*" ervoor dat het hele systeem (de complete file hiërarchie) gewist wordt. Binnen Systeem V zijn er wat extra accounting-mogelijkheden zoals het loggen van alle superuser log-ins.
Niet alleen "ingelogd" zijn als superuser is een probleem maar ook programma's die van 'root' zijn (dus de Superuser), en een speciaal permissiebit aan hebben staan, vormen een mogelijk risico. Als inbrekers die programma's kunnen vervangen door eigen 'getrapte' programma's, dan kunnen inbreekactiviteiten worden uitgevoerd met permissies van de Superuser.

4.2 De beveiliging van files

Omdat UNIX een file georiënteerd systeem is, kan gezegd worden dat als een 'file' goed beveiligd kan worden, het hele systeem beveiligd kan worden (in concept). Voor iedere file is een aantal bits in de i-node van de file opgenomen dat aangeeft, hoe de permissies ten opzichte van die file zijn geregeld, zie figuur 5. Dit is van wezenlijk belang en daarom zal kort worden aangegeven hoe in detail de protectie is opgebouwd.
Per file is in drie niveaus geregeld wie er iets mag doen, de eigenaar (user) zelf, de groep waarin de eigenaar en anderen in zitten (group) en de rest van de gebruikers. Een gebruiker die iets met een file wil doen heeft altijd één vaste status, hij is of de eigenaar, of hij zit in dezelfde groep als de eigenaar van de file of hij is een 'andere' gebruiker. Voor elk niveau is aangegeven of men bevoegd is om te lezen (R), te schrijven (W) of te executeren (X). Een file heeft in de hogere bits ook een aanduiding voor een directory of file (bit 12 t/m 14) en een i-node-teken bit (15). Deze permissiestructuur wijkt af van de structuur bij veel andere systemen, waar per file expliciet een lijst van gebruikers is aangegeven met permissies tot die file. Dit kan in UNIX gesimuleerd worden door groepen goed in te delen en ook applicaties (bijvoorbeeld het "Source Code Control System", SCCS) te laten controleren op bevoegdheden.

Directories zijn, zoals al opgemerkt, een speciale vorm van files. De RWX-permissies zijn vrijwel identiek aan die van 'gewone' files. Omdat de consequenties van de bits intuïtief gezien toch 'vreemd' overkomen, hieronder een overzicht van de betekenis van de drie bits:

- R: Uit de directory mag gelezen worden, bijvoorbeeld om file-namen met wildcards (zoals ? en *) te expanderen en om een inhoudsopgave te geven aan de gebruiker;
- W: In de directory mag geschreven worden. Het verwijderen van een file betekent niets anders dan het schrijven in een directory dat die bepaalde file niet meer aanwezig is. Derhalve kan iedereen voor wie het W bit aanstaat voor de directory, alle files uit die directory verwijderen, ook al heeft de betreffende persoon geen lees- of schrijfpermissie op de files in die directory!
- X: Uit de directory mag gelezen worden als de directory onderdeel is van een padnaam.

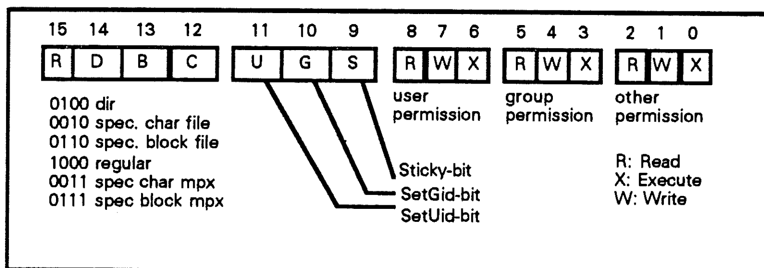
Om bijvoorbeeld de file /usr/klaas/MijnFile te kunnen openen moet degene die de file wil openen, X-permissie op /usr en /usr/klaas hebben, en R-permissie op /usr/klaas/MijnFile. Om de file /usr/klaas/MijnFile te kunnen verwijderen moet degene die de file wil verwijderen, X-permissie op /usr en /usr/klaas hebben, en bovendien W-permissie op /usr/klaas. De permissies van de file /usr/klaas/MijnFile zelf zijn dan niet van belang.
Voor de beveiliging zijn naast de RWX-bits voor user, group en other ook de SETUID- en SETGID-bits[5] belangrijk. Deze twee bits geven namelijk 'extra' permissies aan de file. De werking van het

5 *Dit SETUID en SETGID permissiesysteem is gepatenteerd door AT&T Bell Labs.*

SETUID is om onder controle van een programma (waarvan dat bit aanstaat en dat geëxecuteerd wordt door een gebruiker) de privileges van de eigenaar van dat programma te verstrekken tijdens het draaien van dat programma. Normaliter heeft een programma de privileges van de gebruiker van een programma. Het SETGID-bit werkt identiek maar dan voor de groeppermissie. De "effective user ID" is de gebruiker waarnaar het systeem kijkt bij de beoordeling van de privileges die het programma nodig heeft, de 'real user id' is de gebruiker die het programma opgestart heeft. Dit kan problemen geven, getuige het volgende voorbeeld:

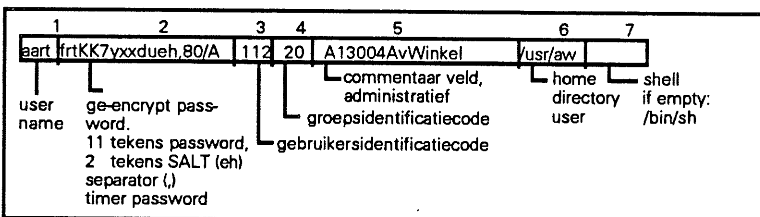
```
main() { system("sh"); }
```

Dit simpele C-programma 'stapelt' een nieuwe shell op de draaiende shell, waarin men dezelfde mogelijkheden heeft als in de gewone shell. Als men echter eenmalig als Superuser dit programma compileert en aanlinkt en daarna het commando "chmod 4111 programma" ingeeft waardoor het SETUID-bit wordt aangezet, dan kan daarna een gewone gebruiker dit programma opstarten en komt hij terecht in een nieuwe shell met de privileges van de Superuser.



Figuur 5: File-permissies.

Het is van belang om, binnen het kader van de veiligheid van UNIX, er zeker van te zijn dat de systeembeheerder op de hoogte is van het bestaan van alle files van de eigenaar 'root' (superuser) met het SETUID-bit aan. Deze vormen bijna per definitie een risico.



Figuur 6: Een regel uit de password-file.

4.3 Het Password-mechanisme

Het password-mechanisme in UNIX is een 'open' systeem. Onder de directory /etc staat een file, /etc/passwd, die de log-in-informatie, passwords en permissies van alle gebruikers bevat. De beveiliging van deze file is dus van groot belang. Normaliter is deze file van de Superuser en mag niemand schrijven op de directory /etc. Als men bijvoorbeeld in staat zou zijn om deze file te overschrijven, dan kan een gebruiker een eigen passwd-file daar neerzetten met voor hem gunstige permissies. De password file is de enige plaats waar passwords, permissies en user logins zijn vermeld. In deze file worden de passwords ge-encrypt opgeslagen met behulp van een variant van het DES[6] algoritme. Met behulp van het zogenaamde salt-programma wordt een string opgebouwd en met deze string kan men 4096 verschillende DES-varianten genereren. Deze salt-string wordt ook in de password file opgeslagen (zie figuur 6).
De password file is de enige plaats binnen het UNIX-file-system waar iets over gebruikers wordt gedefinieerd. Daarom kan de systeembeheerder een gebruiker toevoegen aan het systeem door een nieuwe regel in te voeren met de nieuwe gegevens. Als op de plaats van het ge-encrypte password niets is ingevuld, kan men zonder password inloggen op de betreffende user.
Het inloggen onder UNIX kan ook werken met een timer, zodat gebruikers regelmatig hun password moeten wijzigen.

4.4 Hardware gekoppelde devices

Het beschouwen van hardware gekoppelde devices als files is een andere risicofactor van het ver doorgevoerde file-mechanisme 'concept' in UNIX. Hardware gekoppelde files die een mogelijke risicofactor vormen zijn (disk)drives, geheugen en terminals. Zo is bijvoorbeeld het intern geheugen van de hardware de file /dev/mem. Dit is gevaarlijk omdat als een willekeurige gebruiker inlogt, zijn password altijd gedurende korte tijd aanwezig is in het geheugen; als een andere gebruiker op dat moment de file /dev/mem kan doorkijken met het commando "strings /dev/mem"[7], dan kan hij dus dat password zien. Read-permissie op dergelijke files moet dus worden afgeschermd. Ook terminals vormen op deze wijze een probleem. Door zogenaamde escape sequences te sturen naar files die eigenlijk terminals zijn, ontstaan mogelijkheden voor diegene die moedwillig passwords te weten wil komen.

4.5 Checkpoints bij beveiliging

Hoewel het niet mogelijk is een complete lijst te geven van checkpoints bij een onderzoek naar de beveiliging van UNIX, kan toch een lijstje gegeven worden van mogelijke lekken:

- Alle directories waarin systeemprogrammatuur staat mogen alleen leesbaar zijn, anders zou het mogelijk zijn een programma te verwijderen en er een eigen versie voor in de

---

6 *Data Encryption Standard.*

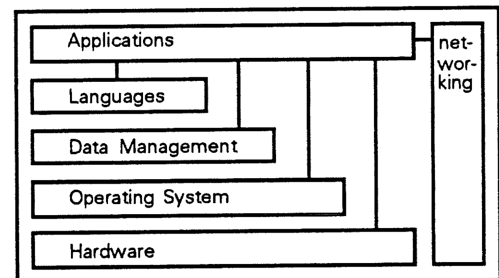
7 *Met het 'strings' commando worden uit een file alle strings afgedrukt die uitsluitend leesbare karakters bevatten.*

plaats te zetten.

- Executeerbare files mogen niet door anderen dan de eigenaar overschrijfbaar zijn, anders zou het mogelijk zijn in software van anderen wijzigingen aan te brengen. Deze wijzigingen zouden zich over het hele systeem kunnen verspreiden (computer-virus).
- Alle directories met onderin de structuur, de files die tegen overschrijven beveiligd moeten zijn, mogen alleen toegankelijk zijn voor lees-operaties. Het zou anders mogelijk zijn om een complete nieuwe sub-boom van directories te maken en deze in de plaats te zetten van de oorspronkelijke sub-boom.
- Alle device files moeten op de juiste wijze afgesloten zijn, anders kan zoals eerder vermeld afgetapt worden wat anderen intikken.
- Maak regelmatig een lijst van alle programma's die als eigenaar 'root' hebben, en ook het SETUID-bit aan hebben staan. Dit moet een zo klein mogelijke lijst zijn; van elk van de programma's op deze lijst moet te verantwoorden zijn waarom ze SETUID aan hebben. Programma's van root met SETUID aan, hebben tijdens executie de privileges van root, de systeembeheerder.
- Als gebruik is gemaakt van shell-scripts, of eigen programma's om gebruikers af te schermen van het normale UNIX-systeem, maar er wordt in die programma's of shell scripts wél gebruik gemaakt van standaard UNIX-programmatuur (zoals bijvoorbeeld de editor), kan men toch het standaard systeem in. In vrijwel alle interactiëve standaard UNIX-programma's is het namelijk mogelijk om, met behulp van een commando, een shell op te starten. Een handige faciliteit die niet eenvoudig af te schermen is.
- Let op wat het standaard zoek-pad (de PATH environment variabele) voor 'root' is. Hierin mag niet de 'huidige' directory voorkomen. Als dit wel het geval is, zou een gebruiker een commando kunnen maken dat precies zo heet als een standaard UNIX-commando, maar met geheel andere effecten. Een voorbeeld: iemand kan een commando ls maken dat naast hetzelfde werk dat het normale commando ls ook zou doen, extra werk verricht. Als de systeembeheerder de huidige directory niet in zijn zoekpad heeft staan, zal de normale ls (uit de directory /bin) uitgevoerd worden. Als daarentegen de huidige directory wel in het zoekpad staat en de systeembeheerder staat in een directory met een executable file die ls heet, zal de locale ls uitgevoerd worden.

5 Standaarden en groepen

Momenteel is UNIX beschikbaar in velerlei versies en uitvoeringen. Verschillende fabrikanten hebben hun eigen implementatie, gebaseerd op een 'standaard' en uitgebreid met eigen ideeën. Ook zijn er specialistische technische versies, commerciële versies en research-versies. De UNIX-versie, waar iedereen zich momenteel op richt, is System V van AT&T. AT&T heeft daar zelf een standaard voor ontwikkeld, de System V Standard Interface Definition (SVID).
Er ontstond ook buiten AT&T een sterke behoefte aan een standaard en er ontstond een samenwerkingsverband tussen een aantal hardware-leveranciers, de X/Open-groep. Deze groep, in 1984 door ICL als Europees samenwerkingsverband opgestart, bestaat uit ICL, Nixdorf, Olivetti, Bull, Siemens en Philips. Met name de deelname van Olivetti is belangrijk omdat AT&T een van de grootste aandeelhouders is van Olivetti. Binnen dit samenwerkingsverband streeft men naar een standaard die tendeert naar System V[8]. De standaard van de X/Open-groep bestaat uit een 'Common Applications Environment' (CAE), zie figuur 7. Voor elk onderdeel is een aantal standaarden gedefinieerd.



Figuur 7: CAE van X/Open [X/OPE85].

In Nederland is er een gebruikersgroep voor UNIX, de NLUUG wat staat voor National UNIX Systems User Group - The Netherlands. De Europese landelijke groepen zijn weer gebundeld in de EUUG, de European UNIX Systems User Group. Een vergelijkbare groep in de Verenigde Staten is de USENIX Association.
Daarnaast is er een aantal groepen van leveranciers met meer commerciële belangen in UNIX-produkten, de /usr/group in de Verenigde Staten en de /usr/group/UK in Groot-Brittannië.

De meeste leveranciers richten zich momenteel op de UNIX System V van AT&T. Hoogstwaarschijnlijk wordt dit dan ook de standaard van UNIX. Buiten deze spontane generatie

8 *In Nederland is X/Open vertegenwoordigd als werkgroep onder de Vifka (Vereniging van Importeurs en Fabrikanten van Kantoor Apparatuur). Zij hebben drie soorten leden: stemgerechtigde (Bull Nederland, Digital Equipment BV, Ericsson Information Systems, Hewlett-Packard Nederland BV, ICL Nederland BV, Nixdorf Computer BV, NCR Nederland NV, Olivetti Nederland BV, Philips Telecommunicatie- en Informatiesystemen BV, Siemens Nederland NV, Unisys Nederland NV), niet stemgerechtige (Data General Nederland BV, IBM Nederland BV, H.A Kramers & Zn BV, MAI Nederland BV, MDS Nederland NV, Omas BV, Prime Computer Benelux BV, Salomons Groep) en gastleden die niet stemgerechtigd zijn (NLUUG).*

van System V zijn twee andere standaarden van groot belang: de POSIX (Portable Operating System for Computer Environment) standaard en de X/Open standaard. De POSIX[9] standaard is opgezet door de IEEE (Institute of Electrical and Electronics Engineers). In deze standaard wordt de C-interface gedefinieerd tussen de gebruiker en het besturingssysteem zodat applicaties portable zijn. De X/Open standaard is opgezet door de X/Open-groep en staat eigenlijk hetzelfde voor als de POSIX standaard.

6 Versies

Vanuit een aantal hoeken worden UNIX-systemen geproduceerd. UNIX blijkt namelijk goed te voldoen in specifieke technische en wetenschappelijke omgevingen maar ook meer en meer in commerciële en technische omgevingen.
Andere produkten willen onafhankelijk zijn of zijn juist gebonden aan bepaalde hardware.
In deze paragraaf zal een aantal UNIX-systemen genoemd worden.
De huidige UNIX-systemen zijn in twee soorten beschikbaar: hardware afhankelijke en hardware onafhankelijke systemen. Bijvoorbeeld HP-UX (gericht op meer grafische systemen) is een UNIX van HP, gebonden aan hun hardware. Aan de andere kant zijn bijvoorbeeld UNIPLUS+ (van Unisoft) en XENIX (van Microsoft) machine-onafhankelijk.

6.1 Mini- en mainframes

Vooral in de technische en wetenschappelijke hoek is UNIX een populair besturingssysteem. Een (onvolledige) opsomming van systemen is:

DEC	VAX	Ultrix-32
	PDP-11	Ultrix-11
Prime	50 serie	Primix
HP	HP9000	HP-UX
IBM	4300, 9370	VM/IX370
	6150	AIX

Verder is UNIX beschikbaar op Amdahl (470, 5800), Sperry (1100), CDC Cyber, Data General MV, IBM Series 1 etc..

6.2 Microcomputers

Vooral op de microcomputermarkt is UNIX populair aan het worden. Ontwikkelingen op dit gebied zijn bijvoorbeeld de AIX-versie van IBM voor de PC/RT-architectuur en de AUX-versie van Apple voor de Macintosh II-architectuur.
Wel aardig om te noemen is een Nederlandse versie, ontwikkeld door Prof. A.S. Tanenbaum van de Vrije Universiteit in Amsterdam. Hij heeft UNIX opnieuw geschreven in de taal 'C' en is daardoor in staat om samen met het boek dat hij over dit project heeft geschreven, ook de sources van MINIX (UNIX) toe te voegen als bijlage [TANE87].

6.3 Secure UNIX

Er zijn ook UNIX-versies die meer gericht zijn op beveiliging zoals bijvoorbeeld UTX/32 van Gould Electronics. Dit was in feite de eerste SUNIX wat staat voor Secure UNIX en wil compatible zijn met de NSDD-145[10]. Hierin zitten uitgebreide accounting-mogelijkheden, meerdere afgeschermde Superusers etc..

6.4 Verdere ontwikkelingen

AT&T en Sun zijn bezig met het ontwikkelen van UNIX System V, versie 4.0. Hierin zullen System V, MicroSoft Xenix System V, SunOS en BSD 4.2 worden gebundeld. Het samenwerkingsverband van AT&T en Sun werkt ook aan een grafische gebruikers-interface voor UNIX, 'Open Look', gebaseerd op de technologie van de Xerox Star. Deze nieuwe versie moet voldoen aan de POSIX standaard van IEEE, de X/Open standaard en uiteraard de SVID standaard zelf. Deze versie moet in 1989 als beta-release beschikbaar komen.

Een onverwacht scherpe reactie op de 'Open Look' ontwikkelingen van AT&T en Sun is, het oprichten van de 'Open Software Foundation' door een groep van onafhankelijke hardware-leveranciers (IBM, DEC, HP, Apollo, Nixdorf, Bull en Siemens). De doelstelling van OSF is ook om een UNIX-standaard te ontwikkelen naast de Open Look versie van AT&T en Sun. De OSF-UNIX zal gebaseerd zijn op AIX van IBM.

Op het gebied van de mini- en microcomputers voor UNIX gaan de ontwikkelingen snel. In UNIX INFO wordt melding gemaakt van 31 leveranciers van wat men UNIX Boxes noemt.
Bijvoorbeeld een lap-top schootcomputer van 'GRID Systems' uit Californië die UNIX draait en werkt met een batterijvoeding (model 1513, 386 processor).

7 Conclusies

- Op de mainframe-markt groeit UNIX langzaam maar zeker, vooral in technische en wetenschappelijke omgevingen.
- Op de microcomputermarkt zal het waarschijnlijk sterk gaan concurreren met MS-DOS en OS/2 van Microsoft.
- Standaardisering is noodzakelijk. Alles tendeerde tot voor kort naar System V. De vraag is of er nu twee onafhankelijke UNIX standaarden komen (OSF en Open Look). In ieder geval komt deze splitsing UNIX niet ten goede.
- Qua beveiliging is UNIX misschien niet

9 *IEEE Std 1003.1.*

10 *National Security Decision Directive, 1984, Verenigde Staten.*

geschikt voor specifieke, zeer geclassificeerde overheidsprojecten maar dat zijn 90% van de besturingssystemen niet. Over het algemeen is UNIX met wat kennis goed af te sluiten.

- De manier waarop UNIX groeit is geen uitbarsting; UNIX houdt geen 'verpletterende zegetocht' maar beweegt zich in een gestage opwaartse spiraal.
- UNIX zal zich verder moeten bewijzen op de volgende punten:
  - het moet in staat blijken de huidige, ingeburgerde systemen te verdringen;
  - de User Interface moet verbeteren;
  - er moet een goede standaard komen.
- UNIX zal in de negentiger jaren alleen maar groeien.

8 Personalia

Aart van der Vlist is sinds oktober 1984 werkzaam bij KPMG Klynveld EDP Audit sectie Software Engineering als Software Engineer. Hij is in 1984 afgestudeerd aan de HIO Den Haag. Zijn interessegebieden zijn UNIX, computer graphics, object oriented languages en artificial intelligence.

Jan Christiaan van Winkel is sinds juli 1984 werkzaam bij KPMG Klynveld EDP Audit sectie Software Engineering als Software Engineer. Hij is in 1984 afgestudeerd aan de HIO Eindhoven. Zijn interessegebieden zijn Operating Systems, waaronder UNIX, computervirussen, datacommunicatie en EFT (Electronic Funds Transfer). Hij is lid van de Vereniging van Registerinformatici (VRI).

De sectie Software Engineering, één van de twee ontwikkelcentra van KPMG maakt gebruik van UNIX als ontwikkelomgeving voor audit software als KPMG's File Analysis Product Line en het pakket Palet.

9 Literatuur

[AUST86] Austen, GJM, Thomassen, HJ: UNIX, het standaard OS Academic Service, Den Haag.
[BINN84] Binnant, BF: Benchmarking UNIX systems, Byte, augustus 1984.
[CURS86] Cursusmateriaal behorende bij de cursussen 'Geavanceerd programmeren met UNIX en C' en 'UNIX voor systeembeheerders'. @ AT Computing bv, Nijmegen.
[DATA85] Databus mei 1985, UNIX themanummer.
[FILI86] Filipski, A, Hanko., J: Making UNIX secure. Byte april 1986.
[KERN78] Kernighan, B., Ritchie, DM: The C Programming Language. New Jersey, Prentice Hall inc.
[KERN84] Kernighan, BW, Pike, R: The UNIX programming Environment. Prentice Hall, New Jersey, 1984.
[KEUN86] Keunig, P, Vermuë, E:UNIX. HIO, Eindhoven, 1986.
[REID86] Reid, B: Lessons from the UNIX Breakins at Stanford, ACM SIGSOFT SE vol 11 Oct 1986.
[RITC74] Ritchie, DM, Thompson, K: The UNIX Time-Sharing System. ACM Communications vol 17, no 7, juli 1974 (pag. 365).
[TANE87] Tanenbaum, AS: Operating Systems; Prentice Hall, Amsterdam, 1987.
[UNIX87] UNIX-Info okt/nov 1987, jaargang 1, nummer 2,.
[UNIX88] UNIXWORLD, bijlage Computerworld februari 1988, jaargang 2, nr 1.
[XENE79] XENIX Programmers Manual Part I and II, Microsoft, 1979.
[X/OPE85] X/OPEN Portability guide. Elsevier Science Publishers B.V. 1985.

Computervirussen

door: Ing. J.C. van Winkel RI

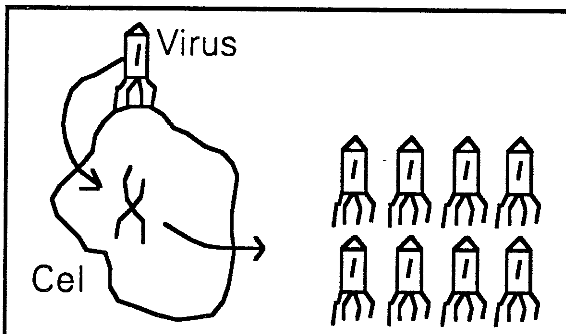
1 Inleiding

In dit artikel zal een uiteenzetting worden gegeven over de eigenschappen van computervirussen; hoe ze werken, hoe computers besmet worden met een virus en hoe dit voorkomen danwel genezen kan worden.

2 Wat is een computervirus?

De werking van een computervirus lijkt zeer sterk op de werking van een biologisch virus. Om de analogie met een biologisch virus duidelijk te maken, zal eerst een beschrijving worden gegeven van de globale werking van een biologisch virus. Vervolgens zal op het computervirus worden ingegaan.

Een biologisch virus bestaat uit een eiwitwand met daarbinnen genetisch materiaal.



Figuur 1: Het biologische virus valt de cel aan, injecteert het genetische materiaal in de cel en zet deze aan tot reproduktie van het virus.

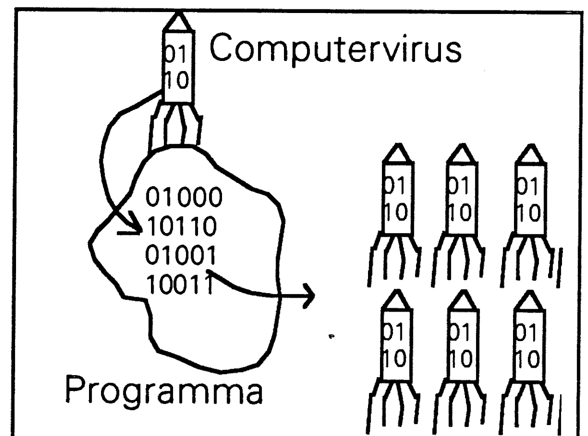
Zodra een biologisch virus in aanraking komt met een slachtoffer-cel in de omgeving, zal het virus via een gaatje in de celwand het genetische materiaal (het 'virus-gen') injecteren. Dit genetische materiaal zal de cel instrueren kopieën van het virus te maken (zowel het gen als de eiwitschaal). De geïnfecteerde cel gaat door met de produktie van nieuwe virussen totdat er zo veel aangemaakt zijn dat de cel openbarst. De virussen stromen hierna de cel uit, op zoek naar nieuwe slachtoffers. Een op deze wijze aangevallen cel kan niet meer normaal functioneren.

Het duurt enige tijd voordat zo veel cellen zijn aangetast dat ziekteverschijnselen merkbaar zijn. De tijd die verloopt tussen eerste aanraking met het virus en het naar voren komen van infectieverschijnselen, is de incubatietijd.

Een biologisch virus hoeft niet meteen actief te worden na injectie van het gen in de cel, het kan ook enige tijd 'rusten' voordat het de cel aanzet tot produktie van meer virussen. Dit kan doordat het injecteren van het virus-gen spontaan gebeurt, terwijl voor het reproduktie proces bepaalde activeringsstoffen nodig zijn.

De werking van een computervirus is analoog aan dat van een biologisch virus.

Een computervirus is een programma dat in staat is zichzelf bij andere programma's in te nestelen. Als het door het gastheerprogramma geactiveerd wordt, zal het virusgedeelte in de 'omgeving' zoeken naar onbesmette programma's. Hierna wordt het virusgedeelte uit het programma gekopieerd naar het onbesmette programma. Het gastheerprogramma kan vervolgens zijn normale functie uitvoeren. Het virus hoeft geen bijwerkingen te hebben zoals het biologische virus dat wel heeft, omdat het slachtofferprogramma niet door de besmetting in functionaliteit hoeft te verminderen.



Figuur 2: Het computervirus valt het programma binnen, injecteert het virusgedeelte en zet het programma aan tot reproduktie van het virus.

De meeste tot op heden ontdekte computervirussen hebben echter wél neveneffecten, doordat ze aan de hand van een bepaald criterium, bijvoorbeeld de systeemdatum, extra activiteiten ontplooien. Voorbeelden hiervan zijn: het onmogelijk maken van de verwerking op het computersysteem door het wissen van extern geheugen, het oproepen van congestie in een netwerk, het plegen van frauduleuze handelingen en dergelijke.

3 Definities

Omdat de verschillende termen die in de literatuur voorkomen vaak inconsequent gebruikt worden, hieronder een drietal definities:

Een Trojan horse is programmatuur die op slinkse wijze verborgen is in een programma. Het

programma zal meer doen dan het lijkt te doen. De meeste voorbeelden van bekende Trojaanse paarden zijn aantrekkelijke programma's (meestal spelletjes of handige utilities) die naast hun legitieme werk ook schijven wissen. Het Trojan horse kan zichzelf niet bij andere programma's innestelen. Een biologisch equivalent van het Trojan horse is een met cyaankali vergiftigde taart: de taart lijkt lekker, totdat er een hap van genomen wordt.

Evenals bij een Trojan horse is bij een time-bomb extra programmatuur opgenomen in normale programmatuur. Het verschil met Trojan horse is gelegen in het feit dat de extra activiteiten pas op een later moment worden uitgevoerd. Time-bombs zijn bijvoorbeeld wel eens door ontslagen programmeurs opgenomen in (financiële) programmatuur om na hun ontslag 'wraak' te nemen door belangrijke gegevens te wissen. Een biologisch equivalent van een time-bomb is een met een kankerverwekkende stof vergiftigde taart.

Een computervirus is een Trojan horse dat in staat is zich steeds weer bij andere programma's in te nestelen, zonder dat in eerste instantie de functionaliteit van de geïnfecteerde software aangetast wordt. De destructieve kant van het Trojan horse hoeft niet per se direct tot uiting te komen, het kan net als bij de time-bomb wachten met toeslaan. Zo kan een computervirus wachten tot het zich ingenesteld heeft in een programma met privileges van de systeembeheerder. Een biologisch equivalent van een computervirus is besmetting met het griepvirus door omgang met besmette personen.

4 Hoe werkt een virus?

We zullen nu op een meer technisch niveau bekijken wat er precies door het virus ondernomen moet worden om zich daadwerkelijk te kunnen verspreiden.

Het virus zal als eerste onderzoeken of er in de omgeving nog onbesmette programma's zijn. Dit kan, door aan het besturingssysteem een lijst te vragen van alle programma's in de huidige directory, op de huidige schijf, of zelfs in het hele computersysteem. Van alle op de lijst voorkomende programma's kan onderzocht worden of ze niet al eerder besmet waren.

De niet eerder besmette programma's zullen nu vóór de eerste instructie, de instructies van het virus ingevoegd krijgen. Dit betekent dat het programma groter wordt. Daarom is het ook noodzakelijk om alleen niet eerder besmette programma's te besmetten, anders zouden de programma's steeds groter worden tot er op den duur geen opslagcapaciteit op het computersysteem meer beschikbaar is. Eventueel kan gebruik gemaakt worden van compressietechnieken om het virus plus het gastheerprogramma samen net zo groot te laten zijn als het gastheerprogramma zelf.

Om het virus uiteindelijk over het hele computersysteem te verspreiden hoeft in feite slechts één programma per keer geïnfecteerd te worden. De verspreiding van het virus zal echter sneller gaan als per keer meerdere programma's geïnfecteerd worden.

Na het virusgedeelte gekopieerd te hebben zal het normale programma uitgevoerd worden. Een voorbeeld kan het een en ander verduidelijken:

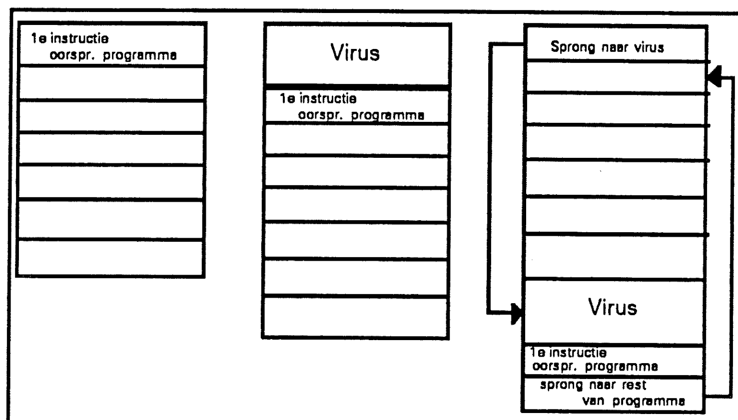
```
program geïnfecteerd;

begin
  procedure virus;
  begin
    for een of meer nog onbesmette
            executeerbare files do
      begin
        kopieer het virus vóór het
            geselecteerde programma
      end
  end

  procedure normal_program;
  begin
    .
    .
    .
  end

  {hier begint de werkelijke executie}
  call virus;
  call normal_program;
end
```

Naast het zichzelf vermenigvuldigen doet het virus niets. Vaak zal het virus ook nog een Trojan-horse-activiteit ontplooien, zoals het wissen van het externe geheugen. Om zo veel mogelijk schade toe te kunnen brengen, en om op zo veel mogelijk systemen te kunnen infecteren, zal het virus proberen om ontdekking zo lang mogelijk te rekken. Daarom zal de maker van het virus het uitvoeren van de kwalijke activiteiten uitstellen totdat aan een bepaalde voorwaarde wordt voldaan (bijvoorbeeld een bepaalde datum). Ook kan het virus wachten met toeslaan totdat het (binnen het computersysteem) zo groot mogelijke privileges heeft.

5 Indeling in het geheugen

In figuur 3 staan drie geheugenindelingen geschetst. De eerste indeling betreft een normaal, onbesmet programma. De computer zal het programma uitvoeren vanaf de eerste in-

structie, bovenin het programma. De eerste instructie van het programma is in de figuur expliciet benoemd. In de tweede indeling is het virus vóór het programma geplaatst. Duidelijk is dat het geïnfecteerde programma groter is dan het onbesmette.



Figuur 3: Geheugenindeling bij onbesmette programma's en twee soorten virussen. De eerste instructie van het oorspronkelijke programma is expliciet weergegeven.

Een probleem dat het virus tegen kan komen is dat alle absolute adressen, die in het gastheerprogramma zijn opgenomen, veranderd moeten worden. Als bijvoorbeeld in het gastheerprogramma een instructie staat: Spring naar de instructie op adres 1000, en het virus is 200 groot, dan zal de instructie veranderd moeten worden in een sprong naar adres 1200. Het virus kan dat tegengaan door tijdens het kopiëren van de viruscode het virus niet vóór, maar juist achter het programma te zetten. Om er toch voor te zorgen dat de viruscode als eerste uitgevoerd wordt, wordt de eerste instructie van het programma vervangen door een sprongopdracht naar de eerste instructie van het virus. Het virus zal dan als laatste activiteit terugspringen naar het normale programma.

Het is voor het virus van belang dat de virale code als eerste wordt uitgevoerd, omdat een programma over het algemeen wel slechts één beginpunt heeft waar het virus ingevoegd kan worden, maar meerdere eindpunten kan hebben. Ieder van die eindpunten zou dan een sprongopdracht naar het virus moeten krijgen. Bovendien zal, als in het programma systeemfouten optreden, het programma door het systeem gestopt worden, zonder dat het virus aan bod is gekomen.

De instructie die overschreven werd door de sprong naar het virus, wordt als één na laatste instructie in het virus opgenomen. De pijlen volgend in de derde indeling, is te zien dat er functioneel gezien hetzelfde gebeurt als in de tweede indeling.

6 Virussen bij verschillende computers

Het effect van virussen wordt sterk bepaald door de beveiligingsmogelijkheden van de 'slachtoffer'-computer. Daarom wordt eerst een opsomming gegeven van een aantal verschillende typen computersystemen waarop virussen kunnen voorkomen.

6.1 De Personal Computer

Bij PC's gelden over het algemeen geen beperkingen wat betreft de mogelijkheid in andere bestanden te kunnen schrijven. Daarom kunnen PC's een broedplaats zijn voor virussen. Programma's worden door gebruikers vaak gekopieerd of van een bulletinboard[1] gehaald, waardoor het virus zich over meerdere PC's kan verspreiden. Als bovendien een aantal PC's in een netwerk is opgenomen, kan een virus zich snel verspreiden. Voorbeelden: MS-DOS[2] computers en de Apple[3] Macintosh.

6.2 Gemeenschappelijk gebruikte computers

Hieronder vallen de computers voor wetenschappelijk gebruik, universiteitscomputers en dergelijke, meestal supermicro's, mini's of supermini's, draaiend onder een multi-user, multitasking besturingssysteem. Zulke computers hebben over het algemeen goede beveiligingsmogelijkheden, maar daar wordt niet altijd even goed gebruik van gemaakt. Bovendien is het beveiligingssysteem van de universiteitscomputers een uitdaging voor studenten. Vaak zullen deze computers opgenomen zijn in (grote) netwerken, waardoor het verspreidingsgebied van een virus wereldwijd kan zijn. Voorbeelden: UNIX[4] systemen opgenomen in USENET, DEC[5] computers opgenomen in DECNET.

6.3 Grote systemen in een 'echte' produktieomgeving

Hieronder vallen computersystemen van grote

---

1 *Een bulletinboard is een centrale computer waar gebruikers met PC's berichten voor elkaar kunnen achterlaten, maar ook programma's voor algemeen gebruik ter beschikking kunnen stellen ('Public Domain' software en 'ShareWare'). Voor public domain software hoeft niet betaald te worden, voor shareware wordt vaak een gering bedrag gevraagd als het programma nut blijkt te hebben.*

2 *MS-DOS is een geregistreerd handelsmerk van MicroSoft Corporation.*

3 *Apple en MacIntosh zijn geregistreerde handelsmerken van Apple Computer Inc.*

4 *UNIX is een geregistreerd handelsmerk van AT&T Bell Laboratories.*

5 *DEC en DECNET zijn geregistreerde handelsmerken van Digital Equipment Corporation.*

financiële instellingen zoals banken en verzekeringsmaatschappijen. Deze computers werken veelal met software die door het bedrijf zelf ontwikkeld is. In ieder geval wordt geen gebruik gemaakt van shareware of public domain software. Vaak is de ontwikkelomgeving zeer strikt gescheiden van de produktie-omgeving, en wordt software van het ontwikkelsysteem eerst na uitgebreide acceptatietesten overgezet naar het produktiesysteem. Derhalve zal een virus in een dergelijk produktiesysteem weinig kans maken behalve als het virus door iemand van de ontwikkelafdeling wordt gemaakt en het virus zich tijdens ontwikkeling en testen niet openbaart (een virus met time-bomb-eigenschappen). Op het ontwikkelsysteem kan een virus wél toeslaan, vooral ook omdat dergelijke omgevingen vaak minder goed afgeschermd zijn.

7 Hoe wordt een computer 'besmet'?

Virussen kunnen zich binnen het computersysteem verspreiden, maar verspreiding naar andere computersystemen is ook mogelijk. Dit kan dan vooral geschieden via netwerken en het uitwisselen van programmatuur met behulp van overdraagbare opslagmedia (bijvoorbeeld diskettes) en de grote netwerken en bulletin boards.

Uit de beschrijving van de werking van een virus is duidelijk geworden dat het voor het virus noodzakelijk is te kunnen schrijven in andere programma's. Daarom zijn PC's zeer kwetsbaar. Alle bestanden zijn immers vrij te beschrijven. Doordat bovendien de systeemdatum zeer eenvoudig te wijzigen is, is het overschrijven van bestanden ook moeilijk te controleren. Alleen computersystemen die een deugdelijk beveiligingssysteem hebben, maken een kans een virusaanval af te wenden.

8 Het bereik van besmetting

Virussen kunnen zich verspreiden binnen de transitieve afsluiting van de bevoegdheid in bestanden te schrijven. Een voorbeeld kan dit verduidelijken: Als gebruiker A in staat is in bestanden van gebruiker B te schrijven, en gebruiker B is in staat in bestanden van gebruiker C te schrijven, kan een virus in programmatuur van gebruiker A via gebruiker B in programmatuur van gebruiker C komen, ook al kon gebruiker A oorspronkelijk niet in bestanden van gebruiker C schrijven. Als één van de gebruikers bovendien de mogelijkheid heeft via een netwerk in andere systemen te komen, is de weg voor het virus open.

Bij UNIX bijvoorbeeld is het voor het virus mogelijk om steeds méér privileges te krijgen. Een programma heeft namelijk de privileges van degene die het programma opstart. Als er in een programma PROGJAN van gebruiker Jan een virus zit, en gebruiker Piet gebruikt het programma PROGJAN, dan heeft PROGJAN de privileges van Piet, en kan derhalve in alle bestanden schrijven waarin Piet kan schrijven, en dus ook in het programma PROGPIET van Piet. Stel dat de systeembeheerder Jan niet vertrouwt (om welke reden dan ook), maar hij vertrouwt wel Piet, dan kan het virus van Jan via Piet alsnóg systeembeheerdersprivileges verkrijgen.

9 Uitkomsten van een experiment

Prof. F. Cohen heeft proeven gedaan met virussen op een UNIX-systeem [COHE84]. Zijn metingen geven verontrustende resultaten te zien: Een virus was in circa 8 uur te bouwen en behelsde 200 regels programmacode. Het eerste virus werd verborgen in een aantrekkelijk, vrij te gebruiken programma dat op een grafische manier de hiërarchische directory-structuur van een UNIX-systeem liet zien. Bij de vijf proeven die genomen werden bleek, dat het virus in gemiddeld 5 minuten systeemprivileges verkregen had. De kortste tijd waarbinnen het virus de macht over het gehele systeem verkregen had was één minuut, de langste tijd 60 minuten. De tijd nodig om één infectie uit te voeren was een halve seconde, waarbij opgemerkt moet worden dat bij het ontwerpen van het virus niet op efficiëntie gelet werd.

10 Detectie van besmetting

Virale infectie is zeer moeilijk te detecteren, zeker als enige intelligentie in het virus is opgenomen en als het virus wacht met nevenactiviteiten tot voldoende programmatuur geïnfecteerd is. Een mogelijkheid is om virussen te detecteren aan de hand van de grootte van executeerbare programmatuur.

Omdat virussen code toevoegen aan programmatuur, is het mogelijk te testen op de grootte. Veel programmatuur bevat echter een dusdanig grote hoeveelheid redundantie, dat met behulp van moderne compressietechnieken (zoals tokenizing[6] en huffman coding[7]) de extra door het virus opgenomen ruimte, eenvoudig kan worden teruggewonnen. Nadat de programmatuur wordt opgestart, zal het virus voor de executie van het

6 *tokenizing is het samenvoegen van veel voorkomende combinaties van codes tot één nieuwe code, zoals ook in het dagelijks gebruik vaak afkortingen gebruikt worden.*

7 *huffman coding is een compressietechniek waarbij voor veel voorkomende codes nieuwe codes gebruikt worden die korter zijn, terwijl voor weinig voorkomende codes langere codes gebruikt worden. Vergelijk: Morse code, daar is bijvoorbeeld de code voor de veelvoorkomende letter E een punt, terwijl de code voor de weinig voorkomende letter Q streep streep punt streep is.*

oorspronkelijke programma deze eerste weer decomprimeren, zodat de normale machinecode terugkomt.
Ook zal door het schrijven in bestaande programmatuur de modificatiedatum van de programmatuur veranderd worden. Bij PC's is het echter iedereen toegestaan de systeemtijd te wijzigen, zodat dit geen probleem oplevert, omdat vóór het infecteren de systeemtijd gezet kan worden op de aanmaakdatum van de programmatuur, waarna ná de infectie de systeemtijd weer teruggezet wordt op de werkelijke tijd.
De infectie zal een zekere hoeveelheid tijd kosten, zeker als meer dan één programma per keer geïnfecteerd wordt. Bij PC's zal de extra tijd wellicht opvallen, maar bij grotere systemen die meerdere taken tegelijkertijd kunnen uitvoeren, hoeft dit niet het geval te zijn. Ten eerste is de responsetijd van dergelijke systemen nooit constant, maar afhankelijk van de drukte op het computersysteem. Ten tweede is het voor het virus mogelijk een proces op te starten dat parallel loopt aan het normale programma. Het normale programma start dan vrijwel meteen op, zoals normaal, terwijl in de achtergrond het virus zijn werk doet.

Het is niet mogelijk een algemeen programma te maken dat van andere programma's kan bepalen of ze geïnfecteerd zijn of niet. Het onbeslisbare 'Halting Problem'[8] is namelijk reduceerbaar tot het probleem van virusdetectie, en daarmee is het probleem van virusdetectie minstens net zo moeilijk als het halting problem. Wel kan, als eenmaal de karakteristieken van een bepaald virus bekend zijn, programmatuur geschreven worden die test of programmatuur met dat specifieke virus besmet is.

Met behulp van hash-totals is het mogelijk verandering van programmatuur te ontdekken, hoewel hierop ook door een virus geanticipeerd kan worden. Derhalve zullen de hash totals opgeslagen moeten worden op een niet overschrijfbaar medium. Bovendien moeten de hash totals zo veel mogelijk gebruik maken van one-way functions[9]. Een voorbeeld: Bij het berekenen van een hash total worden alle bytes van het programma modulo 256 opgeteld. Als het virus nu bytes toevoegt aan het programma met als hash-waarde 56, kan het virus eenvoudig een extra dummy byte toevoegen met als waarde 200. Hierdoor wordt het virus wat betreft de hash-functie neutraal (200+56) mod 256 = 0 mod 256. Zo zal het hash-totaal van het geheel (programma + virus) dezelfde waarde krijgen als die van alleen het programma. Door gebruik te maken van one-way functions is het niet meer mogelijk eenvoudig te berekenen wát aan het virus moet worden toegevoegd om een goede hash-waarde te verkrijgen.

11 Hoe wordt besmetting voorkomen?

Besmetting kan worden voorkomen door het aanpakken van de 'vrije' communicatie binnen het computersysteem en tussen computersystemen onderling. Dit kan onder andere door:

- geen programmatuur van anderen te 'lenen';
- geen gebruik te maken van public domain en shareware software;
- uitsluitend officiële programmatuur te gebruiken;
- niet toe te staan dat in executeerbare programmatuur geschreven wordt;
- programmatuur te encrypten en vlak voor executie te decrypten. Een probleem is dan het beheer van de encryptie- en decryptiesleutels;
- alleen zeer specifieke programma's de privileges te geven om executeerbare files aan te maken, bijvoorbeeld compilers en linkers.

Het is een misvatting dat computervirussen zich uitsluitend in machinetaal programma's zouden kunnen verschuilen. Virussen kunnen zich nestelen in ieder programma dat geschreven is in een voldoende krachtige geïnterpreteerde taal. Voorbeelden zijn: BASIC-programma's, shell scripts of job control statements, database queries of zelfs editor macro's. De enige eis die gesteld wordt is dat de taal waarin programma en virus geschreven zijn, voldoende faciliteiten biedt om stukken programma te kopiëren naar andere programma's.
Voornoemde is meteen ook een bedreiging voor computersystemen: het is mogelijk de meest ingewikkelde beveiligingen op te werpen tegen virussen op machinetaalniveau. Als virussen op een niveau hoger toch nog kunnen toeslaan hebben de beveiligingen op de lagere niveaus minder zin dan verwacht.
Virussen bestaan bij de gratie van interpreteerbaarheid van data. Een programma dat uitvoer levert, genereert deze uitvoer alléén opdat deze uitvoer later geïnterpreteerd zal worden, anders zou de executie van het programma geen zin hebben. Deze uitvoer kan echter ook instructies bevatten om de 'uitvoer' weer in andere programma's te implanteren (besmetting). Daarom is het zeer moeilijk virussen in algemene zin uit te

[8] *Het halting problem zegt dat het niet mogelijk is om een algoritme te maken dat van alle programma's kan bepalen of ze zullen stoppen of niet.*

[9] *Een one-way function $f(x)=y$ is een functie waarvan het zeer moeilijk is om aan de hand van het beeld y het origineel x te berekenen. Zo is het eenvoudig om $y = x^a \bmod b$ te berekenen, maar gegeven a, b en y is het zeer moeilijk $x = {}^a\log(y) \bmod b$ te berekenen. Dit soort functies wordt veelvuldig gebruikt in de moderne cryptografie.*

bannen. Immers, virussen zijn uitsluitend uitgebannen als programma's geen interpreteerbare uitvoer meer zouden hebben, maar ieder programma heeft uitvoer die op een of andere manier geïnterpreteerd wordt.

12 Hoe wordt een computer 'genezen'?

Doordat het onmogelijk is in algemene zin besmetting vast te stellen, is er ook geen algemene genezing mogelijk. Wel kan met behulp van een voor een specifieke besmetting geschreven vaccin, de besmetting genezen worden. Dit vaccin kan pas geschreven worden als de werking van het virus geheel duidelijk is.
Een genezend programma kan alle programma's in het computersysteem onderzoeken op het specifieke virus. Als geconstateerd wordt dat een bepaald programma geïnfecteerd is, kan het virus uit het programma verwijderd worden.
De beste maatregel is, steeds bij het binnenkrijgen van nieuwe software, het distributiemedium te beveiligen tegen overschrijven en uitsluitend met kopieën te werken. Als daarnaast ook consequent back-up's gemaakt worden van de gegevens, kan bij een infectie eenvoudig helemaal opnieuw begonnen worden zonder verlies van gegevens.

13 Onzichtbare virussen

Zeer gevaarlijk zijn virussen die zich hebben weten te nestelen in programma genererende programmatuur, zoals compilers, assemblers, linkage editors en loaders. Omdat deze programma's bedoeld zijn om andere programma's te manipuleren, zal het niemand opvallen als deze programma's op slinkse wijze extra code in programma's opnemen. Zeker als de compilers en linkers nieuwe versies van zichzelf aanmaken, kunnen ze ervoor zorgen dat de extra code ongemerkt in de programmatuur blijft zitten, zoals hieronder wordt verhaald:
Het is mogelijk Trojaanse paarden of virussen op te nemen in programmatuur, zonder dat er ook maar één statement in broncode van het virus op het computersysteem aanwezig is. Kenneth Thompson, een van de makers van UNIX, heeft in de lezing die hij gaf naar aanleiding van de door de ACM[10] aan hem uitgereikte Turing Award, hierover al gesproken [THOM84]. Hij had het met name over de mogelijkheid om in een compiler een zichzelf reproducerend virus op te nemen dat bij het genereren van een nieuwe compiler steeds weer blijft terugkomen. De moraal van zijn lezing was: Vertrouw geen enkele programmatuur tenzij je die zelf in de computer inbrengt.

10 ACM: Association for Computing Machinery, de Amerikaanse collega-organisatie van het Nederlands Genootschap voor Informatica (NGI).

14 Voorbeelden van virussen

Er is de laatste tijd een aantal virussen ontdekt dat vermeldenswaard is:

14.1 Mail-virus

Bij sommige IBM[11] systemen is het mogelijk om berichten te versturen aan medegebruikers, eventueel zelfs via een netwerk. De gebruikers hebben een lijst met daarin vermeld de medegebruikers, waaraan wel eens berichten verstuurd worden. Het is mogelijk om bij de berichten ook opdrachten te versturen, die uitgevoerd moeten worden door de ontvanger. In december 1987 heeft iemand een kerstboodschap verstuurd aan een aantal medegebruikers, maar in het bericht de opdracht opgenomen het bericht automatisch ook te versturen aan alle mensen die in de berichtenlijst van de ontvanger staan. Dit gaf een dusdanig cascade effect dat het netwerk, waarop de machines aangesloten waren, totaal verstopt raakte. Hoewel het geen virus is in de strikte zin van het woord, leek het toch zinnig dit voorbeeld op te nemen.

14.2 Israël-virus

In Israël is een virus ontdekt waarvan het molestgedeelte geactiveerd had moeten worden op 13 mei 1988 [MCLE88]. Het virus werd ontdekt omdat het zichzelf ook implanteerde in reeds besmette programmatuur. Sommige programma's waren zelfs 300 keer besmet. Hierdoor werd zoveel schijfruimte gebruikt en werd het computersysteem zo traag, dat normale verwerking niet meer mogelijk was. Hierdoor gealarmeerd heeft men het virus op tijd (vóór 'ontploffen') kunnen localiseren en men heeft het onschadelijk kunnen maken.

14.3 PC-virus

Er is een virus ontdekt op MS-DOS-computers. Het virus nestelt zich in de commando interpretator (COMMAND.COM) van alle schijven die bereikbaar waren. Na het uitvoeren van vier MS-DOS-commando's worden bestanden gewist. Het detecteren van de aanwezigheid van dit virus is mogelijk doordat het moet schrijven in de commando interpretator. Door deze op een niet overschrijfbare schijf te zetten, zal de computer een foutboodschap genereren als het virus toch probeert COMMAND.COM te overschrijven. De fout in dit virus is, dat het niet vóór infectie test of het wel in het slachtoffer kan schrijven.

15 Conclusies

Zeker PC's met hun vrijwel totaal vrije commu-

11 IBM is een geregistreerd handelsmerk van International Business Machines.

nicatie binnen de PC, zijn zeer kwetsbaar. Het gebruik van onbekende software moet zeer zorgvuldig geschieden. Bulletin boards bieden grote verspreidingsmogelijkheden voor virussen. Ook netwerken geven virussen vrij baan, zeker als het één type computer betreft, zoals bij fabrikantgebonden netwerken, zoals AppleTalk[12], DECNET en SNA[13]. Virussen in machinetaal weten zich dan zeker van een goede 'innestelingsplaats' aan de andere kant van de communicatielijn.

Omdat virussen zich kunnen verplaatsen is het niet mogelijk te zeggen welke soort programma's virussen zouden kunnen bevatten. Hoewel de allereerste generatie van een virus vaak geïmplanteerd is in een spelletje of handige utility, kunnen na verspreiding in principe alle programma's besmet zijn. Een advies in de trand van: "Kijk uit voor niet-officiële of niet zelf gemaakte spelletjes en kleine utilities" is bij virussen dan ook minder waardevol dan bij simpele Trojan horses.
Bij tot op heden ontdekte virussen is eigenlijk een (domme) fout in het virus de oorzaak van vroegtijdige ontdekking. Als de virusbouwers minder slordig zijn, wordt het virusprobleem een stuk complexer.
Om ingedekt te zijn tegen besmetting, dienen regelmatig back-ups gemaakt te worden, waarbij data en programmatuur gescheiden moeten worden bewaard om herstel met 'veilige' programmatuur mogelijk te maken. Mocht zich ooit een besmetting voordoen, is het mogelijk om alle schijven te initialiseren/formatteren, de originele (onbesmette) software van distributiemedia te kopiëren, en de gegevens van de back-up. Een probleem dat zich kan voordoen is dat virussen eventuele back-up's hebben kunnen verminken door zich in de back-up-programmatuur te nestelen.

Ten slotte kan gezegd worden dat er op dit terrein nog veel onderzoek noodzakelijk is. Met name dienen meer mogelijkheden voor detectie van virussen, de genezing van computersystemen en het voorkomen van infecties gevonden te worden.

12 *AppleTalk is een geregistreerd handelsmerk van Apple Computer Inc.*
13 *SNA is een geregistreerd handelsmerk van International Business Machines.*

16 Personalia

Jan Christiaan van Winkel is sinds juli 1984 werkzaam bij KPMG Klynveld EDP Audit sectie Software Engineering als Software Engineer. Hij is in 1984 afgestudeerd aan de HIO Eindhoven. Zijn interessegebieden zijn Operating Systems, waaronder UNIX, computervirussen, datacommunicatie en EFT (Electronic Funds Transfer). Hij is lid van de Vereniging van Registerinformatici (VRI). Momenteel is hij, in het kader van zijn studie Informatica aan de Vrije Universiteit onder Prof. A.S. Tanenbaum, bezig met een afstudeerproject over computervirussen.

17 Literatuur

[COHE84] Cohen, F: Computer viruses, Theory and experiments, Dept. of Computer Science and Electric Engineering, Lehigh Univ. USA.

[DIER86] Dierstein, R: Computer Viruses a new threat to computers and computing, The First European Conference on Computer Audit, Control & Security Oslo Norway.

[POZZ8?] Pozzo, MM, Gray, ET: An approach to containing computer viruses, Computer Science Department, UCLA.

[MCLE88] McLellan, V: Computervirus verspreidt zich snel over de wereld, De Volkskrant van zaterdag 6 februari 1988, voorblad van het katern Wetenschap en Samenleving.

[THOM84] Thompson: Reflections on the trusting trust, CACM aug. 1984 pp 761-763 (Turing Award lecture).

OBJECTS

door: Ing. L.J.M.W. Gielen

1 Inleiding

De ontwikkeling van programmeertalen en -technieken staat niet stil. Met de regelmaat van een langzaam tikkende klok wordt melding gemaakt van nieuwe ontwikkelingen. Sommige daarvan zijn gebaseerd op het gebruik van 'objects'. Het werken met objects, het object georiënteerd programmeren, is hard op weg zich een belangrijke plaats te veroveren in de informatietechnologie.

In dit artikel over object georiënteerd programmeren wordt ingegaan op de algemene begrippen en de daarbij gebruikte terminologie. Voor meer diepgaande beschrijvingen wordt verwezen naar de literatuurlijst ter afsluiting van dit artikel.

2 Software-crisis

In een snel tempo doet de computer zijn intrede in de wereld van alledag. Het aantal toepassingsgebieden schijnt onbeperkt. Een computer is een medicus behulpzaam bij het stellen van een diagnose, administraties van bedrijven en verenigingen worden geautomatiseerd en kerncentrales worden bestuurd en gecontroleerd door computers.
Programma's die deze computers besturen worden complexer en fouten in diezelfde programma's kunnen steeds grotere gevolgen hebben. Mede hierdoor worden steeds hogere eisen gesteld aan te ontwikkelen programmatuur en wordt het steeds moeilijker om aan de groeiende vraag naar programmatuur te voldoen. De toenemende complexiteit maakt het onderhouden van programma's een kostbare aangelegenheid. Ook kan bij aflevering van een programma blijken dat het niet voldoet aan de gestelde eisen; tijdrovende en daardoor kostbare aanpassingen zijn dan nodig.
Het tekort aan goede programmatuur, dat is ontstaan als gevolg van bovenstaande problemen, wordt vaak aangeduid met de term 'software-crisis'.

De bestrijding van de software-crisis richt zich hoofdzakelijk op het toepassen van verbeterde ontwikkelmethodieken. Ook door hergebruik van software en door het inschakelen van gebruikers in het ontwikkeltraject, kan een beter produkt gerealiseerd worden.
Door daarnaast eisen te stellen aan de structuur van een programma, poogt men te komen tot een goed, aanpasbaar produkt.
Twee van de belangrijkste programmeertechnieken zijn information hiding en data abstraction die een goede opbouw van het produkt in modulen afdwingen.
Data abstraction is een techniek waarbij de representatie van een door een ontwikkelaar gedefinieerd data-type en de op dit type werkende subroutines, verborgen worden in één module. De term 'information hiding' refereert naar meer algemene richtlijnen voor het opdelen van de programmacode in afzonderlijke modulen.

Het object georiënteerd programmeren is een 'techniek' waarin data abstraction impliciet is opgenomen. Ook biedt de techniek extra constructies die het hergebruik van software bevorderen.
Bij het zoeken naar oplossingen voor de software-crisis, wordt de laatste jaren in toenemende mate gebruik gemaakt van de door het object georiënteerd programmeren geboden voordelen.

3 Concepten

Het object georiënteerd programmeren is gebaseerd op een aantal concepten; zie ook [PASC86]. Deze concepten kunnen niet geheel gescheiden behandeld worden. Helaas is er geen sprake van een consistente terminologie van de concepten bij de verschillende object georiënteerde programmeertalen. Daarom zullen bij bespreking van de concepten de meest gehanteerde Engelstalige termen gebruikt worden.

3.1 Object

In de wereld om ons heen is een object een voorwerp, zaak of persoon. Een object heeft zekere kenmerken, kan worden gemanipuleerd en soms kan een object zelfstandig acties uitvoeren. Een object is een duidelijk aan te wijzen, afgebakend deel van onze wereld. Alle objecten samen vormen de wereld. Voor object georiënteerd programmeren geldt in feite precies hetzelfde: een object beschrijft een deel van het probleemgebied, alle objecten samen het programma.

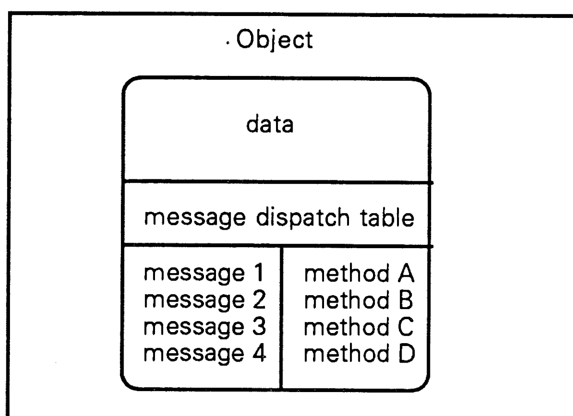
In object georiënteerd programmeren is een object een software-component, te beschouwen als een 'intelligente' combinatie van data en procedures.
Een object kan bijvoorbeeld gezien worden als een document (data) met bijbehorende handleiding (procedures). Indien gegevens op het document gewijzigd moeten worden wordt gekeken in de handleiding hoe dit moet gebeuren.

3.2 Message dispatch table

Bij object georiënteerd programmeren wordt een object door middel van een message, een aanvraag voor een actie en/of resultaat, verzocht een actie uit te voeren. Alle objecten communiceren

met elkaar door middel van deze messages. Na ontvangst van een message zal het object een aan dit message gerelateerde method uitvoeren. Een method kan in het eerder genoemde voorbeeld worden gezien als een deel van de bij het document behorende handleiding. Bij elke message van een object hoort een method en vice versa. De tabel, waarin de relatie wordt gelegd tussen de messages en de methods, wordt de message dispatch table genoemd.

In figuur 1 is een schematische weergave van een object gegeven. Te zien is dat het object uit twee delen bestaat: de data en de message dispatch table.



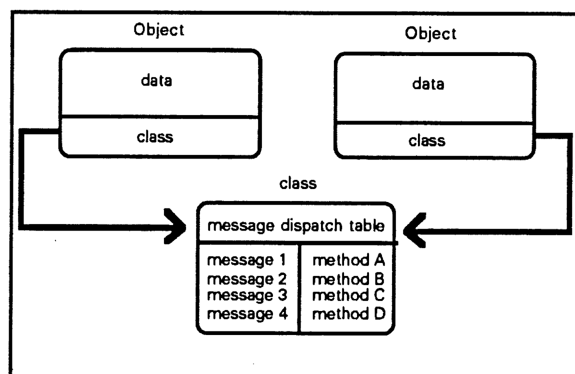
Figuur 1: Schematische weergave van een object

Een message is vergelijkbaar met een subroutine-aanroep in een conventionele taal en kan ook parameters bevatten. Een method is in deze vergelijking de procedure-body en een object kan gezien worden als een record. De record-definitie wordt dan class genoemd.

3.3 Class

Object georiënteerd programmeren zou problematisch zijn als voor elk object afzonderlijk de methods en messages gespecificeerd moeten worden. Vergelijkbaar zal niet voor elk document een aparte handleiding geleverd worden. Groepen van objecten die op dezelfde wijze reageren op identieke messages, kunnen natuurlijk meer efficiënt beschreven worden. Zo wordt veelal voor een aantal documenten een en dezelfde handleiding gebruikt. De handleiding wordt dan 'losgekoppeld' van het document. Evenzo kan de message dispatch table worden losgekoppeld van het object. Deze 'message dispatch table' wordt dan class genoemd. De class geeft niet alleen aan hoe een object gemanipuleerd kan worden, maar ook hoe deze eruit ziet. Moet een nieuw object aangemaakt worden of wordt een object vernietigd dan wordt aan de class gevraagd om dit te doen. Daarom wordt de class ook wel een factory genoemd. De class bevat beschrijvingen van de in zo'n situatie te ondernemen acties.
Een object dat bij een class hoort wordt de instance van de class genoemd. Bij elke class kunnen verscheidene instances behoren. De verzameling messages, behorende bij een class, wordt aangeduid met de term class protocol.
Door het verbergen van interne representaties (er is alleen communicatie mogelijk via het protocol) kan een class veranderd worden zonder dat andere delen van de programmatuur, die instances van de class gebruiken, er weet van hebben en daardoor eveneens aangepast zouden moeten worden.
Messages worden niet naar classes gezonden maar naar objects die bij classes horen.

In figuur 2 is een schematische weergave van een class gegeven. Behalve de class zijn ook twee objecten, die instances zijn van de class, weergegeven. Door de introductie van de class 'verliest' een object zijn 'eigen' message dispatch table. Daarvoor in de plaats komt een verwijzing naar de class waartoe het object behoort.

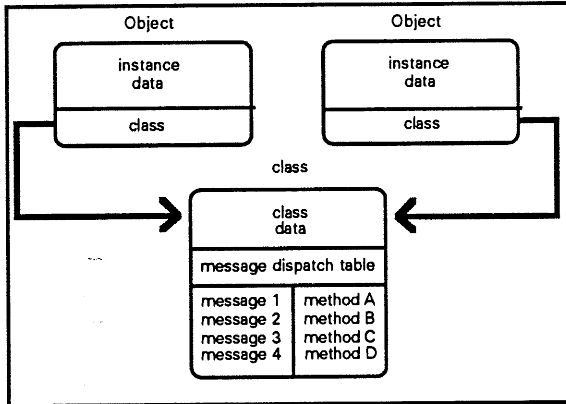


Figuur 2: Schematische weergave van een class

3.4 Class en instance variabelen

In traditionele talen zoals Pascal en C zijn datatypen beschikbaar. De programmeur kan dan met deze typen variabelen aanmaken en zelfs homogene en heterogene datatypen definiëren. Het type van een variabele bepaalt de verzameling van toegestane bewerkingen. Alhoewel er een sterke gelijkenis bestaat tussen de datatypen uit bijvoorbeeld de taal C en de classes van object georiënteerde talen (een object is een instance van een class net zoals een variabele in een procedurele taal een instance van een datatype is) is er toch een duidelijk verschil.
De class kan zelf datavelden bevatten, de zogenaamde class variabelen. Deze variabelen kunnen gezamenlijk worden gebruikt door alle objecten van de class. Om ze te onderscheiden van de class variabelen worden de datavelden in een object de instance variabelen genoemd. Deze instance variabelen kunnen alleen gebruikt worden door het object waartoe ze behoren. De meeste object georiënteerde talen bieden geen mogelijkheid voor de definitie van unieke instance variabelen. Een uitzondering hierop is de taal Object Logo. In deze taal is het mogelijk dat twee instances van dezelfde class verschillen vertonen.

Er is nog een tweede verschil tussen de traditionele datatypen en de classes. De datatypen zijn passief, de classes zijn actief. Het aanmaken en vernietigen van objecten (instances) zijn voorbeelden van door een class uit te voeren acties.
Figuur 3 geeft de locatie aan van instance en class variabelen in objects en classes.



Figuur 3: Schematische weergave van instance en class variabelen

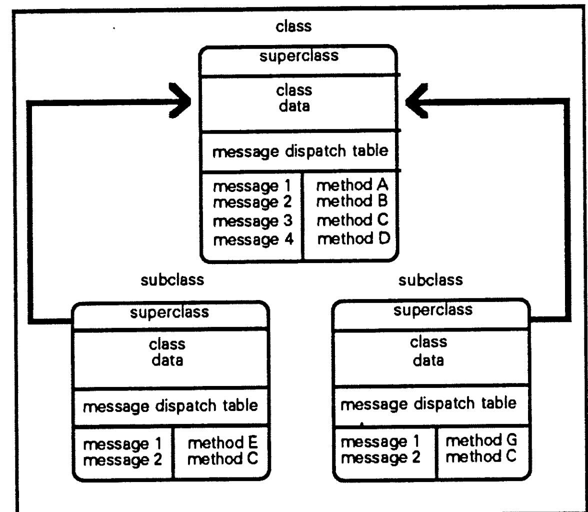
3.5 Inheritance
In de traditionele talen kunnen door een programmeur nieuwe typen gedefinieerd worden. Voor zo'n nieuw type moet de programmeur vervolgens een geheel nieuwe verzameling operaties definiëren. Dit terwijl het nieuwe type grote gelijkenis kan hebben met bestaande types en daarom ook gebruik zou kunnen maken van bestaande operaties.

Ook in object georiënteerd programmeren worden vaak nieuwe typen (classes) gedefinieerd. Het grote verschil is echter dat onderkend wordt dat verschillende typen objecten grote gelijkenissen kunnen hebben. Zo zullen bijvoorbeeld een factuur en een credit-nota veel karakteristieken gemeen hebben. Om van deze gemeenschappelijke kenmerken gebruik te kunnen maken kent het object georiënteerd programmeren het concept inheritance.

Onder inheritance wordt de mogelijkheid verstaan om een nieuwe class te definiëren als specialisatie van een bestaande class en zo in een aantal stappen verfijningen aan te brengen.
Uitgaande van een class, de superclass, wordt een nieuwe class gedefinieerd, de subclass, die in eerste instantie volkomen gelijk is aan de reeds bestaande class. De subclass erft alle eigenschappen (instance en class variabelen alsmede methods) van de superclass. Vervolgens wordt voor de nieuwe class aangegeven welke de verschillen zijn. Het aanmaken van een subclass wordt vaak aangeduid met subclassing.
De aldus ontstane hiërarchie van classes wordt weleens aangeduid met de term "generalisatie-specialisatie hiërarchie".

Nadat een subclass gedefinieerd is, kan de programmeur nieuwe variabelen en methods toevoegen en/of methods aanpassen.
Het grote voordeel van inheritance is, dat bij definitie van een nieuw type (een subclass) alleen de verschillen met bestaande typen (classes) gespecificeerd hoeven te worden. Een groot deel van de reeds bestaande code kan dus opnieuw worden gebruikt.
Het aanpassen van een method in een subclass geschiedt door het invoegen van de method van de superclass in een nieuw aangemaakte method. Zo kan de method van een subclass bestaan uit een aanroep naar de method van de superclass met daarop volgend de extra instructies die specifiek gelden voor de subclass.

In figuur 4 is een hiërarchie van classes weergegeven. Elke class bevat nu een verwijzing naar de bijbehorende superclass.



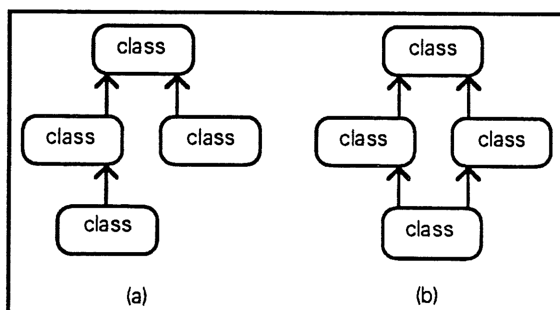
Figuur 4: Schematische weergave class-hiërarchie

In figuur 4 is duidelijk te zien dat de message dispatch tables van de subclasses zijn gewijzigd. De messages 1 en 2 kunnen in de subclasses zelf worden gevonden. Messages 3 en 4 moeten in de superclass worden gezocht. In de subclasses zijn voor message 1 verschillende methods opgegeven.

Het lezen van artikelen en boeken over object georiënteerd prgrammeren levert enige verwarring over het concept inheritance: is de aanwezigheid wel een eis voor object georiënteerd programmeren? Is het niet voldoende om alleen met objecten, classes en het zenden van messages te werken? Deze verwarring lijkt geïntroduceerd door de voorstanders van de taal ADA. Deze taal biedt namelijk wel objecten en classes maar geen inheritance.
De voordelen rechtvaardigen het eisen van de aanwezigheid van inheritance als één van de concepten van het object georiënteerd programmeren.

3.6 Soorten inheritance
Veelal wordt een aantal classes gedefinieerd met generieke methods. Nieuwe classes, gedefinieerd als subclasses van zo'n generieke class, kunnen gebruik maken van de bijbehorende verzameling algemene methods. Sommige object georiënteerde talen, waaronder Smalltalk-80, kennen abstract classes.
Een abstract class is een generieke class waarvoor geen objecten aangemaakt kunnen worden. Zo'n abstract class kan wel gebruikt worden bij de definitie van subclasses. Aan de top van de hiërarchie van classes staat de abstract class object. Deze class bevat onder andere methods voor het vergelijken van objecten en het retourneren van de class van een object. In de volgende paragraaf zal op deze speciale class worden ingegaan.

Een superclass kan worden gebruikt voor de definitie van meerdere subclasses. In de meeste object georiënteerde talen is het slechts mogelijk om één superclass te gebruiken bij de definitie van een subclass, hetgeen aangeduid wordt met de term simple hierarchical inheritance. Uitzonderingen hierop zijn Smalltalk-80 alsmede de laatste versie van de taal C++. Deze talen bieden multiple inheritance. Een subclass met meerdere superclasses bevat de kenmerken van alle superclasses. Het voordeel van multiple inheritance is dat er meer mogelijkheden geboden worden om programmadelen op meerdere plaatsen te gebruiken. Bijvoorbeeld: de class speelgoed auto heeft de classes speelgoed en auto als superclasses. Een subclass is dan sneller en efficiënter gedefinieerd hetgeen de ontwikkeltijd verkort.
Er zijn echter ook een tweetal nadelen. Allereerst is er sprake van een toenemende complexiteit in zowel implementatie als gebruik. Daarnaast kunnen er conflicten optreden: indien meerdere superclasses dezelfde namen gebruiken voor bijvoorbeeld class variabelen, kan bij toekenning van een waarde aan zo'n variabele niet bepaald worden welke class variabele bedoeld wordt.



Figuur 5: Schematische weergave van simple hierarchical (a) en multiple inheritance (b)

In figuur 5 is het essentiële verschil tussen simple hierarchical inheritance en multiple inheritance aangegeven. Bij simple hierarchical inheritance zijn de classes in een boomstructuur geordend. Bij multiple inheritance worden de classes volgens een a-cyclisch netwerk geordend.

3.7 Metaclass
In de voorafgaande paragrafen is een overzicht gegeven van de belangrijkste concepten van het object georiënteerd programmeren. Een aantal vragen is echter nog onbeantwoord. Hoe weet een class hoe een object eruit moet zien? Is er een verschil tussen een class, een factory en een message dispatch table? Wat is het verschil tussen een instance variabele en een object? Wat is het wezenlijke verschil tussen een class en een object? Indirect gerelateerd is de vraag: hoe zijn de methods te plaatsèn in de gegeven figuren? Wat gebeurt er in een method?

Voor de liefhebbers zullen de bovenstaande vragen in het kort beantwoord worden.
Het antwoord op de eerste vraag is simpel: de class bevat een omschrijving, een blauwdruk, van de aan te maken objecten.
Om een antwoord te geven op de volgende drie vragen is het belangrijk te kijken naar de implementatie van classes en objecten. Het blijkt dan dat classes op dezelfde wijze geïmplementeerd worden als objecten, met andere woorden: het zijn objecten! En, en nu wordt het gecompliceerd, de classes zijn instances van de class metaclass. De class metaclass is zelf geïmplementeerd als een object en is een instance van de class object. De class object is de top van de hiërarchie van classes. Voor deze class zijn geen verdere referenties nodig, de implementatie van de taal verzorgt een correcte afhandeling.
Om terug te komen op de gestelde vragen: een class is een object met als speciale eigenschap dat het andere objecten definieert, hetgeen de bijnaam factory verklaart. Een message dispatch table is een onderdeel van de class evenals de blauwdruk ten behoeve van het aanmaken van instances.
Een instance variabele is zelf ook weer een object. Het 'type', de class waartoe een instance variabele behoort, wordt impliciet bepaald.

De laatste vragen laten zich nu iets eenvoudiger beantwoorden. Ook methods kunnen worden geïmplementeerd als objecten, Deze objecten beschrijven de uit te voeren operaties. Een interpreter leest dan de data van een method en weet zo, welke operaties uitgevoerd moeten worden. Een method is in dit model een instance van class method. De message dispatch table verwijst naar een of meerdere instances van class method. Verschillende message dispatch tables kunnen wijzen naar een en hetzelfde instance, hetgeen het hergebruik van code verder bevordert.

Het zenden van een message is een veel voorkomende operatie. De meeste methods

doen in feite niets anders: messages worden gezonden naar tijdelijk aangemaakte objecten of naar bestaande objecten. De indirect aangeroepen methods zullen vaak op hun beurt weer messages gaan zenden. Gelukkig komt er ook een einde aan de 'lawine' van messages. Een aantal operaties in een method zendt geen message en worden de primitieve operaties genoemd.

Tot slot: de antwoorden op de gestelde vragen zijn niet helemaal volledig. Zo zijn zowel de class Class als de factory objects buiten beschouwing gelaten om taal-afhankelijkheden te vermijden.

4 Geschiedenis

Alvorens de karakteristieken en voor- en nadelen te behandelen, wordt een kort overzicht van de nog jonge geschiedenis van het object georiënteerd programmeren gegeven.

4.1 Aanzet

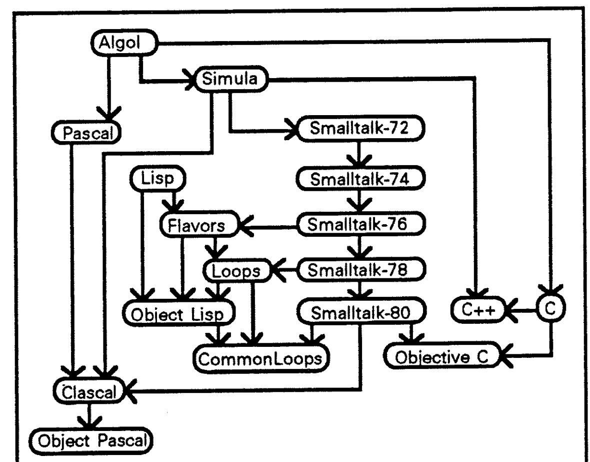
Voortdurend worden nieuwe programmeertalen en concepten uitgedacht. In de jaren vijftig werden programmeertalen als COBOL en Fortran ontwikkeld. In de jaren zestig en zeventig kwamen de zogenoemde 'blok georiënteerde' talen als Algol, Pascal en C. Ook zijn verschillende probleemgerichte talen, ontwikkeld om specifieke problemen te lijf te gaan, tot volle wasdom gekomen. Een bekende probleemgerichte taal is SIMULA. De geschiedenis van het object georiënteerd programmeren begint eind jaren zestig met de ontwikkeling van deze taal, die gebaseerd is op Algol. SIMULA werd ontwikkeld om beter en sneller simulatie-problemen op te lossen (zie [BIRT73]). In deze taal staat het object echter niet centraal: het is mogelijk instance variabelen te wijzigen buiten het class protocol om.

4.2 Smalltalk-80

De 'echte' geboorte van het object vond plaats in het Xerox Palo Alto Research Center (PARC) onder leiding van Alan Kay. Doel van de ontwikkelingen bij PARC was, het ontwerpen van een programmeertaal en ontwikkelomgeving die steun bieden bij de produktie van interactieve programma's voor microcomputers.
De ontwikkelingen resulteerden via Smalltalk-72, Smalltalk-74, Smalltalk-76 en Smalltalk-78 in Smalltalk-80. Smalltalk-80 was de eerste volledig object georiënteerde taal. In deze taal is praktisch alles geïmplementeerd als object. De interpreter en de debugger zijn twee voorbeelden van gespecialiseerde objecten.
De Smalltalk-80 user interface is het klassieke voorbeeld van een 'direct manipulation' gebruikersinterface; een interface waarin de gebruiker de controle heeft over een grafisch model van een systeem. De Apple Macintosh user interface is afgeleid van de Smalltalk-80 user interface. Direct manipulation user interfaces zijn eenvoudiger in gebruik dan conventionele user interfaces. Ze zijn echter wel moeilijker te programmeren. Smalltalk-80 had tot doel de implementatie van direct manipulation interfaces te vereenvoudigen. De toegankelijkheid van de taal laat echter te wensen over. Voor een beschrijving van Smalltalk-80 en de taal worden [GOLD83], [GOLD84] en [DIGI86] aanbevolen.

4.3 Hybride talen

De invloed van Smalltalk-80 is vooral gelegen in het feit dat in de jaren tachtig verschillende traditionele talen werden uitgebreid om het object georiënteerd programmeren te ondersteunen, zie hiervoor figuur 6.



Figuur 6 : Ontwikkeling van talen

Opmerking: in figuur 6 is geen chronologische informatie opgenomen!

C werd uitgebreid tot Objective C en C++ (zie [STRO86]), Pascal werd uitgebreid tot Objective Pascal en LISP werd uitgebreid tot Object LISP. Ook aan andere talen werden of worden objects toegevoegd.
De traditionele talen die uitgebreid zijn met mogelijkheden voor object georiënteerd programmeren worden hybride genoemd. Dit in tegenstelling tot Smalltalk-80, dat een pure object georiënteerde taal is.

Alhoewel object georiënteerd programmeren veelal gelijkgesteld wordt met het programmeren in deze talen, is er toch een duidelijk onderscheid. De talen moedigen het object georiënteerd programmeren aan of bieden het zelfs als enige mogelijkheid. Het programmeren met 'objects' kan echter in principe met elke taal, het is een discipline.

4.4 Hergebruik van software

Het ontwikkelen van een programma is mede een moeizaam en ambachtelijk karwei omdat er geen of (te) weinig pogingen worden ondernomen om delen van eerder ontwikkelde soft-

ware te gebruiken. Bij het ontwikkelen van een programma wordt meestal begonnen met een schone lei! Toch zal veel programmeurs het gevoel van 'déjà vu' niet vreemd zijn: programma's vertonen immers grote gelijkenissen (zie [JONE84]).

Het opnieuw gebruiken van delen van eerder gemaakte programma's kan het produceren en aanpassen van programma's vereenvoudigen en versnellen. Helaas is dit hergebruik niet eenvoudig. De te gebruiken 'oude' code zal zelden precies passen in een nieuwe applicatie. Aanpassingen zijn dan noodzakelijk.
Traditionele talen bieden helaas onvoldoende mogelijkheden om de aanpassingen door te voeren: de te gebruiken stukken code, aangeboden als subroutines, zijn atomische eenheden. Dit houdt in dat het wijzigen van te gebruiken subroutines niet mogelijk is en de programmeur vaak gedwongen wordt rond de aanroep naar de subroutine extra code te plaatsen. Deze tekortkoming leidt ertoe dat de programmeur gebruik maakt van door ervaring verkregen 'conceptuele templates'. Hij voert deze in de computer en maakt gelijktijdig de noodzakelijke kleine wijzigingen. Het gevoel van 'déjà vu' is dan ook niet verwonderlijk.

De bij de ontwikkeling van het object georiënteerd programmeren opgedane kennis en ervaring, werden al snel benut bij pogingen om het opnieuw gebruiken van bestaande software te stimuleren (zie [MEYE87]). Hierdoor hoopte men te komen tot een hogere produktiviteit bij het ontwikkelen van toepassingen. De 'pogingen' resulteerden onder andere in de ontwikkeling van 'object georiënteerde bases'.
Zo'n basis bestaat uit een set samenhangende routines en fungeert als een kapstok waaraan de ontwikkelaar nieuwe routines hangt. Het maken van een applicatie is dan 'niets meer en niets minder' dan het invullen en aanpassen van de basis; bestaande routines worden aangepast en nieuwe routines worden toegevoegd. Afhankelijk van de te bouwen programma's kan in de basis al enige nuttige functionaliteit worden aangebracht.

De object georiënteerde bases blijken zeer krachtig en flexibel! Deze bases ondersteunen de stelling dat voor een efficiënt hergebruik van software, aanpassingen veelal gewenst zijn. Op het flexibele karakter van object georiënteerde bases zal nog worden teruggekomen.

4.5 Inschakelen gebruikers

Naast de eerder genoemde ontwikkelingen wordt ook voortdurend getracht, de gebruikers meer te betrekken bij het oplossen van hun eigen problemen.
Zo is er een groot scala aan pakketten ontwikkeld die het een gebruiker van computers maakt zelf programma's te schrijven.
Een van de laatste ontwikkelingen op dit gebied is HyperCard, ontwikkeld door de microcomputerleverancier Apple. Met HyperCard doet het object georiënteerd programmeren zijn intrede in de wereld van de eindgebruikers.
Een andere poging op dit gebied, de ontwikkeling van Smalltalk-80, leidde niet tot het gewenste resultaat.

4.6 Databases

Nieuwe ontwikkelingen zijn gaande op het gebied van de database management systemen. Na de hiërarchische databases van de jaren zestig en de netwerk/codasyl databases in de jaren zeventig, werden in de jaren tachtig de relationele databases ontwikkeld.
Hieraan kleven nadelen: complexe vragen vergen een complexe structuur in de vraagstelling. Voorts zijn de resultaten van een operatie verzamelingen terwijl een applicatie normaliter werkt met afzonderlijke records.
Het is dan ook niet verwonderlijk dat veel universiteiten en bedrijven intensieve research plegen om tot betere database management systemen te komen.
Een aantal nieuwe ontwikkelingen op het database management gebied maakt gebruik van de concepten van het object georiënteerd programmeren teneinde het mogelijk te maken, niet alleen de 'ruwe' data op te slaan maar ook de aan deze data toegekende betekenis. Deze ontwikkelingen zouden er in de toekomst toe kunnen leiden dat het bewaken van de consistentie meer en meer verplaatst wordt van de applicatie naar het database management systeem.

5 Karakteristieken

Mede door de concepten heeft het object georienteerd programmeren een aantal karakteristieken die het vermelden waard zijn. Deze zijn de basis van enkele van de voordelen van deze programmeertechniek.

5.1 Abstractie

Bij het maken van een programma wordt een abstractie van de werkelijkheid gecreëerd (zie [FAIR85]). In veel gevallen bepaalt de te manipuleren data de grenzen van de door een programma te ondernemen acties. Derhalve is het veelal van groot nut om in eerste instantie niet zozeer te kijken naar de functies van een programma maar naar de te manipuleren data.
Data-abstractie is één techniek waarbij niet gekeken wordt hoe de data eruit ziet of hoe het genoemd wordt, maar waarbij de nadruk ligt op de relevantie van de data. Dit biedt de ontwerper van een programma de mogelijkheid om zich te concentreren op de essentie van een probleem, zonder dat de blik vertroebeld wordt door details van representatie en manipulatie. Bij data-ab-

stractie wordt het geheel van data gegroepeerd in afzonderlijke delen. Twee data-elementen zijn van hetzelfde type als zij dezelfde algemene vorm en inhoud hebben. Vervolgens wordt bepaald hoe de manipulatie (creëren, raadplegen, veranderen en verwijderen) van een data-type plaatsvindt. Objects kunnen in dit perspectief gezien worden als 'machines' die elk een stukje data van een bepaald type kunnen manipuleren. Elk object bevat twee typen informatie: de status waarin het stukje data zich bevindt en een verzameling operaties (de object module) die de data in een andere status brengen.

5.2 Dynamic binding

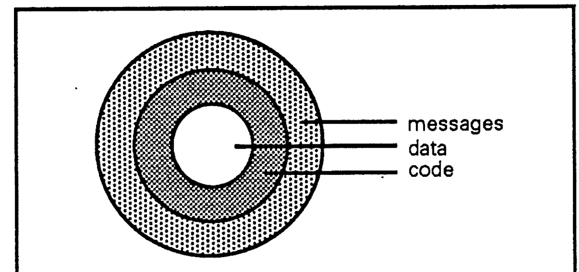
Binding, het 'toekennen van een type' aan een variabele, geschiedt in veel object georiënteerde talen dynamisch. Bij static binding, hetgeen gebruikelijk is in procedurele talen, vindt het toekennen van het type eenmaal, tijdens compilatie, plaats. Bij dynamic binding vindt dit toekennen tijdens executie van het programma plaats. Dynamic binding biedt een grote flexibiliteit bij creatie en manipulatie van datastructuren. Een nadeel is onder andere de vaak noodzakelijke dynamic type checking: tijdens executie moet bepaald worden wat het type van de variabele is. Een ander nadeel is de leesbaarheid c.q. het bewijzen van de correctheid van een programma. In sommige object georiënteerde talen kan het type van een object, binnen zekere grenzen, gewijzigd worden. Een klein voorbeeld: het statement HetObject = 5 geeft aan dat HetObject een instance is van de class 'IntegerClass'. Het vervolgens uitvoeren van het statement HetObject = 'Een paar letters' maakt HetObject een instance van de class 'StringClass'.
Dynamic binding heeft tot gevolg dat het tweemaal zenden van dezelfde message naar een object, niet in beide gevallen tot dezelfde acties hoeft te leiden. Het principe en de voor- en nadelen van dynamic binding worden uitgebreid beschreven in [GHEZ82].
Voor de stimulering van het hergebruik van software is dynamic binding essentieel: software wordt gebouwd en getest door een leverancier die het vervolgens in binaire vorm aan gebruikers ter beschikking stelt. De in traditionele talen noodzakelijke controles op het type van een variabele kunnen achterwege blijven. Dit is mede mogelijk door de aanwezigheid van polymorfisme.

5.3 Polymorfisme

Bij object georiënteerd programmeren kan sprake zijn van polymorfisme. Dit is een flexibele en geheel eigen variant op wat bij traditionele talen operator-overloading wordt genoemd: verschillende operatoren hebben dezelfde naam, maar de werking van deze operatoren is afhankelijk van het type data dat gemanipuleerd wordt. Een eenvoudige vorm van operator-overloading is het gebruik van de optel-operator '+' voor zowel gehele als reële getallen. Het is vrij eenvoudig voor te stellen dat de optel-operator ook gedefinieerd is voor concatenatie van strings. Overloading is dus het toekennen van meer dan één betekenis aan één naam.
Object georiënteerd programmeren biedt de mogelijkheid om voor elke class telkens weer dezelfde message te gebruiken waarbij de method telkens anders is. Dit heeft als resultaat dat deze message een verschillende uitwerking heeft bij verschillende typen objecten.

5.4 Samenvatting

Het object georiënteerd programmeren is niet zozeer een codeertechniek maar meer een verpakkingstechniek. De bijdrage van de geboden abstractie bestaat uit het plaatsen van een muur van code rond elk stukje data.



Figuur 7: Afscherming van data door code en messages

Dynamic binding maakt het mogelijk stukken code van verschillende 'leveranciers' te integreren. De code bevat geen veronderstellingen over het type data dat gemanipuleerd moet worden. De integratie wordt mede mogelijk gemaakt door het achter messages verstoppen van de operaties. Verschillende typen data kunnen dezelfde messages ontvangen als gevolg van de aanwezigheid van polymorfisme.
Zoals in figuur 7 is weergegeven heeft elk stukje data derhalve twee muren: die van code en de muur van messages.

6 Kracht en zwakte

Object georiënteerd programmeren heeft een aantal voordelen. Er kleeft echter ook een aantal nadelen aan deze nog jonge programmeertechniek. Een aantal significante voor- en nadelen wordt in het navolgende gegeven.

6.1 Gebruik

Er is een duidelijk onderscheid tussen de traditionele talen en de object georiënteerde talen. Object georiënteerde talen zijn die talen waarin het zenden van een message naar een object de fundamentele actie is. Dit in tegenstelling tot traditionele talen die alleen het data procedure paradigma ondersteunen: actieve procedures werken op passieve data die hun wordt aangeboden.

Voor ervaren programmeurs in traditionele, procedurele, talen is het vaak moeilijk wennen aan de object georiënteerde concepten. Bij de overstap naar een object georiënteerde taal is het niet 'zomaar een nieuwe syntax' die geleerd moet worden, maar het zich eigen maken van een nieuwe denktrant. Toepassingen moeten vertaald worden in termen van objects. Deze modellering staat op een hoger, abstracter niveau. Het programma wordt gedefinieerd door de data abstractie. Dit in tegenstelling tot de meer gangbare procedurele abstractie.

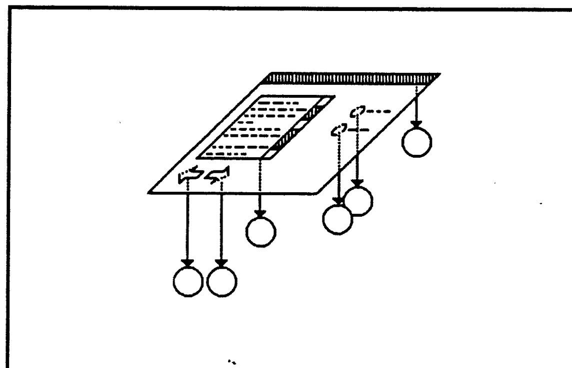
Daarnaast moet een programmeur een goede kennis hebben van de class library hetgeen ook enige tijd vereist. De class library bevat een aantal gedefinieerde classes. Een method is in dit kader te zien als onderdeel van een class.

De door object georiënteerde concepten geboden abstractie biedt significante voordelen (zie [GORL87]). De eventuele complexiteit van een class is onzichtbaar voor de programmeur. Hierdoor wordt het eenvoudiger om instances van een class te gebruiken en om fouten te traceren: indien een instance variabele een foute waarde bevat, is het aantal 'verdachte' procedures beperkt. De 'gedwongen' abstractie maakt onderhoud, noodzakelijk door het ontdekken van (kleine) fouten en nieuwe wensen van gebruikers, eenvoudiger. De kans dat door veranderingen fouten worden geïntroduceerd is relatief klein (zie [DIED87]).
Zowel een groot voordeel als een gevaar van de indirecte aanroep van operaties, die geboden wordt door het zenden van messages, is gelegen in het feit dat een message een operatie kan initiëren die niet overeen hoeft te komen met hetgeen verwacht wordt. Bij een slechte keuze van de messages kan de leesbaarheid van een programma sterk afnemen. In wezen is hier echter geen onderscheid met de keuze van namen voor subroutines in traditionele talen.

Object georiënteerd programmeren echter is meer dan het verstoppen van data en operaties in objecten. Het concept inheritance levert een essentiële kracht. Voor het efficiënt toepassen van inheritance lijkt een passende taal noodzakelijk. Alhoewel, zoals al eerder vermeld, het object georiënteerd programmeren in elke taal mogelijk is, is er dan wel sprake van een serieuze afzwakking van de voordelen (zie [MEYE87]).

6.2 Toepassingen

Object georiënteerd programmeren is met name geschikt voor sterk interactieve gebruikersvriendelijke interfaces, zoals die van de Apple Macintosh (zie [SCHM86]). Dit omdat het niveau van interactie tussen de gebruiker en de applicatie direct gespiegeld kan worden in de interactie tussen objecten die de basis van het object georiënteerd systeem vormen. In figuur 8 is aangegeven hoe objects, weergegeven als bollen, gebruikt kunnen worden bij de controle van een grafische gebruikers-interface.



Figuur 8: Gebruik van objects in een interface

Elk element van de interface, weergegeven in figuur 8, is gekoppeld met een object. Acties van de gebruiker worden door het zenden van een message aan een gerelateerd object gemeld. Het werken met objecten wordt door velen als meer natuurlijk ervaren dan het werken met de structuren van traditionele programmeertalen.

De voordelen die deze manier van programmeren biedt zijn ook bij Microsoft, de ontwikkelaar van onder andere MS-DOS en OS/2, bekend. Microsoft overweegt het toevoegen van object georiënteerde aspecten in de geboden produkten (zie [BYT88]). Bill Gates, de grote man van Microsoft hierover: "This is necessary in part because of the difficulty programmers are having, and will continue to have, writing sophisticated graphics-based applications....Because we've been using classic programming languages, it is particularly hard to write these applications.......... What we are going to do is extend our classic languages like BASIC and C to be object oriented in the same strong sense that Smalltalk has been...."
Over de ontwikkeling van nieuwe talen zegt Bill Gates: "I question anyone's strategy of introducing new computer languages after 1988 and trying to get people to write applications in them. I really think that our current languages with object management extensions can permit us to deal with the graphical world....."

De jaren vijftig, zestig en zeventig werden overheerst door data processing: data werd getransformeerd tot informatie. In de jaren tachtig is door het vakgebied van de Artificial Intelligence de knowledge processing geïntroduceerd waarbij de nadruk ligt op kennis-representatie en kennis-interpretatie. Tal van expert-systemen zijn als gevolg hiervan ontwikkeld.
Een expert system is een programma dat de kennis bevat van een materie-deskundige. De vastgelegde kennis kan zowel theoretisch als praktisch van aard zijn. Naast het vastleggen van

kennis moet ook de interpretatie van de kennis ondersteund worden. Met deze kennis en de vastgelegde wijze van interpretatie, is de gebruiker van zo'n systeem in staat complexe problemen op te lossen.
Vaak wordt een relatie verondersteld tussen object georiënteerd programmeren en Artificial Intelligence. Alhoewel het object georiënteerd programmeren een meer algemene techniek is, bieden de aanwezige concepten een goede basis voor het vastleggen en interpreteren van kennis.

6.3 Efficiëntie

Het grote voordeel van object georiënteerd programmeren is het concept inheritance. Hierdoor wordt het opnieuw gebruiken van reeds bestaande code gestimuleerd. Daarnaast wordt de programmeur aangemoedigd efficiënte datastructuren te ontwikkelen. Ook bestaat de mogelijkheid een produkt op te splitsen in totaal afgeschermde delen, die afzonderlijk gecodeerd kunnen worden.

De code, het programma, bestaat in een object georiënteerde taal vaak uit een grote collectie van kleine methods. In Smalltalk-80 is de code verspreid aanwezig in het systeem, hetgeen het plegen van grote aanpassingen en het bepalen van de functies van het programma kan bemoeilijken. Daar staat tegenover dat de referenties naar methods indirect zijn (via messages), hetgeen de aanpasbaarheid van een programma positief beïnvloedt.
Programma's, geschreven in een taal die dynamic binding biedt, moeten uitgebreider getest worden. De kans bestaat dat een object een message ontvangt die op dat moment niet begrepen wordt, hetgeen resulteert in een foutmelding of erger.
Het zenden van een message naar een object kost meer tijd dan het rechtstreeks aanroepen van een procedure. Daarnaast moet rekening worden gehouden met het feit dat het zoeken van de message in de method dispatch table ook voor 'enige' vertraging zorgt. Hier staat tegenover dat soms meer efficiënte code geschreven kan worden en dat met de huidige generatie snelle machines dit nadeel niet altijd meer als storend ervaren hoeft te worden.
Bij research naar de efficiëntie van Smalltalk-80 bleek dat een in C geschreven programma ongeveer tweemaal zo snel was in vergelijking met een in Smalltalk-80 geschreven programma (zie [DIED87]).

6.4 Talen

De afgelopen jaren is er een aantal programmeertalen ontwikkeld met het object als basiselement.
De implementatie op een machine van object georiënteerde talen wordt vaak als meer complex gezien, omdat de onderliggende machine niet in de taal wordt gespiegeld. Indien een object georiënteerde taal dynamic binding biedt is er sprake van implementatie via een interpreter. Deze interpreter kan zelf in een portable taal geschreven worden hetgeen de implementatie van de taal op een nieuwe machine vereenvoudigt.
Het toevoegen van objects aan een traditionele taal kan bewerkstelligd worden door de implementatie van een extra vertaler, die de 'taal met objects' omzet in de 'taal zonder objects'. Een bekend voorbeeld van deze methode is de taal C++. Deze wijze van implementatie vereenvoudigt eveneens de implementatie van de taal. Een nadeel is de extra vertaalslag die natuurlijk tijd kost.

Sommige object georiënteerde talen zijn geïsoleerde talen. Smalltalk-80 is daar een voorbeeld van; het is een compleet ontwikkelsysteem waarin het niet mogelijk is om in andere talen geschreven bibliotheken te gebruiken.
Bestaande bibliotheken voor input/output, grafische manipulatie, communicatie e.d. zijn niet bruikbaar en moeten in Smalltalk-80 herschreven worden.

Hybride talen zoals C++ hebben, op dit moment, geen class library. De voor een applicatie benodigde set van classes moet van de grond af opgebouwd worden. Het gevaar bestaat dan dat het procedureel programmeren de overhand blijft houden en dat de voordelen van object georiënteerd programmeren niet optimaal benut worden. Op verschillende fronten wordt momenteel gewerkt aan een class library voor C++ (zie bijvoorbeeld [GORL87]).
Indien een set van classes geleverd wordt, heeft de ontwikkelaar een schat aan programmatuur ter beschikking voordat hij daadwerkelijk begint.

7 Conclusies

De invloed van het object georiënteerd programmeren is reeds duidelijk zichtbaar; objects worden aan steeds meer traditionele talen toegevoegd. De 'ambities' van het object georiënteerd programmeren gaan verder. Het toevoegen van de concepten aan de bestaande relationele database technologie is een goed voorbeeld. De kans is redelijk aanwezig dat de concepten ook op andere gebieden, bijvoorbeeld netwerk-technologie of programmatuur voor multi-processor systemen, toegepast zullen worden. Bij het gebruik van bijvoorbeeld object georiënteerde databases is het programmeren in een passende, natuurlijk object georiënteerde, taal een voor de hand liggende keuze.

Het object georiënteerd programmeren kent een aantal voordelen. Inheritance biedt grote voordelen indien sneller betere programma's ontwikkeld moeten worden. Het hergebruik van software wordt gestimuleerd. De isolatie van

methods en objectbeschrijvingen (alleen de bij de class behorende methods weten welke class en instance variabelen gedefinieerd zijn) komt de onderhoudbaarheid ten goede.

Met name op Amerikaanse universiteiten wordt veel research gepleegd naar object georiënteerd programmeren. Toekomstig universitair geschoolden hebben kennis van en ervaring met het object georiënteerd programmeren. Hierdoor zou eenzelfde stimulans kunnen ontstaan als voor het besturingssysteem Unix. Via de universiteiten is dit besturingssysteem geïntroduceerd in het bedrijfsleven.

Het is te verwachten dat de invloed van het object georiënteerd programmeren in de komende jaren zal toenemen.

8 Personalia

Laurent Gielen is sinds juli 1985 als Software Engineer werkzaam bij KPMG Klynveld EDP Audit. Hij is in 1985 afgestudeerd aan de HIO te Eindhoven. Zijn interessegebieden zijn User Interfaces, programmeertalen en -technieken en artificial intelligence.

8 Literatuur

[BIRT73] Birtwistle, GM, e.a.: Simula Begin. Van Nostrand Reinhold, 1973.

[BYT88] Microbytes, Microsoft plans Object Oriented Enhancements, Byte, april 1988.

[DIED87] Diederich, J, Milton, J: Experimental prototyping in Smalltalk. IEEE Software, maart 1987.

[DIGI86] Smalltalk/V, Tutorial and Programming handbook, Digitalk inc, 1986.

[FAIR85] Fairley, R: Software engineering concepts. McGraw-Hill Book Company, 1985.

[GHEZ82] Ghezzi, C, Jazayeri, M: Programming Language Concepts. John Wiley & Sons, Inc, 1982.

[GOLD83] Goldberg, A, Robson, D: Smalltalk-80, The Language and its Implementation. Addison-Wesley Publishing Company, 1983.

[GOLD84] Goldberg, A: Smalltalk-80, The Interactive Programming Environment. Addison-Wesley Publishing Company, 1984.

[GORL87] Gorlen, K: An Object-Oriented Class Library for C++ Programs. Software practice and experience, volume 17, december 1987.

[JONE84] Jones, T: Reusability in programming: A Survey of the State of the Art. IEEE Software, september 1984.

[MEYE87] Meyer, B: Reusability: The case for Object-Oriented Design. Interactive Software Engineering, IEEE Software, maart 1987.

[PASC86] Pascoe, G: Elements of Object-Oriented programming. Byte, augustus 1986.

[SCHM86] Schmucker, K: Object–Oriented programming for the Macintosh. Hayden Book Company, 1986.

[STRO86] Stroustrup, B: The C++ programming language. Addison-Wesley Publishing Company, 1986.

HyperCard

door: J. Schalk

1 Inleiding

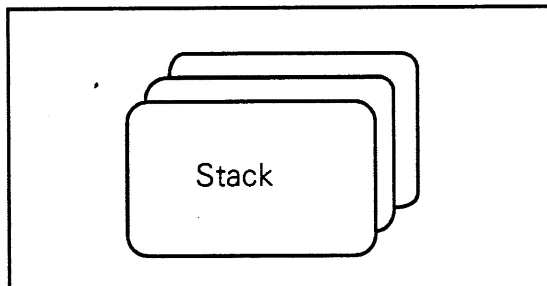
"HyperCard is a software erector set", Bill Atkinson.

Bovenstaand citaat van de maker van HyperCard (HC) geeft aan hoe men dit pakket het best kan beoordelen: het is een constructieset voor Apple Macintosh-applicaties. Het met iedere Macintosh gratis meegeleverde pakket geeft aan de gebruiker de mogelijkheid om zijn eigen toepassingen te ontwikkelen. Deze toepassingen, 'stacks' in HC-terminologie, kunnen liggen op het gebied van bestandsbeheer. Ook kan men HC gebruiken als gebruikersvriendelijke interface voor bijvoorbeeld een LaserDisk of CD-ROM-speler.

In dit artikel wordt gepoogd een indruk te geven van de mogelijkheden van HC. Hierbij wordt extra aandacht besteed aan voor de accountant belangrijke gebieden zoals protectie van HC-applicaties.
Het artikel dient niet als tutorial. Daarvoor wordt verwezen naar de in de literatuuropgave genoemde manuals, boeken en HyperCard stacks.

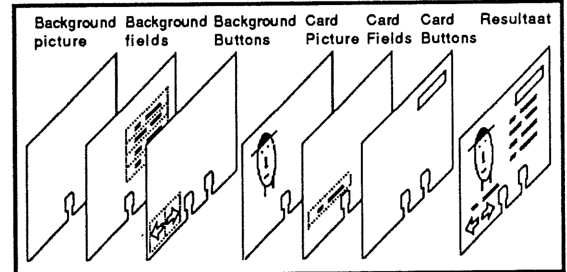
2 Structuur van een HyperCard-applicatie

Iedere HC-applicatie bestaat uit een stapel kaarten (stack) die elk weer uit twee lagen bestaat: de achtergrond (background) en de kaart (card) laag. Ieder van deze twee lagen bestaat weer uit een grafische voorstelling, de picture, en uit twee sets van objecten, buttons en fields. De buttons bepalen over het algemeen het gedrag van de stack (dus de functionaliteit ervan), en de fields representeren de inhoud van de stack. In database terminologie zal men per record één card hebben, terwijl een veld binnen dat record door een field wordt gerepresenteerd.



Figuur 1: HyperCard stack

De background (d.w.z. de background picture, fields en buttons) wordt gedeeld door alle kaarten in de stack. Een voorbeeld is een button die de mogelijkheid geeft om een kaart uit te printen. Deze functionaliteit is voor alle kaarten gewenst en daarom zal het als background button worden geïmplementeerd.
De card picture, fields en buttons zijn per kaart verschillend. Een per kaart verschillende card picture is te gebruiken om een bestand van plaatjes op te bouwen.

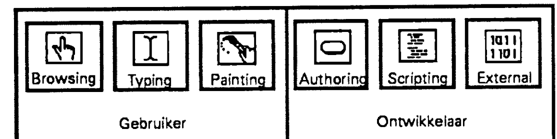


Figuur 2: Opbouw van een stack

In werkelijkheid kan de structuur van een HC stack ingewikkelder zijn, omdat men in een stack meerdere backgrounds kan hebben. De kaarten in de stack worden dan gepartitioneerd over deze backgrounds, d.w.z. een deel der kaarten heeft telkens dezelfde background (met bijbehorende background picture, buttons en fields). Deze complicatie is echter niet van wezenlijk belang. Men kan zo'n stack altijd opdelen in een aantal afzonderlijke stacks die telkens een background hebben.

3 De zes gebruikersniveaus van HyperCard

Een van de eigenschappen van HC is het feit dat er zes gebruikersniveaus zijn te onderscheiden, ieder met hun eigen mogelijkheden en beperkingen. De zes niveaus kan men onderverdelen in twee groepen van elk drie. Kenmerkend voor de eerste groep is dat de gebruiker van HC als eindgebruiker functioneert: hij/zij gebruikt een door anderen ontwikkelde stack. In de tweede groep functioneert de HC-gebruiker als stack-ontwikkelaar:



Figuur 3: Gebruikersniveaus

3.1 Browsing

Dit laagste en eenvoudigste niveau van HC geeft een eindgebruiker de mogelijkheid om een reeds ontwikkelde stack te bekijken en er kaarten van af te drukken. Er kunnen echter geen kaarten veranderd, toegevoegd of verwijderd worden. Deze vorm is uitermate geschikt voor het verspreiden van bijvoorbeeld een catalogus.

3.2 Typing
In dit niveau krijgt de eindgebruiker ook de mogelijkheid om de (tekstuele) inhoud van een kaart te wijzigen en om kaarten te verwijderen en toe te voegen. Zeer geschikt voor bijhouden van bijvoorbeeld een adressenbestand.

3.3 Painting
Dit niveau geeft de gebruiker toegang tot de tekenmogelijkheden van HC. Hierdoor kan het aanzien van een kaart (d.w.z. de picture van de background en de card) worden veranderd. De tekenmogelijkheden van HC zijn zeer uitgebreid. Immers, Bill Atkinson heeft ook MacPaint ontwikkeld, zodat in HC alle mogelijkheden die hij in MacPaint had willen hebben zijn toegevoegd. Een tip voor mensen die een tekenpakket zouden willen kopen is dan ook om eerst eens naar de mogelijkheden van het gratis HyperCard-pakket te kijken. Het grootste nadeel is dat tekeningen slechts de grootte van een Mac SE scherm kunnen hebben.

3.4 Authoring
Het authoring niveau is het eerste niveau waarop de gebruiker een nieuwe stack kan gaan ontwikkelen. In de eerste drie niveaus konden er alleen bestaande stacks worden bekeken en/of veranderd, maar dit niveau geeft ook de mogelijkheid om de functionaliteit van verschillende, reeds bestaande, stacks te combineren. Dit combineren (d.w.z. het hergebruiken van functionaliteit) is een van de sterkste punten van HC. Door een slimme combinatie van buttons en fields uit een aantal stacks kan men op intuïtieve manier een nieuwe stack creëren.

3.5 Scripting
In het scripting niveau gaat men het gedrag van een object in een HC-stack beïnvloeden. Dit gedrag wordt bepaald door een programma geschreven in de bij HC behorende programmeertaal HyperTalk. Dit programma, script genaamd, wordt tijdens executie door de HC-applicatie geïnterpreteerd.

3.6 External
Dit laatste en hoogste niveau is eigenlijk een uitbreiding van het scripting niveau. Het geeft aan dat de stack-ontwikkelaar de mogelijkheden van HC uit gaat breiden met behulp van in een conventionele programmeertaal (Pascal of C) geschreven functies (XFCN's in HC-terminologie) en commando's (XCMD's). Het is juist deze mogelijkheid die de ware kracht van HC kan vormen: alles wat HC mist of niet goed doet kan alsnog worden toegevoegd of veranderd.

4 Beveiliging HyperCard stacks

Het is veelal wenselijk om de mogelijkheden van een (eind)gebruiker te beperken. HC geeft de mogelijkheid om dit te doen door het zogenaamde User Level te beperken tot een van de bovenvermelde niveaus. Dit kan een stack zelf doen als die wordt opgestart. Er is wel enige kennis van de interne werking van HC nodig om dit 'foolproof' te doen. HC heeft namelijk nogal wat achterdeurtjes om het User Level toch hoger te kunnen zetten (de nog te bespreken Message Box is het meest voor de hand liggende voorbeeld).

Een tweede aspect is de beveiliging van de stack als geheel: men wil niet dat iedereen maar een stack met vertrouwelijke gegevens in kan zien. HC biedt dan ook de mogelijkheid om een stack met behulp van een password te beveiligen. Bij opstarten van de stack zal telkens om het password worden gevraagd. Dit beveiligingsmechanisme is echter niet betrouwbaar. Software Engineering heeft inmiddels de beschikking over een HC-stack, die de beveiliging van andere stacks ongedaan kan maken.

Als laatste aspect van beveiliging is er de beveiliging tegen gebruikersfouten. Vanaf het Typing niveau kan een eindgebruiker kaarten en vanaf het Painting niveau zelfs de hele stack weggooien. Men moet zich realiseren dat in het laatste geval dan niet alleen de (misschien vitale) inhoud van de stack weg is, maar ook de functionaliteit van de stack. Gelukkig biedt HC voldoende mogelijkheden om dit soort ongewenste neveneffecten tegen te gaan. Kaarten afzonderlijk en stacks als geheel kunnen beschermd worden. Merk op dat dit natuurlijk geen bescherming biedt tegen het in de Finder in de Trash gooien van een HC-stack. Dit moet men met andere middelen tegengaan.

Een eigenschap, die verband houdt met de beveiliging van HC-stacks, is om gemaakte veranderingen in een stack, zoals het veranderen van de inhoud van een field, meteen te effectueren. Er is geen Undo optie beschikbaar. Indien men de eindgebruiker de mogelijkheid wil geven om een stack in de oorspronkelijke staat achter te laten, zal men bij het opstarten van de stack eerst zelf een kopie moeten maken. HC biedt gelukkig voldoende mogelijkheden om dit eenvoudig te doen.

5 Object Oriented Programming in HyperCard: HyperTalk

In de bijdrage van L. Gielen wordt uitleg gegeven over het verschijnsel Object Oriented Programming. Ook HC maakt van dit programmeringsparadigma gebruik, zij het dat er een aantal vereenvoudigingen is doorgevoerd. Ieder object in een HC-applicatie (stack, background, card, button, field) heeft een standaard (overigens

wel uitbreidbare) verzameling messages, maar de bijbehorende verzameling message handlers (method's) ontbreken. Per message moet er dan ook een handler worden geschreven in de bij HC behorende programmeertaal HyperTalk. De totale verzameling handlers wordt het script van het object genoemd.

HyperTalk is volgens de makers van HC een 'high-level, object-oriented, interpreted programming language'. Men kan HyperTalk ook omschrijven als een 'object-oriented BASIC'. Ook deze bekende programmeertaal wordt geïnterpreteerd, variabelen hoeven niet te worden gedeclareerd, en de syntax benadert die van gewoon Engels. Kortom, ook het gebruik van HyperTalk is zeer snel te leren. Er hoeven geen cryptische handboeken te worden doorgewerkt.

Voorbeeld: maak een variabele die de inhoud "hallo" bevat:

HyperTalk:	put "hallo" into theString
BASIC:	THESTRING$ = "hallo"
C:	auto char theString[6]; strcpy(theString,"hallo");

HyperTalk heeft een volledige set van controlestructuren die in iedere hogere programmeertaal zitten, zoals if... en repeat.... Ook zit er een groot aantal commando's en functies in. Commando's kunnen worden gebruikt om variabelen te manipuleren (add, multiply etc.) of om de interface met de gebruiker te verzorgen (ask... om een waarde in te laten toetsen door de gebruiker). De groep functies omvat mathematische functies (sinus etc., maar ook functies voor annuïteiten) en systeemfuncties (zoals de positie van de muis).
Een sterk punt van HyperTalk is dat alle functies van HC, die normaliter onder de menubalk van HC aanwezig zijn, kunnen worden uitgevoerd. Als men bijvoorbeeld vanuit een HyperTalk-script de Paint-functies van HC gaat gebruiken, kunnen hiermee animaties worden uitgevoerd.

Zoals al vermeld is, worden naar het script van een object messages gestuurd. Deze messages geven aan wat voor actie de gebruiker heeft uitgevoerd. Een muisklik zal bijvoorbeeld de boodschap mouseDown geven. Een button script kan hierop reageren door in zijn scripts deze boodschap af te werken:

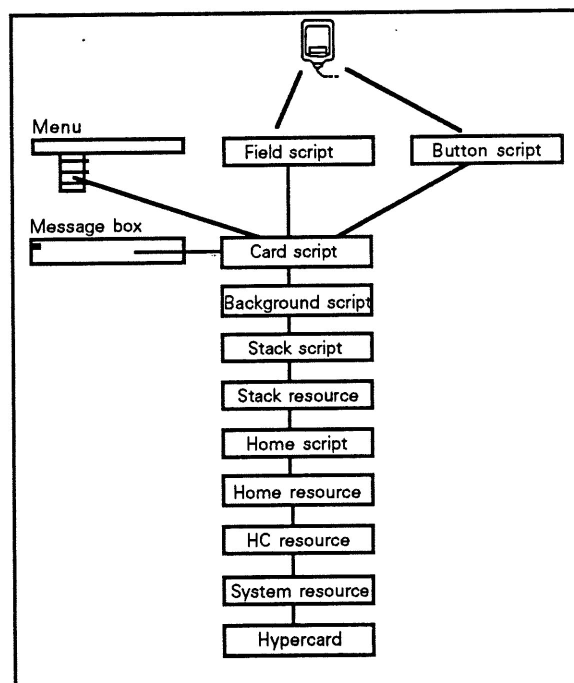
on mouseUp	- deze message handler herkent de message mouseUp.
beep	- HyperTalk command, geeft alleen een piepje.
end mouseUp	- einde message handler, er kunnen er nog meer volgen in dit script.

Messages kunnen echter ook een andere oorsprong hebben dan de gebruikershandelingen. Een message handler kan op zijn beurt nieuwe messages genereren. Ook geven bijvoorbeeld menukeuzes boodschappen. Een belangrijke oorsprong is ook de Message Box. Hierin kan de gebruiker intypen welke message hij naar welk object wil sturen.
In bovenstaand simpel voorbeeld van een button genaamd "Beep", die piept als je er op klikt, heeft men dan ook de volgende mogelijkheden:

- klik op de button "Beep".
- klik op de button "Indirect Beep" die het volgende script heeft:
  on mouseUp
    send mouseUp to button "Beep"
  end mouseUp
- typ 'send mouseUp to button "Beep"<return>' in de Message Box.

In alle drie de gevallen zal de button "Beep" de message mouseUp ontvangen en het hierbij behorende script (zijnde het geven van een piepje) uitvoeren. Zou de button de message mouseUp niet afhandelen, dan zal de mouseUp verder in het zogenaamde message inheritance path zijn weg vinden, totdat het wel wordt afgehandeld.

Onderstaande afbeelding geeft aan welke weg zo'n message volgt door de verschillende objecten. Een van deze objecten (in laatste instantie de HC-applicatie zelf) zal een handler hebben voor deze message en hem verwerken. Resources zijn de verzamelplaatsen van de reeds eerder genoemde XFCN's en XCMD's.



Figuur 4: Message inheritance

Dit grote aantal mogelijkheden om iets toe doen,

gecombineerd met het feit dat een message als mouseUp op ieder willekeurige plaats in het message inheritance path kan worden afgehandeld, is tegelijkertijd een sterk en een zwak punt van HC. Men kan namelijk aan enerzijds zeer flexibel met scripts omgaan, maar anderzijds kan het voor ieder ander, behalve de stack-ontwikkelaar, volstrekt onduidelijk zijn waar nu precies een message wordt afgehandeld. De personen die bij Software Engineering met HC gewerkt hebben zijn dan ook tot de conclusie gekomen dat het ontwikkelen van een HC-applicatie een eenmansproject is.

Voor het ontwikkelen van grote HC-applicaties zal het feit, dat de stack-ontwikkeling zich niet leent voor traditionele software-ontwikkelingstechnieken, zeker een nadeel zijn. Onderhoudbaarheid van bestaande applicaties kan helemaal rampzalig zijn.

6 Navigeren in een HyperCard Stack

Een typische HC-applicatie bestaat uit een groot aantal kaarten. Om hierin de weg te vinden zal HC een groot aantal mogelijkheden moeten bieden om snel een bepaalde kaart op te zoeken. Buiten dan dat men sequentieel door de stack heen kan gaan, biedt HC ook een zoekfunctie waardoor men zeer snel naar een bepaalde kaart kan gaan. Bijvoorbeeld het HyperTalk-commando Find "Schalk" in bkgnd field "Name" zal zeer snel de gewenste kaart selecteren.
Ook kan men op iedere kaart buttons plaatsen die de kaart verbinden met een andere kaart. Na het aanklikken van een button zal de betreffende kaart worden geselecteerd. De combinatie van deze methoden geeft de stack-ontwikkelaar genoeg gereedschap om de eindgebruiker de mogelijkheid te geven op associatieve en intuïtieve manier de stack te doorlopen. Welke kaart telkens getoond moet worden als de gebruiker op een button klikt of een andere actie onderneemt, is eenvoudig in scripts op te nemen.

HC begint echter problematisch te worden als men meerdere stacks met elkaar moet gaan combineren. Beschouw het voorbeeld van een stack die personeelsnummer en naam bevat, en van een stack die datzelfde personeelsnummer en adresgegevens bevat. Wil men nu bij een bepaalde persoon de adresgegevens ophalen, dan zal men eerst (vanuit HyperTalk) een andere stack moeten openen, de betreffende kaart zoeken, de gegevens ophalen en terug gaan naar de originele stack.

Deze omslachtige procedure geeft al aan dat HC niet een echt DBMS-gereedschap is. Er worden bijvoorbeeld geen index-bestanden bijgehouden waarmee men gerelateerde gegevensbestanden aan elkaar kan koppelen. De stack-ontwikkelaar zal zelf al deze verbanden moeten leggen, en (wat nog erger is) deze ook moeten onderhouden als er kaarten (records) worden toegevoegd.

7 Toepassingsgebieden HyperCard

De discussie in voorgaande paragraaf geeft aan dat men HC beter een gereedschap kan noemen voor het presenteren van informatie, waarbij men het begrip informatie in de ruimste zin van het woord kan zien. Niet alleen tekst, maar ook beeld en geluid kunnen eenvoudig in een stack worden gecombineerd. Tekst wordt dan gepresenteerd als inhoud van een field, beeld als de inhoud van een (card of background) picture en geluid als apart in een HC-stack op te nemen resource. HyperTalk biedt voldoende mogelijkheden om opgeslagen geluiden weer te geven.

HC kan ook uitstekend functioneren als prototype-gereedschap voor nieuw te ontwikkelen applicaties. De ontwikkelaar kan zich concentreren op de functionaliteit van het te ontwikkelen produkt, zonder zich zorgen te maken over details van de interface met de gebruiker. Juist deze interface kost in een typische Macintosh-applicatie de meeste ontwikkeltijd. Nadat het ontwikkelde prototype door de eindgebruiker is geëvalueerd, kan gestart worden met het in een traditionele programmeertaal implementeren van de applicatie.

8 Conclusies

Onderstaand wordt een opsomming gegeven van de voor- en nadelen van HC.

Nadelen HyperCard:

- Kaarten hebben de vaste grootte van een Macintosh SE scherm.
- HC is geen DBMS. Voor het leggen van verbanden tussen verschillende bestanden (stacks), zal men veel werk zelf moeten doen. De functie van DBMS-pakketten is juist om de ontwikkelaar van deze last te bevrijden.
- Leent zich niet voor toepassen traditionele software-ontwikkelingstechnieken.
- Slechte onderhoudbaarheid HC-applicaties. Het toevoegen van functionaliteit aan een al onder meerdere gebruikers verspreide stack is moeilijk, omdat iedere gebruiker zijn eigen inhoud in de stack heeft. Functionaliteit (code) en inhoud (data) zijn niet te scheiden.
- Beperkte mogelijkheden voor genereren van uitvoer op de printer.
- Doordat de scripts van objecten worden geïnterpreteerd (zoals BASIC), is de snelheid van een HC-applicatie laag als het script lang is. Daarentegen zijn de ingebouwde functies van HC, bijvoorbeeld het zoeken van een trefwoord ergens in de stack, weer erg snel.

Voordelen HyperCard:

- GRATIS, dus iedere Macintosh-gebruiker zal het binnenkort hebben.
- Zeer snel kleine applicaties te ontwikkelen. De aanprijzing waarmee microcomputers vroeger werden verkocht, namelijk dat je er de administratie van je CD/LP/Video verzameling zo makkelijk op kon bijhouden, lijkt nu pas werkelijkheid geworden. HC is de eerste bedreiging voor het bestaan van de traditionele kaartenbak die met de hand wordt bijgehouden.
- Uitbreidbaar met zelf geschreven commando's (XCMD's) en functies (XFCN's). Een goed voorbeeld zijn de beperkte mogelijkheden die HC biedt voor het creëren van rapporten op de printer. Binnen Software Engineering zijn reeds functies ontwikkeld die de stack-ontwikkelaar toegang geven tot alle printfaciliteiten die de Macintosh heeft.
- HyperCard-ontwikkelaars zijn over het algemeen bereid om veel van hun ontwikkelde stacks te delen met anderen. Hoewel er commerciële stacks beschikbaar beginnen te komen zijn de meeste ontwikkelingen nog 'public domain'. Scripts en XCMD's/XFCN's van anderen mag men zonder beperking in eigen stacks gebruiken.

9 Personalia

Jeroen Schalk is sinds 1 april 1984 werkzaam bij de sectie Software Engineering van KPMG Klynveld EDP Audit. Daarnaast studeert hij Theoretische Informatica aan de Vrije Universiteit Amsterdam.

10 Literatuur

[APDA87] APDA: HyperTalk Technical Reference Package.
[APPLUG] HyperCard User's Guide.
[APPLHS] HyperCard Help Stack.
[GOOD87] Goodman, D: The Complete HyperCard Handbook.

Programmeertheorie

door: J. Schalk

1 Inleiding

Programmeren wordt door velen nog gezien als een kunst die niet in logische of mathematische wetten te vangen valt. Dit artikel probeert hierin verandering te brengen. Het geeft aan hoe men ook op formele wijze tegen het oplossen van programmeringsproblemen kan aankijken.

Bij het programmeren kunnen twee complementaire problemen worden geformuleerd:

- Het eerste probleem bestaat uit het vanuit een bepaalde doelstelling ontwerpen van een programma. De formele specificatie van het te bereiken doel leidt tot de uiteindelijke programmatekst. Het creëren van deze tekst vindt in feite 'van achter naar voren' plaats: er wordt teruggewerkt vanuit de te bereiken doelstelling.
De huidige methodologie van programma-ontwerp gaat uit van informele specificaties die door de programmeur op intuïtieve wijze tot een programmatekst worden verwerkt. Als men echter een zeer formele specificatie van het op te lossen probleem zou kunnen geven, dan kan men met de in de volgende paragraaf geschetste methode van programma-ontwerp op automatische wijze tot de definitieve programmatekst komen.

- Het tweede probleem: bij een gegeven programmatekst te bewijzen dat het programma vanuit eveneens gegeven begincondities, inderdaad het gestelde doel vervult. Hierbij zijn de transformaties die de programmaregels uitvoeren op die begincondities van belang. De werkwijze is 'van voren naar achter'.
Ook bij dit probleem wijkt de meestal gebruikte werkwijze af van het ideale geval. In plaats van te bewijzen dat een programmatekst correct is, wordt er meestal met hulpmiddelen als debuggers en padentesters getracht om fouten op te lossen of te voorkomen. Het probleem hierbij is echter, dat men wel de **aanwezigheid** van fouten vast kan stellen, maar nooit de **afwezigheid** ervan. Deze laatste doelstelling is wel te verwezenlijken als men er in slaagt een correctheidsbewijs van een programma te geven.

In de volgende paragrafen worden beide problemen afzonderlijk behandeld. Dit kan natuurlijk slechts globaal gebeuren. De lezer zal echter merken dat zelfs met de gemaakte vereenvoudigingen het probleem van programma-ontwerp en -bewijs zich al snel verliest in een afschrikwekkende, erg wiskundig lijkende, notatie. Deze tekstuele complexiteit, die bij nadere bestudering wel mee valt, is echter nodig om de problemen op formele wijze te kunnen formuleren. Zo'n formulering is nodig als men de geschetste methodieken gaat automatiseren.

2 Het ontwerpen van programma's

Het doel van ieder programma is het vervullen van een vooraf door de ontwikkelaar gestelde eis. Deze eis wordt de postconditie genoemd, hij wordt immers na afloop van het programma vervuld. Het is een predikaat (logische uitspraak) over de variabelen die een rol spelen in het programma.

Voorbeeld: Het berekenen van het maximum (max) van twee getallen (x en y) heeft als te vervullen postconditie:

(max = x **or** max = y) **and** max ≥ x **and** max ≥ y.

In woorden: Het maximum is een van de twee getallen x of y, en het maximum is groter of gelijk aan zowel x als y.

Van belang is nu te bepalen bij welke begincondities het uitvoeren van een programma zal termineren op dusdanige manier dat de postconditie wordt vervuld. Deze begincondities vormen ook weer een predikaat over de programmavariabelen, genaamd pre-conditie. Als we het programma S noemen, dan zijn we dus op zoek naar een constructie van de vorm {PRE}S{POST}.
Aangezien iedere programmatekst S uit een combinatie van slechts een eindig aantal constructies bestaat, kan het {PRE} predikaat maar op een beperkt aantal manieren uit het {POST} predikaat ontstaan.
De volgende primitieve constructies worden onderkend:

- S is een toekenningsopdracht van de vorm x := E(xpressie).

Pre-conditie: $PRE(x := E, POST) = POST_{E \rightarrow x}$

In woorden: $POST_{E \rightarrow x}$ wordt verkregen uit POST door in plaats van de variabele x de expressie E in te vullen.

Voorbeeld: Bij de toekenningsopdracht a := 2 * b + 1 met postconditie "a = 13" zal de pre-conditie zijn: $POST_{2*b+1 \rightarrow a}$ = "2 * b + 1 = 13" = "b = 6". Als men dus weet dat vooraf b gelijk is aan 6, zal na afloop van het statement a gelijk zijn aan 13.

- S is een opvolging van twee opdrachten S_1 en S_2 van de vorm S_1; S_2.

Pre-conditie: $PRE(S_1; S_2, POST) = PRE(S_1,$

PRE(S_2, POST))

In woorden: De pre-conditie van de totale constructie is de pre-conditie van S_1 met als postconditie de pre-conditie van S_2.

- S is een voorwaardelijke opdracht van de vorm **if** B **then** S_1 **else** S_2 **endif.**

Pre-conditie: PRE(**if**, POST) =
(B **and** PRE(S_1 ,POST)) **or**
(**not** B **and** PRE(S_2 ,POST))

In woorden: De pre-conditie van het **if**-statement is de pre-conditie van S_1 (de **then**-tak) als voorwaarde B wordt vervuld, anders de pre-conditie van S_2 (de **else**-tak).

- S is een herhalingsopdracht van de vorm **while** B **do** S_1 **endwhile.**

Pre-conditie: PRE (**while**, POST) = REP_0(POST) **or** REP_1(POST) **or** REP_2(POST) ...
Waarbij REP_i(POST) staat voor het geval dat de lus precies i keer wordt doorlopen. De definitie ervan is als volgt:
REP_0(POST) = POST **and not** B
REP_N(POST) = POST **and not** B **or** PRE(**if**, REP_{N-1}(POST))

In woorden: De pre-conditie is afhankelijk van het aantal malen dat de lus wordt doorlopen. Als dit geen enkele keer is, moet de conditie B meteen onwaar zijn en moet de postconditie op dat moment dus al vervuld zijn. Dit wordt uitgedrukt in POST **and not** B. Voor n repetities wordt de **while**-lus opgevat als een aantal **if**-statements na elkaar. Op het moment dat B niet langer geldt, moet de postconditie zijn vervuld.

Bovenstaande formulering kan worden omgewerkt als we aannemen dat de waarheid van POST **and** B de waarheid van PRE(**if**, POST) impliceert, d.w.z. dat als zowel B als POST gelden, dan ook aan de postconditie van de geassocieerde **if** wordt voldaan.
Dit impliceert dat een enkele executie van de **while**-lus (weer opgevat als een enkel **if**-statement) de geldigheid van de postconditie niet aantast: de postconditie is een **invariant** van de **while**-lus.

Na enige manipulaties met de pre-conditie van de **while** zoals boven gegeven volgt dan:

POST **and** PRE(**while, true**) →
PRE(**while**, POST **and not** B)

PRE (**while, true**) kan men opvatten als die conditie waaraan voldaan moet worden opdat de **while**-lus termineert. De postconditie **true** legt immers geen verdere eisen op.
Deze formulering wordt het *Fundamental Invariance Theorem of Loops* genoemd. Het postpredikaat is invariant tijdens uitvoering van de lus. Na afloop geldt het post-predikaat dus nog steeds, en is B (de stopconditie) onwaar geworden.

De gegeven taalconstructies komen in bijna iedere programmeertaal voor. Men zou kunnen opwerpen dat door het ontbreken van een 'procedure call' mechanisme, dus het ontbreken van subroutines, de beschreven taal niet krachtig genoeg is. Dit is natuurlijk niet waar: men kan immers iedere aanroep van een procedure vervangen door de uit te voeren statements.
Zou men subroutines als een aparte taalconstructie **sub** opvatten, dan kan men zich voorstellen dat de PRE(**sub**, POST) zéér ingewikkeld is. Dit is de reden dat deze constructie hier niet wordt behandeld.

3 Een toepassing

De behandelde theorie zal nu worden gebruikt voor het ontwikkelen van een algoritme dat het maximum berekent van N getallen a_1 ... a_N (zie [DIJK76]). Evident is, dat dit algoritme van een **while**-lus gebruik zal moeten maken om voor willekeurige N te kunnen functioneren.

De postconditie moet aangeven dat er een getal is, stel a_k, groter dan alle andere getallen:

POST: $0 \leq k \leq N$ **and** $A_{i:0 \leq i < N}: a_i \leq a_k$

Dit predikaat moet worden geïmpliceerd door P **and not** B, waarbij P invariant is voor de **while**-lus, en B de stopconditie is. Een kandidaat voor P is dan ook:

P: $0 \leq k < j \leq N$ **and** $A_{i:0 \leq i < j}: a_i \leq a_k$

Merk op, dat (k = 0 **and** j = 1) → P. Merk ook op, dat (P **and** j = N) → POST. Hieruit volgt dat de stopconditie $j \neq N$ zal zijn.

De **while**-lus wordt doorlopen door j op te hogen:
PRE("j := j + 1", P) = $P_{j+1 \to j}$ =
$0 \leq k < j+1$ **and** $A_{i:0 \leq i < j+1}: a_i \leq a_k$ =
$0 \leq k < j+1$ **and** $A_{i:0 \leq i < j}: a_i \leq a_k$ **and** $a_j \leq a_k$ =
(P **and** $j \neq N$ **and** $a_j \leq a_k$) → PRE("j := j + 1", P)

Eveneens zal gelden als $a_j \geq a_k$:
(P **and** $j \neq N$ **and** $a_j \geq a_k$) →
PRE("k := j; j: = j+1",P)

Merk op, dat P dus invariant is. Uit deze overwegingen volgt dan de programmatekst:

```
/* maak P initieel waar */
k := 0; j := 1;
while (j ≠ N) do
        /* zorg dat P blijft gelden */
        if (a_j ≥ a_k)
        then
          k := j; j := j + 1;
        else
          j := j + 1;
        endif
endwhile
/* P and j = N → POST */
```

Merk op dat de stopconditie $j \neq N$ is, en niet $j < N$. De meeste programmeurs zouden geneigd zijn de laatste stopconditie te gebruiken omdat die 'extra' veiligheid geeft: het programma stopt dan zeker. Ze realiseren zich dan echter niet dat een fout in de initialisatie van j (j := j i.p.v. j := 1) een veel kleinere kans heeft om opgemerkt te worden.

4 Correctheidsbewijs

Het complementaire probleem van een programma-ontwerp is het geven van een correctheidsbewijs. Er wordt weer uitgegaan van de constructie {PRE}S{POST}. De pre-conditie geeft hier de beginvoorwaarden van de programmavariabelen aan, en de postconditie de eindvoorwaarden. S is het programma dat wordt uitgevoerd.

Voorbeeld: {x = 0}x := 1{x = 1}. Deze constructie is evident consistent, d.w.z. de preconditie, de postconditie en het programma zijn onderling met elkaar in overeenstemming.

Om nu iedere consistente {PRE}S{POST} af te kunnen leiden, introduceren we een formeel bewijssysteem voor een programmeertaal zonder 'procedure-call' mechanisme. Zo'n bewijssysteem bestaat uit een aantal axioma's en bewijsregels. Dit valt te vergelijken met de normale wiskunde, die bijvoorbeeld uitgaat van axioma's met betrekking tot de gehele getallen en verder gebruik maakt van een aantal toegelaten bewijsmethodes, zoals 'bewijs uit het ongerijmde'.

Als axioma's gebruiken we:

- Axioma voor de toekenningsopdracht van de vorm x := E(xpressie).
  $\{POST_{E \to x}\}x := E\{POST\}$ is consistent.

  In woorden: $POST_{E \to x}$ heeft hier dezelfde betekenis als bij het ontwerpen van programma's. Merk op dat hiermee het gegeven voorbeeld is 'bewezen'.

De bewijsregels zijn als volgt:

- S is een opvolging van twee opdrachten S_1 en S_2 van de vorm S_1; S_2.

  Regel: Als $\{PRE\}S_1\{POST_1\}$ en $\{POST_1\}S_2\{POST\}$ beide consistent zijn, dan geldt dat ook voor $\{PRE\}S_1; S_2\{POST\}$.

  In woorden: Als we de postconditie na uitvoering van S_1 gebruiken als preconditie voor S_2, dan zal bij uitvoeren van S_1; S_2 de postconditie van S_2 natuurlijk ook gelden.

- S is een voorwaardelijke opdracht van de vorm **if** B **then** S_1 **else** S_2 **endif.**

  Regel: Als {PRE **and** B}S_1{POST} en {PRE **and not** B}S_2{POST} beide consistent zijn, dan geldt dat ook voor {PRE}S{POST}.

  In woorden: Als B geldt, wordt S_1 (de **then**-tak) uitgevoerd. Als nu uitgaande van de pre-conditie PRE de executie van S_1 de postconditie POST vervult, zal dit ook gelden voor het **if**-statement. Als B initieel onwaar is zal de **else**-tak worden uitgevoerd.

- S is een herhalingsopdracht van de vorm **while** B **do** S_1 **endwhile.**

  Regel: Als {PRE **and** B}S_1{PRE} consistent is, dan geldt dat ook voor {PRE}S{PRE **and not** B}.

  In woorden: Als een enkele executie van S_1 (het 'body' van het **while**-statement) de waarheid van het predikaat PRE niet beïnvloedt, zal dit ook gelden voor herhaalde uitvoering ervan. Na afloop zal de conditie B onwaar zijn, anders zou immers de lus niet zijn gestopt.

Het gegeven bewijssysteem kan gebruikt worden om iedere consistente {PRE}S{POST} af te leiden (zie [BAKK80]). Ieder statement wordt steeds verder ontleed met gebruikmaking van de gegeven bewijsregels, totdat men op een toekenningsopdracht uitkomt. Het consistent zijn van zo'n toekenningsopdracht is eenvoudig met het gegeven axioma na te gaan.

5 Een toepassing

Als toepassing geven we een correctheidsbewijs voor een programma dat een geheel getal tot een bepaalde macht verheft. Voorbeeld: $2^3 = 8$. Men kan deze machtverheffing berekenen door herhaald (3 maal) vermenigvuldigen met 2, met als startwaarde het getal 1.

De pre-conditie van het probleem moet de start-

waarden van alle relevante variabelen geven. Het programma bestaat uit de **while**-lus met daarin de vermenigvuldiging, en de postconditie moet aangeven dat het gewenste resultaat is bereikt.

Dus, gegeven de pre-conditie {PRE}:
{x = m **and** m ≥ 0 **and** y = 1}
het programma S:
while x ≠ 0 **do** y := y * z; x := x - 1 **endwhile**
en de postconditie {POST}:
{x = 0 **and** m ≥ 0 **and** y = z^m}
verloopt het bewijs dat {PRE}S{POST} consistent is als volgt:

(1) Pas de eerste bewijsregel toe:
{x ≥ 1 **and** m ≥ 0 **and** y = z^{m-x}}
=
{x ≥ 1 **and** m ≥ 0 **and** y*z = z^{m-x+1}}
y := y * z
{x ≥ 1 **and** m ≥ 0 **and** y = z^{m-x+1}}

(2) Pas nogmaals de eerste bewijsregel toe:
{x ≥ 1 **and** m ≥ 0 **and** y = z^{m-x+1}}
=
{x-1 ≥ 0 **and** m ≥ 0 **and** y = $z^{m-(x-1)}$}
x := x -1
{x ≥ 0 **and** m ≥ 0 **and** y = z^{m-x}}

(3) Evident waar is:
{x ≥ 1 **and** m ≥ 0 **and** y = z^{m-x}}
=
{x ≠ 0 **and** x ≥ 0 **and** m ≥ 0 **and** y = z^{m-x}}

(4) Combineer (1), (2) en (3) met gebruikmaking van de tweede bewijsregel:
{x ≠ 0 **and** x ≥ 0 **and** m ≥ 0 **and** y = z^{m-x}}
y := y * z; x := x -1
{x ≥ 0 **and** m ≥ 0 **and** y = z^{m-x}}

(5) Pas de derde bewijsregel toe met B = (x ≠ 0):
{x = m **and** m ≥ 0 **and** y = 1 = z^0 = z^{m-x}}
while (x ≠ 0) **do** y:=y*z; x:=x -1 **endwhile**
{x = 0 **and** m ≥ 0 **and** y = z^{m-x}}

In vijf stappen is nu dus bewezen dat de gegeven combinatie {PRE}S{POST} consistent is.

6 Conclusies

Probleem bij het eventueel gaan toepassen van voorgaand geschetste formele methoden voor programma-ontwerp en -verificatie is dat het opstellen van de gewenste postconditie (bij programma-ontwerp) voor de meeste problemen zeer moeilijk is. Voor het berekenen van het maximum van een aantal getallen is dit nog wel te doen, evenals voor andere programma's die een puur wiskundige operatie op een aantal waarden uitvoeren. Echter, voor bijvoorbeeld programma's die met de gebruiker interactie plegen is het opstellen van de postconditie vrijwel onmogelijk. De precieze toestand van bijvoorbeeld een bestand waarin een gebruiker kan hebben gewijzigd, is nauwelijks in een predikaat uit te drukken.

Een tweede probleem is dat men zich moet realiseren dat, hoewel de gegeven transformatieformules voor de te onderscheiden taalconstructies (**if, while**) zich afspelen op het diepste niveau van programma-executie, ze in feite heel oppervlakkig zijn.
De uitspraken die men over afzonderlijke statements doet, leiden niet noodzakelijk tot een dieper inzicht over de applicatie als geheel.
Men kan dit vergelijken met de situatie dat het begrijpen van algebraïsche wetten zoals commutativiteit en associativiteit van de optelling en vermenigvuldiging niet echt helpt bij het begrijpen van de Hoofdstelling van de Integraalrekening, hoewel deze laatste wet wel gebaseerd is op de eerder genoemde diepe (en tegelijkertijd oppervlakkige) wetten.

Toch kan men zich indenken dat de geschetste methoden van programma-ontwerp en -verificatie, een rol spelen bij het ontwikkelen van zeer kritische software.

Er vindt op dit gebied veel research plaats. Het automatisch evalueren van predikaten is een van de eerste toepassingen geweest van kunstmatige intelligentie (zie [NEWE57]).
Ook zijn er hulpmiddelen die de programmeur kunnen helpen bij het opstellen van de invariante predikaten die, zoals aangetoond, een belangrijke rol spelen in zowel het ontwerp- als het bewijsprobleem.
Andere, meer specialistische, voorbeelden zijn te vinden in [BYTE0388] en [BYTE0788], waarin respectievelijk het volgens formele methode ontwerpen van een 'floating point unit' en een processor wordt beschreven. Bewezen wordt dan dat de ontwikkelde hardware volgens de opgestelde specificaties werkt. Merk op dat dit zeer nauw samenhangt met programmering: het maakt niet uit of een algoritme in software of hardware wordt uitgevoerd.
Van belang is ook op te merken dat programmeertalen steeds meer faciliteiten gaan bieden ter ondersteuning van programma-ontwerp en -verificatie. Een voorbeeld is beschreven in [MEYE88], waar pre- en postcondities een integraal onderdeel van het programma (kunnen) vormen.

Het ontwikkelen van deze geautomatiseerde hulpmiddelen voor het evalueren van de transformatieregels zal er toe bijdragen dat programmering van kunst tot kunde wordt.

Pas als dergelijke hulpmiddelen voor iedere programma-ontwikkeling beschikbaar zijn en worden toegepast, is het gebruik van de term 'Software

Engineering' op zijn plaats.

7 Symbolen- en woordenlijst

Predikaat	Logische uitspraak. Gebruikt relationele expressies ($x \neq 0$) en logische connectoren (**and, or, not**).
Pre-conditie	Predikaat dat beginwaarden van de programmavariabelen beschrijft.
Postconditie	Predikaat dat eindwaarden van de programmavariabelen beschrijft.
$P \rightarrow Q$	Logische implicatie. Drukt uit dat het predikaat Q waar is als het predikaat P waar is.
A_i	Een tekstuele afkorting van 'voor alle i geldt'.

8 Personalia

Jeroen Schalk is sinds 1 april 1984 werkzaam bij de sectie Software Engineering van KPMG Klynveld EDP Audit. Daarnaast studeert hij Theoretische Informatica aan de Vrije Universiteit te Amsterdam.

9 Literatuur

[BACK86] Backhouse, RC: Program Construction and Verification, 1986.
[BAKK80] de Bakker, JW: Mathematical Theory of Program Correctness. 1980.
[BYTE0388] Byte March 1988, pagina 218: Implementing Floating Point.
[BYTE0788] Byte July 1988, pagina 305: VIPER Processor Design.
[DIJK76] Dijkstra, EW: A Discipline of Programming. 1976.
[MEYE88] Meyer, B: Object Oriented Program Construction. 1986.
[NEWE57] Newell, A, Shaw, JC; Simon, HA :GPS, General Problem Solver,.1957.

Het AppleTalk-netwerk, een beschouwing

door: J.L. Ramos Najera

1 Introductie

Nu de Apple[1] Macintosh zijn intrede heeft gedaan als hulpmiddel in de accountantscontrole van KPMG, zal het gebruik van het AppleTalk-netwerk een steeds vaker voorkomend fenomeen zijn. Het begint met een tamelijk eenvoudige opstelling van Macintoshes en laserprinters en groeit dan in vele gevallen uit tot complexe netwerken. Gezien het relatief transparant gebruik van het AppleTalk-netwerk zouden we gemakkelijk kunnen vergeten dat hierachter een doordachte netwerkarchitectuur staat met een zeer groot aantal protocollen. Een beschouwing waard.

Het jaar 1988 is voor Apple het jaar van de connectivity. Het AppleTalk-netwerk neemt hierbij een strategische plaats in. Reeds in 1983 introduceerde Apple het AppleTalk Personal Network. Het doel van het ontwerp was een zo simpel, goedkoop en flexibel mogelijke koppeling tot stand te brengen tussen computers, randapparatuur en servers. De eerste en meest voorkomende toepassing van dit netwerk was het sharen van de relatief dure laserprinters. Later volgden andere toepassingen zoals het sharen van bestanden (AppleShare) en seriële poorten (MultiTalk) en electronic mail (InBox, InterMail), toepassingen die niet meer uit de ontwikkelomgeving van Software Engineering zijn weg te denken. Al met al is dit ogenschijnlijk simpele netwerk uitgegroeid tot een van de meest toegepaste netwerken.

In dit artikel zullen we ingaan op de AppleTalk protocolarchitectuur alsmede op de functies van de verschillende AppleTalk-protocollen en hun onderlinge relatie. Voorts kijken we met de EDP-Audit-bril naar de getroffen beveiligingsmaatregelen terwille van de continuïteit van de gegevensstroom en integriteit van gegevens.
We zijn in dit geval geïnteresseerd in de afdekking van de volgende hoofdgroepen van risico's:

- transmissiefouten;
- transmissieverliezen;
- transmissie-onderbrekingen;
- ongeautoriseerde kennisneming, beïnvloeding en initiëring van berichten.

2 De AppleTalk protocolarchitectuur

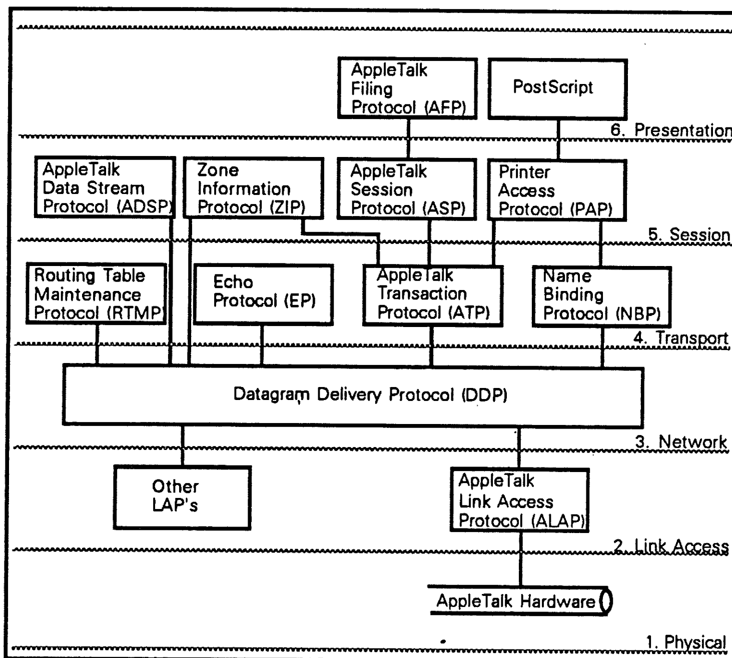
Het AppleTalk-netwerk is gebaseerd op een lagen-concept, vergelijkbaar met het wel bekende Open Systems Interconnection (OSI) referentiemodel van de International Standards Organization. Dit model groepeert netwerkfuncties in zeven lagen met elk hun eigen protocollen. De functies van de OSI-lagen zijn:

- Application layer (laag 7).
  Deze laag biedt communicatiefaciliteiten aan eindstations, zodat de verzender van een bericht een ander eindstation kan bereiken dat operationeel is in een ander systeem.
- Presentation layer (laag 6).
  Deze laag transformeert gegevens zodat deze van gelijke betekenis zijn voor zender en ontvanger. Hieronder vallen onder meer codeconversie, data-compressie, de wijze van data-opslag, database-structuren e.d.
- Session layer (laag 5).
  Deze laag zorgt voor het opzetten van een sessie (dialoog), bewaken van de gegevensuitwisseling en afbreken van de opgezette dialoog.
- Transport layer (laag 4).
  Deze laag zorgt voor het opzetten van een logische verbinding, de bevestiging van de ontvangst van berichten door de eindstations en voor het herstel van fouten.
- Network layer (laag 3).
  Deze laag maakt de connectie met de eindbestemming en bepaalt de route door het netwerk (routing).
- Data link of link access layer (laag 2).
  Deze laag zorgt voor een juist en volledig berichtenverkeer tussen twee nodes (knooppunten) in het netwerk.
- Physical layer (laag 1).
  In deze laag vindt de feitelijke transmissie van bits plaats. Deze wordt bepaald door de elektrische en mechanische karakteristieken van de verbinding tussen gebruikers- en netwerkapparatuur.

De lagenstructuur van AppleTalk alsmede de relatie ervan met het OSI-model, is weergegeven in figuur 1. De lagen 1 tot en met 5 (Physical t/m Session Layer) vormen de kern van de Apple-protocolarchitectuur. De AppleTalk-architectuur opent de weg voor ontwikkelaars om zelf protocollen te ontwikkelen ter uitbreiding van netwerkfuncties en opties. Immers, de protocollagen zijn functioneel gescheiden entiteiten die toegang tot en toevoegen van alternatieve protocollen vereenvoudigen. Een voorbeeld hiervan is het Apple-produkt EtherTalk, dat later in dit stuk wordt beschreven.

[1] *Apple, AppleTalk, AppleShare en LaserWriter zijn geregistreerde handelsmerken van Apple Computer, Inc.*

Het AppleTalk Data Stream Protocol zal in dit artikel niet worden behandeld, aangezien de documentatie hieromtrent nog niet definitief is. Het zeer uitgebreide AppleTalk Filing Protocol zal daarom worden behandeld. PostScript ten slotte, is een door Adobe ontwikkelde page description language en voor verdere detaillering wordt u verwezen naar de desbetreffende documentatie. De functies van de overige protocollen zullen worden behandeld in opeenvolgende orde van laag.



Figuur 1: De AppleTalk-architectuur

3 AppleTalk hardware en Physical Layer

De physical layer van het AppleTalk-netwerk bestaat uit een serieel communicatiesysteem in de vorm van een busstructuur met aftakkingen naar de op het netwerk aangesloten apparatuur (devices). De aftakking wordt gevormd door een T-verbinding (connector kit) die de signalen van het netwerk aftapt. Dit proces gaat met een zodanige verzwakking van het elektrische signaal (demping) gepaard dat er maximaal circa 32 devices kunnen worden aangesloten. Het voordeel van het aftakkingsconcept is evenwel, dat de gegevensstroom niet wordt onderbroken indien een van de devices niet goed functioneert of wordt afgekoppeld. Voor de bedrading wordt gebruik gemaakt van twee-aderig shielded twisted pair kabels met een maximum totaallengte van 300 meter. De transmissiesnelheid bedraagt 230.4 kilobits (Kb).

3.1 Controller, modulatie en framing

Buiten de bekabeling is al de benodigde hardware voor het gebruik van het AppleTalk-netwerk standaard in elke Macintosh aanwezig. Voor de liefhebbers van de techniek: de AppleTalk-hardware is uitgerust met de Zilog Z6530 serial communications controller die opereert met SDLC (synchronous data-link control) frame-formaat, gebruik makend van de zelf-klokkend FM-0 modulatietechniek, zero-bit stuffing en automatische generatie van cyclic redundancy check (CRC) voor detectie van transmissiefouten. De ontvanger synchroniseert automatisch bij ontvangst van een begin flag van een frame en kan zodanig worden geprogrammeerd dat het pakketten weigert met een ander adres dan het gebruikers- of broadcast-adres. Dit wordt dan ook als adresfilter gebruikt door AppleTalk.

4 Het AppleTalk Link Access Protocol

Het AppleTalk Link Access Protocol (ALAP) verzorgt, conform de OSI Data Link Layer, de communicatie tussen twee nodes die aan het netwerk zijn verbonden. De hoofdfuncties van het ALAP zijn:

4.1 Verzorgen van de toegang tot de bus

AppleTalk regelt de toegang tot de bus door middel van het Carrier Sense Multiple Access/Collision Avoidance (CSMA/CA) concept. Dit betekent dat er vermeden wordt dat er meerdere zenders tegelijkertijd gebruik maken van de bus, waardoor een botsing (collision) ontstaat. Dit in tegenstelling tot CSMA/Collision Detection (ook wel bekend als IEEE 802.3), waar collisions wel toegelaten worden maar vervolgens worden gedetecteerd en afgehandeld. Collisions worden vermeden door het gebruik van Request-to-Send (RTS) en Clear-to-Send (CTS) control-pakketten. Collisions kunnen alleen optreden tijdens de RTS-CTS cyclus, resulterend in het verloren gaan van een van de twee control-pakketten, waardoor er geen datapakketten worden verzonden.

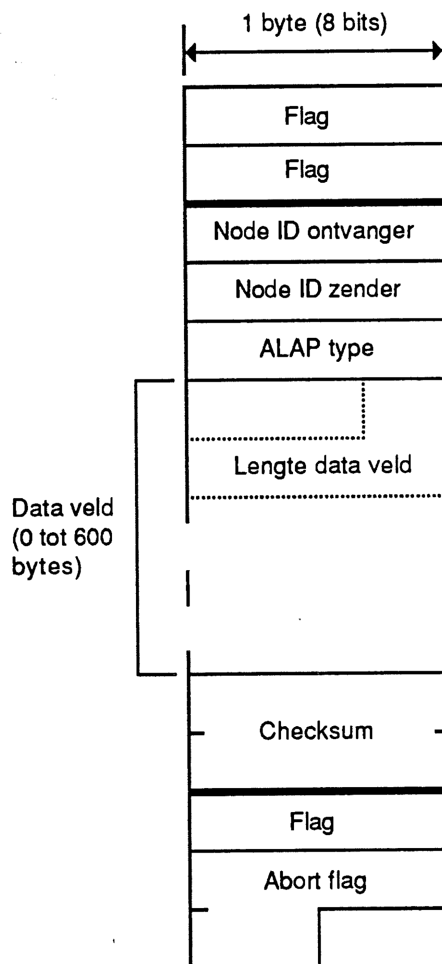
4.2 Node-adresseringsmechanisme

Nodes in een AppleTalk-netwerk hebben geen permanente adresidentificatie, zoals b.v. bij Ethernet. Als een node geactiveerd wordt, wordt dynamisch een adres (node ID) gegenereerd. Indien dat adres ongebruikt is wordt het adres toegewezen, anders wordt opnieuw een ID gegenereerd.

4.3 Data-verzending en data-ontvangst

AppleTalk maakt gebruik van een bit oriented data link protocol voor verzending en ontvangst van berichten. Dit houdt in dat alle bit-combinaties binnen een frame toegestaan zijn en als zodanig transparant zijn voor het ALAP-protocol. Een ALAP-frame bestaat uit een 3-byte header gevolgd door een data-veld van variabele lengte (van 0 tot 600 bytes) en een 16-bits Frame Sequence Check (FSC) gebaseerd op het CRC-CCITT-algoritme. De header bevat de node ID van de ontvanger, de node ID van de zender en een ALAP-frame-type-veld van één byte (zie figuur 2).

4.4 Integriteit en lengte van pakketten
Ontvangst van pakketten is in wezen het tegenovergestelde van het verzendproces. Pakketten worden uitgepakt en vervolgens gecontroleerd op lengte, type en integriteit. Deze voorzieningen zijn echter detectief en gaan niet gepaard met correctieve maatregelen . Er vindt geen hertransmissie plaats noch garandeert ALAP dat een pakket aankomt. Immers, er wordt door ALAP geen protocol gevoerd waarbij een verzending gepaard gaat met een ontvangstbevestiging. Dit betekent dat recovery en ontvangst van berichten gewaarborgd moet worden in de protocollen van de hogere lagen. Bij AppleTalk is dit een van de taken van het AppleTalk Transaction Protocol.



Figuur 2: Formaat van een ALAP-frame

5 Andere Link Access protocollen
Zoals eerder genoemd is het mogelijk om andere LAP's te implementeren binnen de AppleTalk-architectuur. Een van de meest recente implementaties is het Apple-produkt EtherTalk. Dit produkt biedt de mogelijkheid om de Macintosch te koppelen aan het relatief snellere Ethernet-netwerk (transmissiesnelheid van 10 megabits). Hiervoor is voor de verschillende types Macintoshes additionele hardware beschikbaar.
Aangezien Ethernet-adressen anderssoortig zijn dan AppleTalk-adressen, is er een protocol toegevoegd, genaamd AppleTalk Address Resolution Protocol, dat AppleTalk-adressen afbeeldt op Ethernet-adressen en AppleTalk-pakketten inkapselt in Ethernet-pakketten. Dit mechanisme is transparant voor de protocollen van de hogere lagen, zodat applicaties hiervan niets merken.

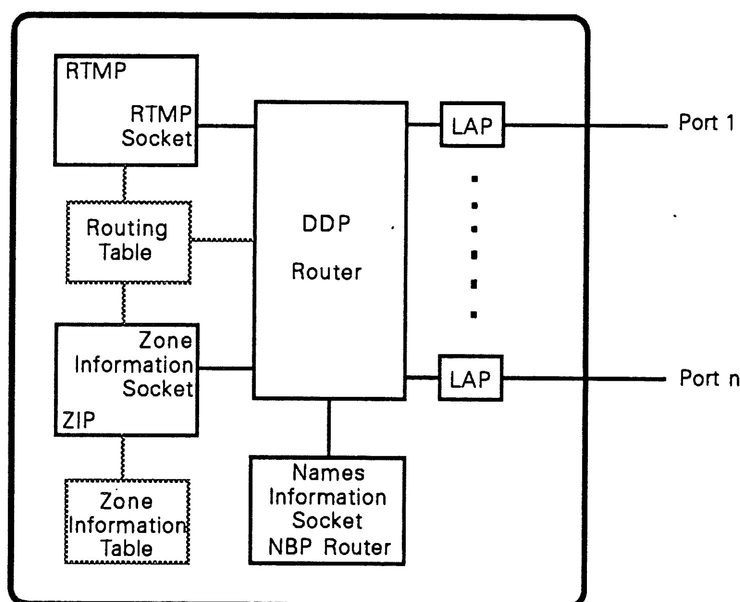
6 Het Datagram Delivery Protocol
Het DDP is een eenvoudig protocol met als hoofddoel het routen van pakketten door het gehele netwerk. Dit netwerk, ook wel internet genaamd, kan bestaan uit een of meerdere netwerken die met elkaar verbonden zijn door middel van bridges. Waar het ALAP zorgt voor de transmissie van pakketten tussen twee nodes, zorgt het Data Delivery Protocol (DDP) voor het verzenden van datagrams (DDP pakketten) tussen twee eindbestemmingen (in AppleTalk sockets genoemd). Een socket is een logische entiteit binnen een node. Sockets zijn eigendom van socket clients of anders gezegd, processen die actief zijn binnen een node. Zij verzorgen de DDP-diensten voor hun cliënten.

6.1 Het internet en netwerknummers
Elk netwerk binnen een internet wordt geïdentificeerd door een uniek 16-bit netwerknummer. Netwerknummer 0 is gedefinieerd als onbekend en identificeert in feite het locale netwerk. Theoretisch kunnen dus 65,535 (2^{16}) netwerken worden gedefinieerd binnen een internet. Door het socket AppleTalk-adres (socket-nummer en nodeID) te combineren met het netwerkadres kan elk socket binnen een internet worden geidentificeerd (ook wel het socket internet adres genoemd).
Een DDP datagram bestaat uit een header gevolgd door een data-veld van maximaal 586 bytes. De header bevat o.a. een DDP checksum-veld en een hop count. Het gebruik van het DDP checksum-veld is optioneel en wordt alleen gebruikt bij verzending van datagrams buiten het locale netwerk. Indien dit veld een waarde ongelijk nul bevat, wordt het gebruikt voor het detecteren van verminkingen van datagrams. Verminking kan worden veroorzaakt door operaties binnen bridges in een internet. Het hop count-veld bevat een waarde die aangeeft hoe vaak een datagram een brug heeft gepasseerd. Als dit veld de waarde 15 bevat, wordt het datagram verwijderd ter voorkoming van eventuele rondzwervende datagrams. Dit laatste komt voor in situaties van korte duur waarbij de routing-tabellen worden bijgewerkt door het Routing Table Maintenance Protocol .

Hoe gaat nu het routen van datagrams te werk? De zendende node bepaalt of de ontvanger aanwezig is binnen het locale netwerk. Indien dit het geval is wordt de datagram via het ALAP naar de node ID van de ontvanger verzonden. Anders wordt aan ALAP een verzoek gedaan om de datagram te zenden naar een bridge. De bridge bepaalt dan aan de hand van routing-tabellen wat de kortste weg is naar de ontvanger.

7 Routing Table Maintenance Protocol

Zoals genoemd in de vorige paragraaf, maakt DDP gebruik van routing-tabellen (RT's) opgeslagen in de bridges voor het routen van pakketten door het internet. De voornaamste taak van het Routing Table Maintenance Protocol (RTMP) is, deze tabellen te onderhouden. Een model van een bridge wordt weergegeven in figuur 3.



Figuur 3: Model van een bridge

7.1 Verschijningsvormen van bridges

De taak van een bridge is om AppleTalk-subnetten met elkaar te verbinden. Er worden drie soorten bridges onderscheiden:

- Local bridges.
  Dit zijn bridges die lokale AppleTalk-netwerken rechtstreeks met elkaar koppelen, bijvoorbeeld de Hayes Interbridge.
- Half bridges.
  Dit soort bridges is geschikt om remote AppleTalk-netwerken met elkaar te verbinden. Twee bridges zijn met elkaar gekoppeld door middel van een lange afstandcommunicatielijn (b.v. een kies- of een huurlijn), maar vormen als het ware een geheel, vandaar de naam. Een voorbeeld hiervan is ook de Hayes Interbridge.
- Backbone bridges.
  Dit zijn bridges die AppleTalk-netwerken met elkaar koppellen door middel van een backbone-netwerk. Zo'n backbone-netwerk betreft meestal een ander soort netwerk met een grotere capaciteit en overdrachtsnelheid. Een betere en gangbare benaming voor zo'n bridge is gateway, wat aanduidt dat er een protocolconversie dient plaats te vinden bij de koppeling tussen netwerken. Een voorbeeld van zo'n gateway is de bij Software Engineering gebruikte Kinetics FastPath, die het Ethernet backbone-netwerk koppelt met het AppleTalk-netwerk.

7.2 Bridges en de Routing Table

Alle bridges onderhouden complete RT's om pakketten te routen naar het subnetwerk van de ontvanger (volgens het store and forward principe). Een RT bestaat uit een aantal entries die o.a. weergeeft welke netwerken bereikbaar zijn via welke poort. Voor het onderhoud van de routingstabellen maken bridges gebruik van het RTMP. Dit gebeurt door het uitwisselen van routingstabellen door middel van RTMP-datapakketten. Indien van een entry binnen de bridge RT geen RTMP-pakketten zijn ontvangen binnen de validity time (20 sec), wordt de waarde van de entry state value gedeclasseerd (van good naar suspect, van suspect naar bad). Vervolgens worden entries met een bad entry state value eventueel verwijderd.

8 Het Name Binding Protocol

Zoals we tot nu toe hebben kunnen constateren, maken protocollen gebruik van node IDs, socketnummers en netwerknummers, om pakketten te kunnen adresseren. Gebruikers willen echter gebruik maken van meer sprekende identificaties, zoals namen. Het systeem zal dan de naam moeten converteren naar het netwerkadres. Welnu, dit is de taak van het Name Binding Protocol (NBP).

8.1 Network-Visible Entities

AppleTalk kent het concept van network-visible entity (NVE). In het algemeen zijn dit alle entiteiten binnen een netwerk waartoe men zich toegang kan verschaffen. Meer specifiek zijn de socket clients in een internet de NVEs. NVEs kunnen zichzelf een entity name toekennen bestaande uit een string van karakters. Een entity name bestaat uit drie velden:

- een objectveld. Dit veld bevat de "aanroepnaam";
- een typeveld. Dit veld bevat het type object, b.v. het type LaserWriter. Dit vergemakkelijkt het zoeken naar NVEs van een bepaald type;
- een zoneveld. Dit veld bevat de naam van de zone waar het object zich bevindt (zie hiervoor de paragraaf omtrent ZIP).

Deze drie strings van maximaal 32 karakters elk worden in deze volgorde van elkaar gescheiden door resp. een dubbele punt en een @-teken. De entity name "LaserWriter Room 11.21:LaserWriter@Software Engineering" b.v.,

identificeert een NVE van het type LaserWriter binnen de zone Software Engineering genaamd LaserWriter Room 11.21. Voorts is het mogelijk om voor het object en type string een '=' teken te gebruiken als aanduiding van alle mogelijke waarden (wildcard) en voor de zone string een '*' teken te gebruiken als default zone. Men kan dan bijvoorbeeld refereren naar alle NVEs binnen de default zone door gebruik van de entity name "=:=@*".
Voordat een entity kan worden benaderd moet het adres worden verkregen a.d.h.v. de entity name. Dit is een proces dat name binding wordt genoemd.

Elke node onderhoudt een names table (NT) bestaande uit een mapping van entity names met entity addresses. Name binding wordt uitgevoerd door gebruik te maken van NBP om entity adressen op te vragen in de names directory (ND). De ND is een gedistribueerde data base van alle names-address mapping, in feite een vereniging van alle NTs in het internet.

9 Het AppleTalk Transaction Protocol

Het AppleTalk Transaction Protocol (ATP) heeft als voornaamste doel het verzorgen van een juiste en volledige transmissie van pakketten tussen sockets.
Bij het ATP wordt de interactie tussen zender en ontvanger een transaction genoemd. De aanvrager initieert een transaction door het zenden van een transaction request (TReq) van één pakket lang naar de ontvanger. De ontvanger voert de transactie uit en retourneert een transaction response (TResp) van een of meer pakketten. Voorts is het mogelijk om meer dan één transaction request uit te hebben staan, doordat transactions worden geïdentificeerd door een transaction identifier (TID). TReqs en TResps zijn dus met elkaar gekoppeld door die TID.

9.1 Transactietypen

Tijdens het proces kunnen zich een aantal foutsituaties voordoen zoals het verloren raken van een TReq, vertraging van de ontvangst van een TResp en onbereikbaar worden van de ontvanger.
De aanvrager van een transaction ontvangt dan geen TResp en moet dus concluderen dat de transaction niet is uitgevoerd. In dit geval voert de aanvrager een herstelprocedure uit die bestaat uit een combinatie van een time-out en automatische retry. Hoe dit in zijn werk gaat is afhankelijk van het type transaction. De twee typen transactions zijn:

- At-Least-Once Transactions.
  Dit mechanisme wordt uitgevoerd in geval van transacties waarvan herhaaldelijke uitvoering gelijk is aan eenmalige uitvoering, zoals het opvragen van een saldo. Dit worden idempotent transacties genoemd. In dit geval vindt een maximaal aantal malen hertransmissie plaats van een TReq, indien de TResp niet is ontvangen binnen een bepaald tijdsinterval. Dit kan leiden, in het geval van verlies (of vertraging) van een TResp, tot herhaaldelijk uitvoeren van de desbetreffende transactie.
- Exactly-Once Transactions.
  Dit mechanisme wordt uitgevoerd bij transacties waarvan niet gewenst is dat zij meer dan één keer worden uitgevoerd, zoals het updaten van een saldo. Om dit te verwezenlijken houdt de ontvanger een lijst bij van alle ontvangen TReqs en verzonden TResps die nog in behandeling zijn. Indien een TReq wordt ontvangen wordt deze eerst vergeleken met de lijst a.d.h.v. de TID. Indien deze niet voorkomt, wordt deze aan de lijst toegevoegd. Komt de TReq wel voor in de lijst, dan kan het zijn dat deze al is uitgevoerd en bevestigd met een of meer TResps of dat deze nog in uitvoering is. In het laatste geval wordt de TReq genegeerd, in het eerste geval worden de bijbehorende TResps gezocht in de lijst.
  Indien een TREsp wordt ontvangen door de aanvrager, zendt deze een TRel uit voor de desbetreffende transaction, zodat de ontvanger de gereserveerde ruimte in de lijst kan vrijgeven. Om te voorkomen dat een transaction eeuwig in de lijst blijft staan bij verlies van een TRel-pakket, wordt deze in de lijst voorzien van een tijdstempel. Transactions die te lang open staan worden uit de lijst verwijderd.
  Er moet hierbij worden opgemerkt dat eenmalige uitvoering van exactly-once transactions alleen gegarandeerd is binnen het locale netwerk en niet voor het internet. Immers, transacties kunnen b.v. vertraagd worden in bridges en dan alsnog aankomen nadat dezelfde transactie was afgerond en aldus niet meer aanwezig in de lijst. Er moeten dus alsnog maatregelen worden getroffen in protocollen van de hogere lagen om deze ongewenste situatie af te handelen.

Een enkele TReq kan resulteren in één of meer TResp-pakketten met een maximaal aantal van acht (de response message genaamd). Een transaction is dan alleen afgehandeld als alle TResp-pakketten, behorende bij de TReq, zijn ontvangen. Het ATP bewaakt de volledigheid en de volgorde van te ontvangen pakketten.

10 Het Echo Protocol

Het Echo Protocol (EP) is een zeer eenvoudig protocol, dat voornamelijk bedoeld is als netwerk management faciliteit. Twee mogelijke doelen van het gebruik van het EP zijn:

- het testen of een node toegankelijk is;

- het bepalen van de tijd die nodig is om een pakket heen en weer te zenden naar een node.

Het EP is geïmplementeerd in elke node als een statische socket met nummer 4 en met een EP proces op dat socket. Als er een DDP-pakket wordt ontvangen van het type 4 en dat is niet leeg (d.w.z. het data-veld heeft een lengte ongelijk aan nul), dan wordt de eerste byte van het data veld geëvalueerd. Bevat dit de waarde 1 (Echo Request) dan wordt er een DDP teruggezonden naar de aanvrager met de waarde 2 (Echo Reply) in de eerste byte van het data-veld. De aanvrager kan bij ontvangst concluderen dat de desbetreffende node bereikbaar is en kan dan bepalen hoelang de wachttijd is voor het zenden van een bericht naar en het ontvangen van een bevestiging van die node.

11 Het Zone Information Protocol

Tijdens beschouwing van de vorige protocollen werd het begrip zone geïntroduceerd. Een zone kan worden beschouwd als een deelverzameling van netwerken binnen een internet. Zorgden bridges voor het onderhoud van de routing tables, zij zorgen eveneens voor het mappen van netwerknummers met zone names. Een bridge houdt van elk van zijn poorten bij wat de naam is van de zone waarop die is aangesloten. Het onderhouden van de mapping wordt verzorgd door gebruik van het Zone Information Protocol (ZIP).
De twee hoofdtaken van het ZIP zijn:

- het onderhouden van het netwerknummer met zone name mapping;
- ondersteuning aan nodes bieden, indien deze de mapping willen aanvragen of wijzigen.

11.1 Zone names en Zone Information Table

Elke bridge bevat een zone information table (ZIT) waarvan elk entry bestaat uit een <netwerk nummer, zone name> paar. De zone name kan de waarde NIL (of wel "geen") bevatten, wat aangeeft dat de naam nog onbekend is. Voor het onderhouden van de ZIT, houdt het ZIP-proces in een bridge nauwgezet bij wat er in de routing table (RT) van het RTMP gebeurd. Als het ZIP-proces ontdekt dat een netwerknummer aanwezig is in de RT maar niet in de ZIT, dan wordt er een entry aangemaakt voor het desbetreffende netwerk en wordt de naam op NIL gezet. De entry wordt verwijderd, indien het netwerknummer wel in de ZIT voorkomt maar niet in de RT.
Het ZIP-proces gaat regelmatig op zoek naar entries met zone name NIL. Indien er een of meer worden gevonden dan wordt de zone name opgevraagd door middel van het verzenden van een Zone Query (met vermelding van de netwerknummers) door het internet. Indien een bridge een Zone Query ontvangt en het heeft betrekking op informatie die hij bezit, dan dient deze een antwoord terug te sturen met vermelding van de bijbehorende zone name.

12 Het Printer Access Protocol

Het Printer Access Protocol (PAP) is een session layer voor het opzetten en controleren van sessie tussen werkstations en printer servers, waarbij een of meer connecties worden toegestaan tussen werkstations en printer servers.
Het principe bestaat uit het openen van een connectie door middel van een PAPOpen call, het ontvangen en verzenden van data door middel van PAPRead resp. PAPWrite calls, het tot stand houden van de connectie en het sluiten ervan door middel van een PAPClose call. PAP maakt hiervoor gebruik van ATP exactly-once transactions, immers het is bijvoorbeeld niet wenselijk dat data meer dan een keer wordt afgedrukt. Voorts kan altijd de status van de server worden opgevraagd door middel van een PAPStatus call.
Printer servers kunnen meer dan een connectie opzetten. Hoeveel connecties worden opgezet is afhankelijk van de server. De LaserWriter b.v. kan maar één connectie tegelijkertijd accepteren. De langst wachtende connectieaanvraag wordt als eerste behandeld.
Tijdens het verzenden van data voert PAP twee hoofdfuncties uit, namelijk:

- Detecteren van half-open connecties.
  Een connectie wordt als half-open beschouwd indien er binnen de connection timer van twee minuten geen data is ontvangen. Een reden hiervoor zal kunnen zijn dat een van de partijen "dood" is. Een half-open connectie wordt na detectie opgeheven. PAP zorgt ervoor dat de connectie open blijft door herhaaldelijk zogenaamde tickling pakketten te laten verzenden door het ATP.
- Het verzenden van de data zelf.
  Het verzenden van data is read-driven. Dit houdt in dat de ene partij aan de andere te kennen geeft bereid te zijn data te ontvangen. Omdat ATP exactly-one transactions niet gewaarborgd zijn in een internet (zie ATP-paragraaf), moeten er aanvullende maatregelen worden genomen door het PAP om dubbele ontvangst van requests uit te filteren. Dit wordt mogelijk gemaakt door elk ATP-pakket te voorzien van een volgnummer. Requests die niet met een opvolgend nummer worden ontvangen, worden door de ontvanger verwijderd.

13 Het AppleTalk Session Protocol

Het AppleTalk Session Protocol (ASP) is voornamelijk bedoeld voor het opzetten en controleren van sessies tussen werkstations en servers. De service interface naar ASP is zo onafhankelijk mogelijk van de onderliggende lagen geschreven, zodat programma's die van ASP

gebruik maken, relatief eenvoudig kunnen worden herschreven voor een andere machine. ASP zelf is wel sterk gerelateerd aan ATP en er als zodanig niet van te scheiden. ASP voorziet in de volgende functies:

13.1 Het openen en sluiten van een sessie

Voor het opzetten van een sessie dienen de ASP-cliënten te weten te komen wat de maximale lengte is van commando's en replies, om te bepalen of de onderliggende lagen geschikt zijn. Als dit het geval is wordt door het werkstation het adres van de server bepaald en wordt er een sessie opgezet. Deze sessie krijgt een sessionID toegewezen die in het verdere verloop moet worden gebruikt.

13.2 Het zenden van een command naar de server en het terugzenden van een reply

Dit concept lijkt sprekend op dat van ATP. Het werkstation geeft een commando aan de server (b.v. lees een blok van een bestand) en de server verzendt een reply terug door middel van een of meer pakketten die de gevraagde data bevatten.

13.3 Het schrijven van datablokken van het werkstation naar de server

Het schrijven van een aantal datablokken van het werkstation naar de server staat haaks op het bovenstaande concept. Om evenwel hierin te voorzien wordt een ASP write call vertaald naar twee transacties. Met de eerste transactie wordt er aan de server te kennen gegeven dat het werkstation wil schrijven naar de server en voorziet dit van bijbehorende informatie (b.v. hoeveel data er geschreven dient te worden). De server analyseert vervolgens de transactie. Indien er niet aan kan worden voldaan volgt een reply met een foutindicatie. Anders zendt de server een ATP request naar het werkstation en kan deze een reply terugzenden met de te schrijven data.
Net zoals PAP houdt ASP een sessie open door gebruik van het Tickling mechanisme en worden pakketten voorzien van een volgnummer voor het filteren van dubbel ontvangen requests.

13.4 Het verzenden van attentions van de server naar het werkstation

Tijdens een openstaande sessie kan de server een attention request verzenden naar het werkstation indien de server de aandacht nodig heeft van het werkstation. Aan de hand van het 2-byte attention data-veld, kan het werkstation bepalen wat de reden is (b.v. de server gaat down) en actie ondernemen.

Buiten een sessie om is een werkstation altijd in staat om de status van de server op te vragen.
ASP voorziet niet in een authenticatiemechanisme zodat, indien gewenst, aanvullende maatregelen moeten worden genomen in de bovenliggende protocollen of in de applicaties.

14 Het AppleTalk Filing Protocol

Het doel van het AppleTalk Filing Protocol (AFP) is het regelen van file sharing tussen een file server en meerdere werkstations. AFP maakt hierbij gebruik van de diensten van ASP. Tot nu toe kunnen deze werkstations zowel Macintoshes- als MS-DOS-computers zijn, zodat deze bestanden van file servers kunnen sharen die op AFP gebaseerd zijn. AppleShare is de Macintosh-implementatie van een op AFP gebaseerde file server. MS-DOS-computers kunnen eveneens bestanden uitwisselen met een AppleShare file server door middel van het Apple-produkt AppleShare PC. Hierbij is evenwel additionele hardware benodigd in de vorm van de AppleTalk PC card.
Normaal geproken verschaft een applicatie zich toegang tot bestanden door middel van file system commando's van het gebruikte operating system. Is nu het bestand aanwezig op een file server dan dienen deze commando's vertaald te worden in AFP-commando's. Hiervoor is de, in het werkstation geïmplementeerde, Translator verantwoordelijk. Deze zendt en ontvangt vervolgens AFP-pakketten van en naar de file server. Aangezien de AFP-pakketten de interface vormen tussen werkstation en server, kan in principe elk soort computer die ASP-diensten ondersteunt, als file server dienen. Een voorbeeld hiervan is het produkt AlisaTalk. AlisaTalk is een implementatie van een op AFP gebaseerde file server op de VAX-computers van DEC.

Het AFP voorziet o.a. in het volgende:

- een authenticatiemechanisme;
- een intern hiërarchisch file system gelijkend op die van de Macintosh;
- protectie op volume- en folderniveau gelijkend op die van Unix (owner, group en other);
- identificatie van bestanden zowel op de Macintosh-manier (31 karakters) als op de MS-DOS-manier (acht karakters gevolgd door een punt en een extensie van drie karakters).

15 Beheersbaarheid-, beveiligings- en performance-aspecten van het AppleTalk-netwerk

Zoals we hebben kunnen constateren is het ogenschijnlijk simpele AppleTalk-netwerk gebaseerd op een doordachte open architectuur, met een verscheidenheid aan protocollen. Veel is er aan gedaan om het netwerk zo transparant mogelijk te maken voor de gebruiker door o.a. ingebouwde hardware, eenvoudige bekabeling, gebruik van namen voor identificatie (NBP), dynamische toekenning van node adressen (ALAP) en transparante routing door het internet (DDP, RTMP). Apple is dan ook van plan om AppleTalk te introduceren als defacto industriestandaard.

Kijken we door de EDP Audit-bril dan kunnen we stellen dat er een aantal bedreigingen niet afgedekt is:

- Doordat de huidige connectors niet zijn vast te zetten, komt het regelmatig voor dat een verbinding verbroken wordt. Hierdoor wordt de bus "verkort" tot de verbroken verbinding. Dit probleem is recentelijk opgelost door introductie van de nieuwe Apple LocalTalk bedrading.
- Indien de ontvanger zo wordt geprogrammeerd dat alle pakketten worden geselecteerd, is men in staat om alle pakketten te ontvangen, wat tot ongeautoriseerde kennisneming van de berichten kan leiden.
  De Apple-programma's Peek en Poke zijn voorbeelden van deze implementatie. Met behulp van Peek is men in staat om alle pakketten, die over het netwerk worden verzonden, te analyseren. Bij het programma Poke is men zelfs in staat om pakketten in het netwerk te injecteren met een ander verzendadres dan het gebruikersadres. Het is dan relatief eenvoudig om ongeautoriseerde initiëring van berichten te veroorzaken. Nog een praktijkvoorbeeld is het in Nederland door PCCA ontwikkelde pakket MacNieuwsgierig. De ontwikkelaar van dit pakket was in staat om alle pakketten van het netwerk uit te lezen, zodat hij b.v. op de hoogte was van de salarissen van zijn collega's. Aangezien de markt terecht nogal huiverig was voor uitgifte van dit pakket, heeft men het niet uitgebracht.
- Er zijn geen waarborgen voor de juistheid van berichten op end-to-end-niveau. Het gebruik van de checksum in datagrams is optioneel.
- Er zijn geen herstellende mechanismen bij langdurige transmissie-onderbrekingen.

Bij het lezen van dit artikel zult u misschien zelf nog een aantal bedreigingen constateren, die niet in bovenstaande rijtje voorkomen. Vanzelfsprekend dienen aanvullende maatregelen in de hardware, applicaties of de organisatie te worden genomen om de openstaande risico's af te dekken.

Blijft over het performance-aspect. Uit ervaring kan worden opgemaakt dat het AppleTalk-netwerk voldoende capaciteit biedt bij gebruik als laserprinter sharing. Echter, indien er intensief gebruik wordt gemaakt van het netwerk door meerdere gebruikers tegelijkertijd (zoals bij een ontwikkelomgeving met gezamenlijk gebruik van bestanden), dan gaat de performance aanzienlijk omlaag en is de beschikbare capaciteit spoedig ontoereikend.

Voorts is de Macintosh in combinatie met AppleTalk niet geschikt voor real-time-toepassingen. De 680x0 processor van de Mac transporteert namelijk gegevens naar de serial communications controller zonder ondersteuning van Direct Memory Access (DMA). Dit te zamen met de relatief hoge transmissiesnelheid en de kleine 3-byte buffer, zorgt ervoor dat de 680x0 processor niets anders kan doen tijdens het ontvangen en verzenden van pakketten.

16 Personalia

José-Luis Ramos Najera is sinds 1981 werkzaam bij KPMG Klynveld EDP Audit. Na de VWO kwam hij in dienst als junior-programmeur bij de toenmalige Automatisering en Controle Groep. Sedert enige tijd is hij software engineer bij de sectie Software Engineering. Hij heeft in 1987 zijn AMBI-diploma behaald. Zijn interesses gaan voornamelijk uit naar tele- en datacommunicatie en meer in het algemeen naar operating systems en databases.

17 Literatuur

[APPL86] Apple Computer, Inc., Inside AppleTalk, Preliminary notes, July 14, 1986.

[APFP87] Apple Computer, Inc., AppleTalk Filing Protocol Version 1.1, February 17, 1987.

[APPL87] Apple Computer, Inc., EtherTalk Preliminary Notes, October 15, 1987.

[BALV83] Balvert, JH, Elbers, DJ: Inleiding datatransmissie, Samson, tweede druk 1983.

[HUGH87] Smith, HT, Armitage, WJ, Duckworth, RJ, Using the Macintosh on a Unix Network, Byte July, 1987.

[KPMG??] KPMG Klynveld EDP Audit Services, Cursus Gedistribueerde Automatisering.

PS/2 - OS/2

door: Ir. J. de Graaff
Drs. D.J.P. Witte

1 Inleiding

IBM is in 1987 met vier nieuwe modellen, onder de naam Personal System 2™ (PS/2) op de personal computermarkt gekomen. Daarnaast is, in samenwerking met Microsoft, een nieuw besturingssysteem ontwikkeld dat in staat is de hardware-faciliteiten van PS/2 ten volle te benutten. Dit systeem heeft men Operating System 2™, of kortweg OS/2 genoemd.

Aangezien IBM nog steeds veruit marktleider is, zorgt de introductie van nieuwe IBM-modellen voor grote commotie in de PC-wereld. Kloonmakers proberen zo snel mogelijk met een PS/2 look-alike te komen, fabrikanten van concurrerende systemen vrezen een dalende omzet, software-ontwikkelaars beraden zich over een overstap naar het nieuwe operating system en de eindgebruiker vraagt zich af of het nu gedaan is met het vertrouwde MS-DOS.

In dit artikel zal een overzicht worden gegeven van de meest in het oog springende eigenschappen van zowel het besturingssysteem als de hardware. Aan het eind wordt tevens een overzicht geboden van de toekomstige ontwikkelingen op dit gebied.

2 Personal System 2

2.1 Hardware

Welke processen zich precies achter het computerscherm afspelen is voor veel gebruikers niet interessant. Wat de gebruiker wil is een snelle machine. De snelheid van een computer hangt af van de verwerkingscapaciteit, die bepaald wordt door de hardware, en hoe efficiënt de verwerkingscapaciteit benut wordt, hetgeen afhankelijk is van het besturingssysteem.

Als eerste zal de vernieuwing die de hardware ten opzichte van de IBM PC en XT heeft ondergaan, worden belicht aan de hand van de daarop gebaseerde PS/2-modellen. Daarna zal aandacht worden besteed aan het besturingssysteem.

2.2 Verwerkingsmodes

De PS/2-hardware-lijn is voornamelijk gebouwd rond de 80286 en 80386 processoren van Intel, die naast de CPU een 'Memory Management Unit' (MMU) bezitten en een nieuwe bus[1]. Eén van de taken van de MMU is, de controle op de toegangsprivileges tot het geheugen wanneer dit door een applicatie wordt geadresseerd. Dit houdt in dat slechts één applicatie toegang krijgt tot een bepaald geheugensegment. Hierdoor wordt het voor de processor mogelijk om meerdere applicaties tegelijk te verwerken (multi-tasking) omdat applicaties niet in elkaars vaarwater kunnen komen.

De Intel 80286 chip heeft een tweeledig karakter hetgeen tot uiting komt in de twee onafhankelijke modes waarin de processor applicaties kan verwerken. De 80286 kan zowel programma's in de zogeheten protected mode, als in de compatibility mode verwerken. De compatibility mode, ook wel real mode genoemd, is een omgeving waarin slechts één applicatie, geschreven onder MS-DOS 3.x, tegelijk kan draaien (single process). Deze verwerkingsmode is te vergelijken met het draaien van een MS-DOS-applicatie op een IBM PC, PC XT of AT. Alleen in protected mode is het systeem in staat om meerdere programma's tegelijk te verwerken. Het is daarentegen niet mogelijk simultaan processen in protected én compatibility mode te verwerken.

Na de 16 bit 80286 is ook de Intel 80386 chip op de markt gekomen. Deze chip bevat een 32 bit processor, datapad en adresbus.

2.3 De PS/2 -modellen

De nieuwe reeks personal computers bestaat uit een low-end-machine (Model 30) die nog gebaseerd is op de oude 8086 processor, twee mid-range-modellen gebaseerd op de 80286 (Modellen 50 en 60) en een high-end 80386-machine (Model 80). Alle nieuwe modellen hebben een 3,5 inch diskdrive en standaard hardware op het moeder-board, die de grafische beeldschermen aanstuurt. Dit in tegenstelling tot de losse kaarten die gebruikelijk waren bij de oude modellen.
Het toetsenbord, het beeldscherm en het operating system dienen apart te worden aangeschaft. Voor het eerst in de geschiedenis van IBM wordt ook een muis aangeboden die in combinatie met het window-georiënteerde OS/2 voor een groter bedieningsgemak moet zorgen. De muis is echter niet noodzakelijk voor de bediening.
De vier in Nederland verkrijgbare modellen worden aan de hand van de tabellen I en II in het kort besproken.

2.3.1 Model 30

Model 30 is bedoeld als overgangsmodel tussen de PC- en de PS/2-lijn. Dit model ziet er aan de buitenkant uit als een PS/2-machine maar de binnenkant lijkt niet op die van de overige modellen. Zo maakt Model 30 geen gebruik van een 80x86 chip maar van de oude 8086 microprocessor. Model 30 heeft geen nieuwe Micro Channel

[1] *Een bus is een set signaallijnen voor synchronisatie van en communicatie tussen systeemcomponenten.*

Architecture bus zoals de Modellen 50, 60 en 80 en is ook niet in staat om in de toekomst IBM's nieuwe operating system OS/2 te ondersteunen. Dit houdt in dat de gebruiker op een Model 30 moet werken met het welbekende DOS operating system, waardoor multitasking niet mogelijk is. Qua verwerkingssnelheid is dit model twee à drie maal zo snel als de PC XT. Dit is te danken aan zowel de hogere klokfrequentie als het twee maal zo brede datapad (16 bits). De trage harde schijf daarentegen is de achillespees van het model. Het voordeel van een IBM PC boven een Model 30 is de mogelijkheid tot het naar eigen inzicht uitbreiden van de machine door middel van extra kaarten. Model 30 is een kant en klaar produkt waar maar weinig aan toegevoegd kan worden. De voordelen van Model 30 bestaan uit de solide 3,5 inch diskdrive en de nieuwe grafische standaard die samen met de monitors voor een scherper en helderder beeld zorgt. Verwacht wordt dat Model 30 in de toekomst voorzien zal worden van de 80286 chip.

	Model 30	Model 50	Model 60	Model 80
microprocessor	8086	80286	80286	80386
klokfreq.	8 MHz	10 MHz	10 MHz	16 MHz
datapad breedte	16 bits	16 bits	16 bits	32 bits
RAM/max RAM	640/640 Kb	1/7 Mb	1/15 Mb	1/16 Mb
slots	3	4	8	8
harddisk / maximum	20/20 Mb	20/20 Mb	44/88 Mb	44/88 Mb
uitvoering	tafelmodel vloermodel	tafelmodel	vloermodel	

Tabel I. Hardware-karakteristieken van de standaard PS/2-lijn.

	Model 30	overige modellen
ROM	4 Kb ROM	128 Kb ROM
Slot type	IBM PC / XT	Micro Channel
diskette	3,5 inch 720 Kb	3,5 inch 720Kb / 1.44 Mb
grafische modes	MCGA	VGA, EGA, MCGA
OS	DOS 3.3	OS/2

Tabel II. De verschillen tussen Model 30 en de andere modellen.

2.3.2 Model 50 en Model 60

Model 50 en Model 60 zijn gebouwd rond de 80286 processor. Deze machines hebben de volgende in het oog springende eigenschappen:

- de Micro Channel-bus. Deze bus zal in een aparte paragraaf worden besproken;
- on-board support voor een nieuwe grafische standaard (VGA);
- een 1,44 Mb disk drive;
- een high-speed hard disk controller;
- 10 MHz klokfrequentie.

De Modellen 50 en 60 verschillen onderling alleen in het aantal uitbreidingsslots, de harddisk capaciteit en het uiterlijk.

2.3.3 Model 80

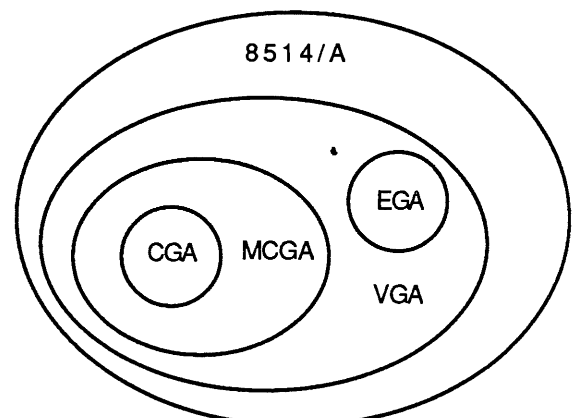
Model 80, dat gebaseerd is op de 80386 processor, heeft een 32 bit versie van de Micro Channel-bus en werkt op een klokfrequentie van 16 Mhz. Een Enhanced Small Device Interface (ESDI) zorgt voor high-speed data transport naar de harddisk (ook sommige modellen uit de 60-lijn hebben ESDI). Hierdoor hebben deze modellen een zes maal zo hoge data-transfersnelheid als die van de AT. Het topmodel van de 80-lijn, de 80-111, is een 20 MHz machine met standaard een intern geheugen van 2 Megabyte en een harde schijf van 115 Megabyte. Aangezien PC-fabrikant Apple met de Macintosh II al langer actief is op de markt van professionele PC's met een grafische interface, is het interessant om na te gaan in hoeverre Model 80 te vergelijken is met de Mac II. Het standaard Model 80 is zonder meer concurrerend te noemen daar beide machines voor hetzelfde geld (begin 1988: ongeveer f 20.000) dezelfde uitrusting bieden. Voor die prijs wordt door beide een systeem geleverd met 1Mb intern geheugen, een mathematische co-processor, kleurenscherm, toetsenbord, muis en een besturingssysteem.

2.3.4 Video-standaards

IBM heeft drie nieuwe video-standaards geïntroduceerd te weten :

- Multicolor Graphics Array (MCGA);
- Video Graphics Array (VGA);
- 8514/A Display Adapter.

Model 30 beschikt standaard over de MCGA video adapter. MCGA emuleert ook CGA, de oude IBM-standaard. De Modellen 50, 60 en 80 hebben standaard de VGA video adapter. VGA emuleert tevens de modes van de MCGA en die van de Enhanced Graphics Adapter (EGA). VGA is optioneel voor Model 30. Het 8514/A Display Adapter board ten slotte, maakt gebruik van de nieuwe bus waardoor het board alleen gebruikt kan worden op de Modellen 50 en hoger. Het board behoort echter niet tot de standaarduitrusting. Indien een gebruiker kiest voor de 8514/A dan wordt daarvoor één slot gebruikt. De 8514/A is upward compatible met VGA. In figuur 1 is het bovenstaande grafisch weergegeven.



Figuur 1: Video-standaards

Alle boards kunnen hun keuze maken uit een palet van in totaal 262.144 verschillende kleuren.

Het aantal kleuren dat **gelijktijdig** op het scherm kan worden afgebeeld ligt lager. In de onderstaande tabel zijn de gegevens betreffende de resolutie en het aantal kleuren van de verschillende standaards weergegeven.

resolutie	board	aantal kleuren
320 x 200	MCGA	4 , 256
	VGA / 8514	4 , 16 , 256
640 x 200	MCGA	2
	VGA / 8514	2 , 16
640 x 350	VGA / 8514	16,monochrome
640 x 480	MCGA	2
	VGA / 8514	2 , 16
	8514	256[2]
1024 x 768	8514	16 , 256

Tabel III. Het aantal kleuren dat simultaan op het scherm wordt weergegeven door de verschillende video-standaards, in relatie met de resolutie [STAF87].

2.3.5 Monitors

Naast de vier nieuwe modellen zijn er ook vier nieuwe monitors geïntroduceerd. De afmetingen variëren van 30 cm tot 40 cm diagonaal. De oude beeldschermen zijn niet meer te gebruiken met de PS/2-machines, omdat de nieuwe beeldschermen analoog worden aangestuurd in plaats van de tot nu toe gebruikelijke digitale aansturing. Het grootste kleurenscherm -de 40 cm 8514- is een high resolution scherm (90 beeldpunten per inch) dat het beste gebruikt kan worden in combinatie met de 8514/A Display Adapter. De 8513 met een diagonaal van 30 cm heeft een iets lagere resolutie. De 8512 heeft een beelddiameter van 35 cm met een lage resolutie. Het enige monochrome beeldscherm (de 8503) voor de PS/2-systemen meet 30 cm.

2.3.6 Micro Channel-bus

De nieuwe bus is het belangrijkste onderdeel van de Micro Channel Architecture (MCA). De Micro Channel-bus is in staat om zeer snel data te transporteren, maar daar staat tegenover dat het nieuwe ontwerp niet compatibel is met de oude PC- en AT-bus. Het Micro Channel bestaat uit een non-multiplexed bus, hetgeen wil zeggen dat data en adressen via afzonderlijke bussen getransporteerd worden. Verder bezit de bus extra lijnen voor controle, arbitrage, ondersteuning en voeding om de snelheid te verhogen. De Modellen 50 en 60 gebruiken een 16 bit brede data-bus terwijl Model 80 voorzien is van een 32-bit brede databus. Omdat MCA door IBM gepatenteerd is, zal met name dit onderdeel problemen opleveren voor de kloonmakers.

[2] *256 kleuren zijn alleen te behalen met de 8514 Memory Expansion Kit.*

2.4 Prestatie

In de onderstaande tabel wordt een snelheidsvergelijking gemaakt tussen Model 30 en een IBM PC XT en Model 50 met de IBM PC AT (alle resultaten in seconden) [PCMA87].

	XT	30	AT	50
NOP	10.1	4.2	4.17	3.36
Instruction mix	32.0	16.5	8.96	7.14
Memory test	4.8	3.2	1.32	1.00
Floating point	4.9	2.4	3.18	1.86
BIOS Disk Seek	95	82	37	81

Tabel IV. Snelheidsvergelijking tussen een aantal modellen [PCMA87].

- No Operation (NOP). De test meet de pure klokfrequentie en geheugentoegangstijden door het uitvoeren van no operation machine code instructies.
- 80x86 Instruction Mix bestaat uit een serie taken waarin de processor intensief wordt gebruikt. De tijd die benodigd was voor het uitvoeren van deze set instructies is gemeten.
- De Conventional Memory Test alloceert 256K geheugen waarna 16.384 records in het geheugen geschreven en weer uitgelezen worden.
- Floating Point Calculation meet de processorsnelheid bij het uitvoeren van een serie floating point berekeningen.
- BIOS disk seek meet de tijd die benodigd is voor een willekeurige zoekactie op de harde schijf, waarbij de ROM BIOS gebruikt wordt. Het test-programma voert 1000 zoekacties uit.

De verschillen tussen Model 50 en de PC AT zijn te verklaren door de verschillen in klokfrequentie. De AT heeft een klokfrequentie van 8 MHz terwijl Model 50 een 10 MHz klok heeft. Theoretisch is Model 50 een kwart sneller dan de AT, hetgeen door de testresultaten bevestigd wordt. De Disk seek op Model 50 is ruim twee maal zo traag als dezelfde test op de AT. De oorzaak ligt in het feit dat er een harde schijf is gebruikt met een hoge toegangstijd (80 ms in plaats van de bij de AT gebruikelijke 30 ms). Model 30 is 2 tot 2,5 maal zo snel als de XT. De disk seek is hierop de enige uitzondering. Deze is slechts weinig sneller dan die op de XT.

Voor beide nieuwe modellen geldt dus dat ze op alle fronten sneller zijn dan hun voorgangers, uitgezonderd de zoektijd op de harde schijven. Het is niet geheel duidelijk waarom IBM deze langzame opslagmedia gebruikt heeft. Waarschijnlijk zal het prijsverschil de doorslag hebben gegeven.

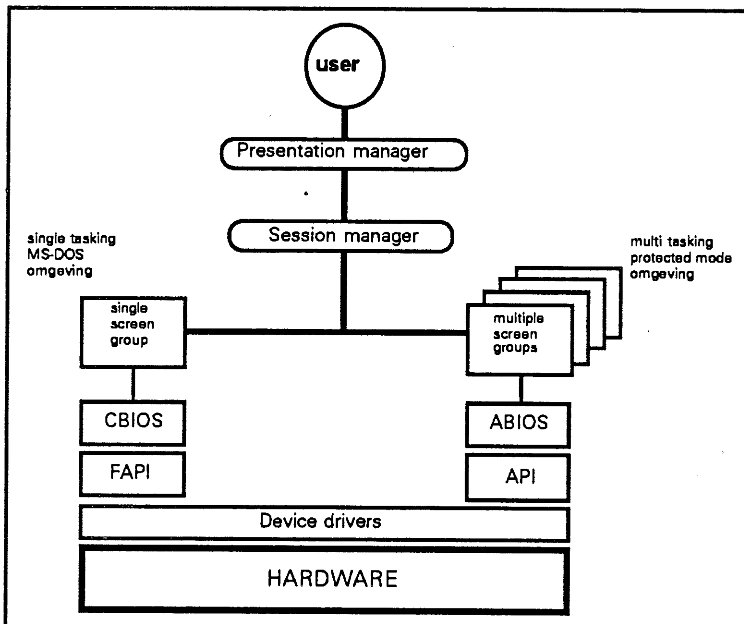
3 Operating System 2

3.1 Inleiding

Microsoft heeft in samenwerking met IBM een nieuw besturingssysteem ontworpen, dat de ca-

paciteiten van de 80286[3] processor volledig moet gaan benutten. De belangrijkste functionele uitbreiding ten opzichte van het 'oude' MS-DOS-systeem, is de mogelijkheid om de computer simultaan meerdere processen te laten verwerken (multi-tasking). Het ondersteunen van multi-tasking brengt een ingrijpende verandering van het besturingssysteem met zich mee. Daarom werd niet besloten om MS-DOS aan te passen, maar om een nieuw besturingssysteem te ontwerpen. Dit besturingssysteem is onder de naam OS/2 op de markt gekomen. De eisen die aan dit nieuwe systeem werden gesteld, zijn :

- het bieden van de mogelijkheid om meerdere processen naast elkaar te verwerken (multi-tasking);
- ondersteuning van de oude programmatuur, ontwikkeld onder MS-DOS 3.x (compatibiliteit).



Figuur 2: Het OS/2-besturingssysteem is opgebouwd uit een aantal lagen.

Niet alleen op technisch niveau is er veel veranderd, maar ook voor de gebruiker die OS/2 opstart gaat een nieuwe wereld open. De user interface wordt grafisch gepresenteerd in de vorm van de sterk in populariteit toenemende windows, iconen en het gebruik van meerdere lettertypen. Hoewel de eerste uitgave van OS/2 (Standard Edition 1.0) nog niet de beschikking heeft over deze geavanceerde grafische toepassingen, zal dit wel worden geïmplementeerd in de volgende versie (SE 1.1) die tegen het eind van 1988 wordt verwacht.
De user-interface wordt binnen OS/2 gestalte gegeven door middel van de Presentation Manager, die afgeleid is van Microsoft Windows.

[3] *Aan een 80386 versie van OS/2 wordt op dit moment nog gewerkt.*

Volgens IBM zal de Presentation Manager in behoorlijke mate verschillen van de user-interface die door Apple, Atari en Commodore wordt gebruikt.
Een latere versie van OS/2 die Extended Edition zal gaan heten heeft ook IBM's DB2 (database) en een versie van de vraagtaal Structured Query Language (SQL) aan boord. Ten slotte zal ook een communication manager worden toegevoegd, die gebaseerd is op de Systems Network Architecture (SNA), een IBM-standaard op het gebied van netwerken.

3.2 User Interface

3.2.1 De presentation manager

Om meer inzicht te krijgen in de werking van OS/2 is het goed de diverse lagen van het besturingssysteem nader te beschouwen (zie figuur 2). De presentation manager zorgt voor een gebruikersvriendelijke, grafische representatie van het besturingssysteem naar de gebruiker. Ook aan de wijze waarop de gebruiker zijn commando's aan het systeem geeft, wordt door de presentation manager vorm gegeven. Was het onder MS-DOS slechts mogelijk om via het toetsenbord commando's in te typen, de presentation manager van OS/2 ondersteunt muis-operaties in een window-structuur. Met behulp van deze user interface kan men bijvoorbeeld met de muis via de menubalk, commando's geven aan het besturingssysteem.

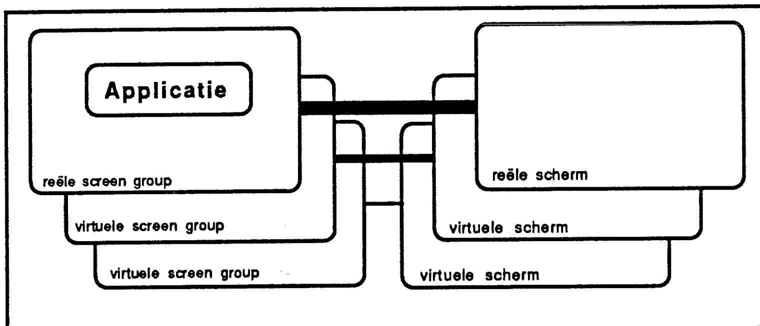
3.2.2 De session manager

Zodra een gebruiker zijn PC onder OS/2 opstart, krijgt hij te maken met de session manager (zie figuur 2). Tijdens het opstarten van een applicatie wordt door de session manager een screen group aangemaakt. Een screen group is gedefinieerd als een omgeving in protected mode, waarin slechts één applicatie kan draaien. Bij het woord applicatie moet men denken aan bijvoorbeeld een 'spreadsheet', een tekstverwerker maar ook 'desktop publishing', de MS-DOS shell, etcetera (zie figuur 3). In wezen is de compatibility mode te beschouwen als een speciale screen group die niet kan co-existeren met andere screen groups, in tegenstelling tot de protected mode, waarin meerdere screen groups gecreëerd kunnen worden. Concreet wordt per screen group het volgende door het besturingssysteem bijgehouden:

- de command processor, een omgeving waarin de gebruiker commando's aan het systeem kan geven, te vergelijken met de 'UNIX IV shell' of command.com in MS-DOS;
- een virtueel scherm geheugenbuffer;
- een virtueel toetsenbord en muis;
- 16 Mb aan virtueel geheugen.

Een applicatie die in de achtergrond draait, zet zijn uitvoer op een virtueel scherm. De gegevens op dit virtuele scherm worden zichtbaar zodra het

programma naar de voorgrond wordt gehaald. Dit betekent dat het niet mogelijk is om tegelijkertijd twee applicaties (zoals een spreadsheet en een texteditor) op de voorgrond te draaien. Het switchen tussen de diverse screen groups, wordt bestuurd door de session manager.

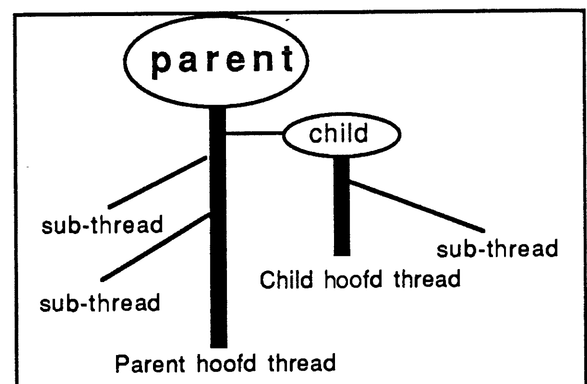


Figuur 3: Relatie tussen schermen en screen groups.

3.3 Multi-tasking binnen de OS/2-omgeving
Multi-tasking betekent voor de gebruiker dat niet op de beëindiging van een programma gewacht hoeft te worden, voordat aan een ander programma begonnen kan worden. Nadat bijvoorbeeld naast een spreadsheet-applicatie een FAT-analyse is opgestart, kan de gebruiker met de session manager een nieuw window openen om hierin tekst te verwerken. Multi-tasking vereist dat meerdere programma's -of onderdelen daarvan- in het RAM-geheugen geladen kunnen worden. Dit betekent dat het geheugen en zijn adresseerbaarheid aangepast en beveiligd moet worden. Onder MS-DOS kon maximaal slechts 640 kb direct door applicaties gealloceerd worden, in OS/2 is dit uitgebreid tot 16 Mb.
Er bestaat een aantal vormen waarin een programma op een computer verwerkt kan worden. Een systeem waarop één gebruiker tegelijkertijd meerdere processen kan draaien, wordt single-user/multi-tasking genoemd. Als de single-user/single-tasking-omgeving van MS-DOS wordt afgezet tegen de multi-user/multi-tasking-omgeving van bijvoorbeeld UNIX - denk hierbij aan een configuratie waarin meerdere terminals aangesloten zijn op één UNIX computer - dan wordt dit gat door OS/2, met zijn single-user/multi-tasking-omgeving overbrugd. De single-user/single-tasking-omgeving van de MS-DOS 3.3 (en ouder), was een direct gevolg van de 8086/8088 architectuur, die geen hardware-ondersteuning (MMU) heeft om geheugen te beschermen. In het navolgende zal een aantal begrippen, die voor multi-tasking van belang zijn, worden toegelicht.

3.3.1 Time-sharing
Multi-tasking is een techniek waarin CPU-tijd verdeeld wordt over meerdere applicaties. Het verdelen van CPU-tijd wordt door een programma verzorgd dat dispatcher wordt genoemd. OS/2 werkt met een dispatcher die met behulp van een interne klok, op prioriteitsbasis, tijdsintervallen (time-slices) aan applicaties toekent. Hoe vaak een taak een time-slice krijgt toegewezen, is ook afhankelijk van CPU- en I/O-gebruik in het verleden.

3.3.2 Threads en child-processen
De verwerkingscapaciteit van de computer in zijn geheel wordt verhoogd doordat het besturingssysteem een proces kan verdelen in één of meer verwerkingseenheden, die threads genoemd worden. Het zijn de threads waaronder de dispatcher de CPU-tijd verdeelt, niet een proces als geheel. Het creëren van meerdere threads binnen één proces werpt zijn vruchten af in die gevallen waarbij een proces gebruik maakt van I/O-devices met verschillende snelheden, zoals het toetsenbord, de harde schijf of het scherm. De threads die binnen een proces zijn gecreëerd, kunnen ten opzichte van elkaar **asynchroon** worden uitgevoerd. Dit betekent dat een proces niet in een wait-state terecht komt tijdens een I/O-call, maar slechts een thread afsplitst die de I/O verzorgt. De informatie die via een I/O-thread een proces binnenkomt kan, met behulp van een zeer snel intern communicatiemechanisme, beschikbaar komen voor andere threads of processen. Naast het feit dat een hoofdproces (parent) asynchrone threads kan aanmaken, is het ook mogelijk om een **synchroon** nevenproces af te splitsen, een zogenaamd child-proces. Tijdens executie beschikt het child-proces tevens over de mogelijkheid om één of meer threads te creëren.



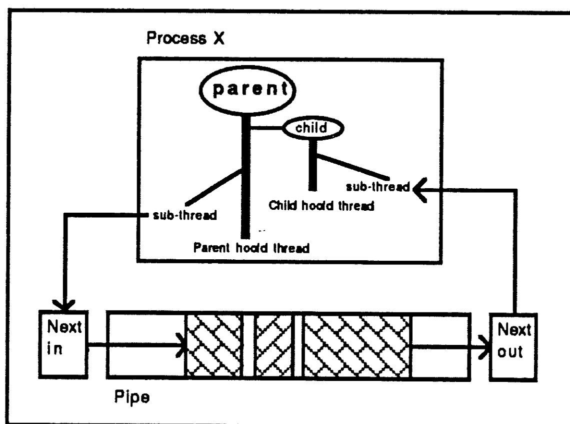
Figuur 4: Creatie van threads en een child proces binnen een hoofdproces.

3.3.3 Multi-systeememulatie
De 80386 processor is in staat om tegelijkertijd meerdere 8086/8088 microprocessoren te imiteren (multi-systeememulatie), met elk hun eigen stukje werkgeheugen. Het wordt zo mogelijk om meerdere virtuele MS-DOS-machines te installeren. Dit is een fundamenteel andere aanpak dan de multi-tasking-omgeving van OS/2.
In de toekomst ligt hierdoor het terrein open om naast MS-DOS en OS/2-systemen op de PS/2 ook andere operating systems zoals UNIX te draaien. Voor de ondersteuning van multi-systeememulatie, zal OS/2 te zijner tijd worden uitgebreid en op de markt komen onder de naam

OS/3. Het eerste besturingssysteem voor 80386 processoren is begin 1988 al door Microsoft op de markt gebracht, onder de naam MS-Windows/386.

3.4 Inter Process Communication
Threads kunnen gegevens uitwisselen met behulp van het Inter Process Communication (IPC) protocol. Als een proces een IPC-object opent, hebben alle threads- en child-processen die binnen het hoofdproces worden gecreëerd ook automatisch toegang tot dit communicatie-object. De objects die het besturingssysteem voor deze communicatiedoeleinden beschikbaar heeft, zijn semaforen, pipes en queues.

3.4.1 Semaforen
Semaforen zijn zeer snel toegankelijke geheugenvelden die maar door één thread tegelijk geadresseerd kunnen worden. Er is dus hooguit één proces in het bezit van een bepaalde semafoor. Het zal duidelijk zijn dat semaforen gebruikt worden voor locking-doeleinden. Als een proces bijvoorbeeld het printer-semafoor in zijn bezit heeft, wordt voorkomen dat meerdere processen tegelijkertijd op één printer aan het printen zijn. Er zijn twee soorten semaforen: RAM-semaforen, voor de controle op de communicatie tussen threads binnen één proces en systeem-semaforen, die zorg dragen voor de controle op de communicatie tussen processen onderling.



Figuur 5: Voorbeeld van communicatie via een pipe.

3.4.2 Pipes
Een pipe is een stuk geheugen dat uitwisseling van data mogelijk maakt tussen twee nauw met elkaar in verband staande processen, bijvoorbeeld parent/child. Een pipe heeft geen naam waardoor de pipe buiten het proces niet te adresseren is (Figuur 5). De gearceerde gebieden geven verschillende data-velden aan die door proces X worden geschreven en gelezen volgens het FIFO-principe. Nadat proces X een data-veld heeft ingelezen, kunnen threads van proces X van dit geheugen gebruik maken. De maximale grootte van een pipe is 64K. De IBM-pipes zijn vrijwel gelijk aan de bekende UNIX-pipes. Een pipe binnen OS/2 kan door meerdere threads worden geopend om te lezen en te schrijven. Om toegang te krijgen tot een pipe, hoeven de threads van een proces niet op elkaar te wachten. De rangschikking van data gaat volgens het FIFO-principe.

3.4.3 Queues
De queue wordt gebruikt voor de communicatie tussen **hoofd**processen. Naar een queue kunnen meerdere processen schrijven, maar alleen het proces dat de queue creëerde, kan er uit lezen. Queues zijn het meest flexibel in gebruik omdat ze de volgende eigenschappen hebben:

- een queue heeft een naam waardoor de queue door elk hoofdproces gebruikt kan worden;
- de grootte van de queue wordt bepaald door de fysieke vrije geheugenruimte;
- een queue bestaat uit records (geheugen blokken met een maximale grootte van 64 Kb);
- de records binnen een queue worden gerangschikt volgens FIFO, LIFO of op prioriteitsbasis.

De toegangssnelheid tot de communicatie-objecten neemt **af** volgens: semaforen -> pipes -> queues.

3.5 Systems Application Architecture
De Systems Application Architecture (SAA) wordt door IBM gedefinieerd als een gemeenschappelijke user-, software- en communicatie-interface voor IBM-mainframes, mini's en micro's. IBM wil dat programma's die op verschillende hardware-lijnen, onder verschillende operating systems draaien, er hetzelfde uitzien voor zowel gebruikers als programmeurs en gebruik kunnen maken van dezelfde communicatiekanalen (Figuur 6.). Hierdoor wordt op de volgende terreinen naar een grotere consistentie gestreefd :

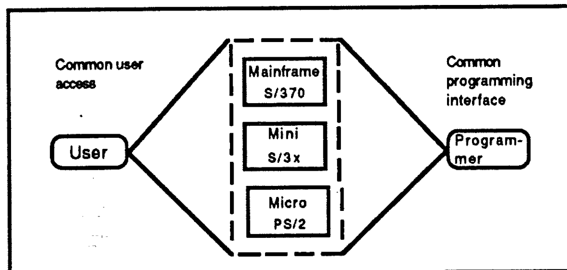
- programmer-interface (de talen, commando's, hulpmiddelen en aanroepen die door een programmeur gebruikt worden);
- user interface (de communicatie tussen de gebruiker en de applicatie);
- communicatie (verbindingen tussen applicaties, systemen, netwerken en randapparatuur);
- applicaties (software geschreven door IBM en derden).

De voordelen voor de klanten van IBM zijn de volgende:

- programmeerervaring is veel algemener toepasbaar;
- applicaties zijn eenvoudig op andere machines te zetten (portable);
- de user interface van deze applicaties is eenvoudiger en consistenter.

Het uiteindelijke effect is volgens IBM dat de ontwikkelingskosten en trainingskosten lager worden.

De Modellen 50, 60 en 80 gaan onderdeel uitmaken van SAA. Dit houdt in dat software, geschreven op deze machines volgens de SAA-standaard onder OS/2, ook draait op de toekomstige micro-, mini- en mainframe-computers van IBM.
Op dit moment is de SAA slechts in zeer algemene bewoordingen omschreven. Het zal waarschijnlijk nog geruime tijd duren voordat het ambitieuze SAA-plan in al zijn aspecten is doorgevoerd.



Figuur 6: Het moet voor zowel de gebruiker als de programmeur niet uitmaken op welk systeem hij is aangesloten.

3.6 Compatibiliteit/portabiliteit

In welke mate moet de bestaande programmatuur, die draait onder MS-DOS, worden aangepast om gebruikt te kunnen worden in een OS/2-omgeving?

In principe is het voldoende dat een MS-DOS-applicatie onder OS/2 opnieuw wordt gecompileerd. Naast pure applicatiecode, geschreven in bijvoorbeeld C of Pascal, maakt een programma echter ook vaak gebruik van specifieke systeemroutines, die taken afhandelen zoals het tekenen van windows, het alloceren van geheugen of het aansturen van device drivers. OS/2-programmatuur kan daartoe gebruik maken van een uitgebreide set systeemroutines, onder de naam: Application Programming Interface (API). Om aan de portabiliteit van 'oude' MS-DOS-software tegemoet te komen, heeft Microsoft van de API een subset afgesplitst, die de naam Family Application Programming Interface (FAPI) heeft gekregen. De functionaliteit van de FAPI komt overeen met die van de MS-DOS 3.x systeemroutines. Deze routines zijn qua functionaliteit onder te verdelen in een aantal groepen die de gebruiker kent als managers.
De communicatie tussen het systeem en de applicatie wordt geregeld door het Basic I/O System (BIOS). Afhankelijk van de omgeving waarin de applicatie draait (compatibility mode of protected mode) worden systeemfuncties geadresseerd via respectievelijk het CBIOS, of het ABIOS. IBM heeft zeer weinig aan de oude BIOS gewijzigd. De CBIOS van OS/2 is compatibel met de BIOS in MS-DOS, doordat CBIOS een superset is van de standaard PC BIOS. Dit houdt in dat alle programma's die aanroepen bevatten naar de oude BIOS-routines daarbij ook op de PS/2-systemen geen problemen zullen ondervinden.

Model 30 heeft geen C- of A-BIOS maar een BIOS-versie die compatibel is met het oude PC BIOS. In protected mode kan de besturing van de hardware niet meer door een applicatie[4] worden verzorgd, maar moet door het besturingssysteem worden geregeld. Een multi-tasking-omgeving vereist dat allocatie van hardware en system resources door het besturingssysteem gedaan kan worden in plaats van door de applicatie zelf.
Samenvattend kunnen vier code-typen onder OS/2 draaien:

- programmatuur gemaakt onder MS-DOS 3.x;
- FAPI-programma's in compatibility mode;
- FAPI-programma's in protected mode;
- API-programma's (nieuwe software die alleen draait in OS/2 protected mode).

3.7 Dynamic Linking (Dynalinks)

Onder MS-DOS was gebruikelijk dat een gehele library tijdens het linken permanent aan de betreffende applicatie werd toegevoegd. In OS/2 worden alleen library routines, tijdens **run-time**, dynamisch aan een applicatie gelinked. Hiervoor is een nieuwe compiler ontworpen die code genereert, die gebruikt kan worden door een dynamic linker. De programmeur moet hiervoor wel een aanroep naar een specifieke systeemroutine uit een bibliotheek doen. De bibliotheekroutines die in OS/2 aan een applicatie gelinked zijn, kunnen ook door andere applicaties worden aangeroepen (sharing), en na gebruik worden vrijgegeven.
Ook het besturingssysteem werkt met dynalinks, zodat niet alle systeemfuncties continu in het RAM-geheugen hoeven te staan. Daarnaast is het systeem relatief eenvoudig uit te breiden door systeemroutines toe te voegen, zoals bijvoorbeeld device drivers. Als nadeel kan worden opgemerkt dat run-time linking van library-routines CPU-tijd kost en daarmee de verwerkingssnelheid vertraagt.

3.8 Datacommunicatie

Tegenwoordig wordt er steeds meer nadruk opgelegd dat een systeem aangesloten kan worden op een ander systeem (connectivity). Met name de mogelijkheid tot aansluiting op een Local Area Network (LAN) krijgt veel aandacht. De OS/2 Communications Manager, die zal worden opgenomen in de extended edition van het besturingssysteem, is in staat om gelijktijdig te communiceren met verschillende computersystemen, waarbij de verbindingen zowel lokaal

[4] *Veel MS-DOS-programmeurs maken regelmatig gebruik van code-shortcuts om de prestatie te verhogen, hetgeen ten koste gaat van de portabiliteit.*

kunnen zijn (IBM NetBios, Token-ring), als op afstand via datacommunicatie.
Dit biedt onder meer de mogelijkheid om een proces op een andere machine te laten draaien (distributed processing), hetgeen een enorme vergroting van de verwerkingscapaciteit oplevert.

4 Toekomst

IBM heeft een standaard Intel chip gebruikt zodat andere PC-leveranciers op dit punt compatibel kunnen zijn met IBM. Ook de BIOS entry points en Micro Channel-signalen zijn vrijgegeven, zodat software- en kaartontwikkelaars hun gang kunnen gaan. De rest van het systeem is moeilijk te kopiëren, met name de besturing van de bus.

4.1 Klonen

Het lijkt erop dat de technologische barrières omtrent het kopiëren van de hardware geslecht zijn. De meeste producenten (waaronder Kaypro, Tandy en Compaq) hebben al een prototype van een kloon gereed maar niemand heeft er al één op de markt gebracht, compleet met de Micro Channel-architectuur. Kaypro heeft meegedeeld dat ze eind mei met een Micro Channel compatible machine op de markt zou komen. Het bedrijf Chips & Technologies heeft een reeks chips aangekondigd, waarmee kloonmakers machines in elkaar kunnen zetten die 100% compatible zijn met Modellen 50, 60 en 80. De kopieën bestaan uit veel minder componenten waardoor ze een stuk compacter kunnen zijn dan de IBM-modellen.
Het zou de angst voor slepende en geldverslindende rechtszaken zijn die tot nu toe alle ervan weerhoudt om de klonen ook daadwerkelijk op de markt te brengen. IBM heeft namelijk gedreigd dat het namaken van PS/2 niet getolereerd wordt. Een andere belangrijke reden kan zijn dat de verkoop van PS/2-machines commercieel gezien nog niet interessant genoeg is. De Nederlandse PC-fabrikant Tulip geeft dit als belangrijkste reden op om de uitgestelde introductie van de PS/2-compatibles te verklaren.

4.2 Octrooiproces van Apple tegen Microsoft

Op dit ogenblik heeft Apple een proces aangespannen tegen MicroSoft en Hewlett Packard, waarin geëist wordt dat MS Windows versie 2.03 en HP's New Wave van de markt worden gehaald omdat ze teveel op de Macintosh user interface zouden lijken. Aangezien Windows 2.03 de basis moet gaan vormen voor de Presentation Manager van IBM's OS/2, heeft de uitkomst van dit proces grote gevolgen voor de software-ontwikkeling. Er zijn nu nog bijzonder weinig applicaties voor OS/2 (op dit moment zijn er zo'n 150, aan het eind van dit jaar moet dat aantal zijn opgelopen tot 1000) en een vertraging van de introductie van de Presentation Manager zou ertoe kunnen leiden dat veel software-ontwikkelaars projecten stopzetten. In een reactie op het proces hebben de meeste software-bureaus echter verklaard, door te gaan met het ontwikkelen van toepassingen. Alleen Borland heeft de ontwikkeling van een Windows versie van Paradox DBMS stopgezet. IBM kan, indien het proces wordt verloren, op drie manieren proberen omzetverlies te beperken. De eerste oplossing betreft het zodanig wijzigen van de Presentation Manager dat geen inbreuk meer wordt gemaakt op het copyright van Apple. De tweede en derde oplossing betreffen het kopen van een licentie van hetzij Xerox (waar de basis van Apple's user interface is gelegd) hetzij van Apple zelf.

4.3 Verkrijgbaarheid

Op het moment van dit schrijven zijn alle besproken hardware-modellen verkrijgbaar. Het schijnt dat er nieuwe PS/2-modellen op komst zijn, zoals twee schootcomputers en een Model 25. Of de nieuwe modellen in Nederland leverbaar zijn is onbekend. OS/2 is nu alleen nog te leveren als Standaard Edition 1.0. Deze versie vertraagt MS-DOS-programma's tot zo'n 20% en draait niet altijd probleemloos op IBM-klonen. Afgezien van de relatief hoge prijs kan als nadeel nog worden aangemerkt dat het besturingssysteem omvangrijk is. Een machine met 1,5 Mb aan intern geheugen is vereist (waaraan geheugenruimte voor applicaties nog moet worden toegevoegd).

4.4 OS/2 voor de 80386 processor

Aangezien OS/2 niet speciaal voor de 386 maar voor de 286 processor is geschreven, bestaat de mogelijkheid dat besturingssystemen die wel speciaal voor de 386 zijn geschreven, zoals het besturingssysteem van Apple Macintosh en UNIX/386, het roer over kunnen gaan nemen. MicroSoft is om dit te voorkomen al begonnen aan een versie die speciaal geschikt is voor de 80386. Deze versie moet tegen het eind van 1989 beschikbaar zijn.

4.5 Prestaties

Begin 1988 is er een serie benchmarks[5] uitgevoerd, waarbij OS/2 werd vergeleken met XENIX[6]. Hieruit bleek dat het snelheidsverschil tussen OS/2 en XENIX is te verwaarlozen zolang de versie van XENIX speciaal geschreven is voor de 286. Als dezelfde tests worden uitgevoerd bij een XENIX-versie voor de 386, dan is OS/2 ongeveer 7 maal zo traag.

4.6 Toekomstverwachtingen

Voorlopig kan slechts gespeculeerd worden over de mogelijkheden en prestaties van OS/2. De

[5] *De benchmarks werden uitgevoerd door Neal Nelson & Associates uit Chicago. Bron: Computable 15 april 1988.*

[6] *Xenix is een PC-versie van UNIX.*

afwezigheid van de Presentation Manager heeft tot gevolg dat de modellen die nu verkrijgbaar zijn, eigenlijk niets meer zijn dan opgevoerde IBM PC's. De algemene mening is dan ook dat OS/2, zeker nu de Presentation Manager er nog niet is, slechts langzaam in populariteit zal toenemen.
Kwantitatieve analyses over de verwerkingscapaciteit van de multi-tasking - protected mode - omgeving, kunnen pas gemaakt worden als er programmatuur op de markt komt die expliciet voor deze verwerkingsmode geschreven is.
Het ziet ernaar uit dat de komende tijd MS-DOS 3.x het meest gebruikte besturingssysteem in de IBM PC-wereld zal blijven.

5 Personalia

Dirk Witte (26) is sinds 31 augustus 1987 werkzaam bij de sectie Software Engineering van KPMG Klynveld EDP Audit. Hij is afgestudeerd in de richting Experimentele Kernfysica aan de Vrije Universiteit te Amsterdam.
Jeroen de Graaff (23) werkt sinds september 1987 als software engineer bij KPMG Klynveld EDP Audit. Hij heeft Industrieel Ontwerpen aan de TU Delft gestudeerd.

6 Literatuur

[STAF87] Byte Staff: The IBM PS/2 Computers, Byte June 1987 (pag. 100 e.v.).
[WHIT87] White, E, Grehan, R: Microsoft's New DOS, Byte june 1987 (pag. 116 e.v.).
[FRAN87] Franklin, C, Grehan, R: The IBM PS/2 model 80. Byte november 1987 (pag. 143 e.v.).
[BYTE87] Byte december 1987, extra edition.
[CORT87] Cortesi., DE: Dynamic Linking in OS/2. Dr.Dobb's Journal of Software Tools, december 1987 .
[MICR87] Microsofts Systems Journal, may 1987, volume 2 number 2.
[DUYM87] Duijm, MC: IBM's PS/2 serie microcomputers, Microbe december 1987 (pag. 2 e.v.).
[PCMA87] First Looks, diverse auteurs. PC Magazine may 1987.
[IBMO] IBM OS/2 Information and Planning Guide.

Elektronisch betalen, de betaalpas

door: Ing. J. Rotteveel

1 Inleiding

Er heerst nogal wat beroering op het gebied van het elektronisch betalen in Nederland. Het gebruik van magneetkaarten als betaalpassen zou niet betrouwbaar zijn. Gelijktijdig wordt de chipkaart als oplossing voor deze problematiek gepresenteerd.
In dit artikel zal kort worden ingegaan op de in de media geschetste problematiek van het elektronisch betalen en zullen mogelijke oplossingen besproken worden.

2 Elektronisch betalen

In de huidige situatie wordt gebruik gemaakt van magneetkaarten waarop de betalingsgegevens van de klant zijn vastgelegd. Identificatie vindt plaats door middel van de betaalpas en authenticatie c.q. legitimatie van de klant door middel van de PIN-code [NEIS87], [DUIN88].
Het blijkt mogelijk om zowel de identificatie- als de authenticatiegegevens te bemachtigen. Wanneer een pomphouder of winkelier een eigen nep-betaalautomaat plaatst, kan de beschikking verkregen worden over de gegevens op de magneetstrip en de bijbehorende PIN-code. De gegevens van de bankpas kunnen op een andere magneetkaart gekopieerd worden, waarna de kopie-bankpas te zamen met de verkregen PIN-code, gebruikt kan worden om geld te verkrijgen uit een geldautomaat.
Een eenvoudig maar in theorie werkend scenario, waarvan in de media uitgebreid aandacht is besteed.

3 Problemen huidige situatie

In wezen is het feit dat de gegevens uit de magneetstrip en de PIN-code gemeenschappelijk voor de betaalautomaat en de geldautomaat wordt gehanteerd, de sleutel voor het uitvoeren van het boven geschetst scenario.

De volgende eigenschappen van het huidige elektronisch betalingsconcept zijn de oorzaken van het falen van het systeem:

- De magneetkaart is een passief medium en herkent daardoor de terminal niet. De magneetkaart heeft niet de mogelijkheid om te onderzoeken in wat voor soort terminal c.q. kaartlezer hij zit. Hij kan geen onderscheid maken tussen een officiële betaalautomaat en een eenvoudige magneetkaartlezer van iemand met frauduleuze bedoelingen.
- De magneetkaart is niet in staat een informatiestop af te dwingen aan de terminal. Wanneer de magneetkaart een frauduleuze terminal zou kunnen herkennen, dan nog heeft de bankpas geen mogelijkheid om de terminal ervan te weerhouden zijn magneetstrip uit te lezen.
- Het is mogelijk om magneetkaartinformatie te kopiëren naar een andere magneetkaart. Apparatuur hiervoor is commercieel verkrijgbaar en betaalbaar.
- De terminal herkent de magneetkaart niet. Wanneer een kopie-betaalpas aan een betaalautomaat of geldautomaat wordt aangeboden, kan door de terminal niet achterhaald worden of het een frauduleuze kopie-betaalpas betreft of een valide[1] betaalpas.

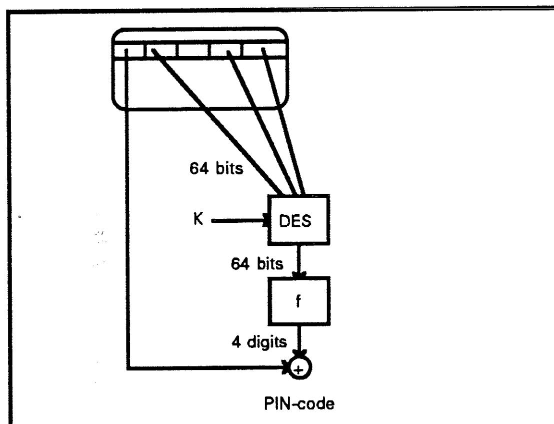
Het is in prinicipe niet noodzakelijk om voor alle punten een oplossing te vinden om een veilig concept te verkrijgen. Het is echter reeds met toepasbare middelen mogelijk. In het navolgende zal beschreven worden hoe met de magneetkaart oplossingen mogelijk zouden kunnen zijn. Daarnaast zal worden ingegegaan op de chipkaart die, wanneer op de juiste wijze gebruikt, in staat is een oplossing te bieden voor alle vier de problemen van de magneetkaart.

4 Oplossingen voor de magneetkaart

In de vorige paragraaf zijn vier oorzaken onderscheiden voor het falen van het systeem. Echter wanneer de laatste oorzaak, het niet herkennen van de kopie-pas, zou kunnen worden voorkomen, is het geschetste scenario niet meer toepasbaar. Een fraudeur is dan namelijk alleen maar in staat een kopie-pas aan te maken die hij nergens kan gebruiken omdat de terminal de kopie-pas als niet-valide herkent.
De terminal is in staat de magneetkaart te herkennen wanneer iedere magneetkaart een uniek kenmerk heeft. Wanneer een niet uniek kenmerk gebruikt zou worden, zou een andere valide pas als kopie-pas kunnen worden gebruikt.
Om dit te voorkomen dient een relatie gelegd te worden tussen het unieke kenmerk op de magneetkaart en de gegevens op de magneetstrip.
Deze relatie is op verschillende manieren te leggen maar het ligt het meest voor de hand om de PIN-code hiervoor te gebruiken.
In het huidige systeem wordt namelijk al de PIN-code afgeleid van enkele gegevens op de magneetstrip [NEIS87]. Dit kan schematisch als in figuur 1 worden weergegeven.
We zien dat enkele velden uit de magneetstrip zoals rekeningnummer, pasvolgnummer en

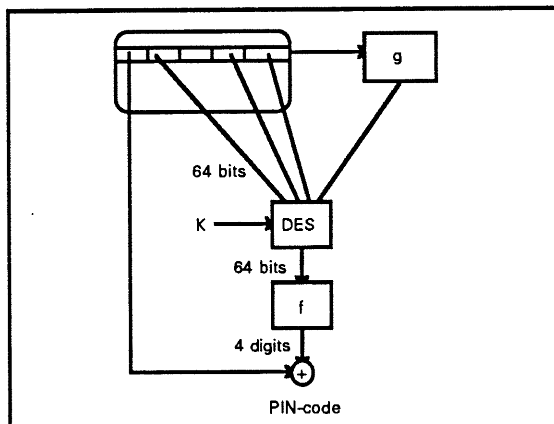
[1] *Met valide wordt in de context van dit schrijven bedoeld: Een door de kaartuitgevende instelling uitgegeven pas met officieel weggeschreven gegevens op de magneetkaart.*

dergelijke worden geëncrypt met een sleutel onder het DES[2] algoritme. Dit levert eveneens 64 bits op die vervolgens na de functie f, zijnde een selectie en het toepassen van een vertaaltabel, vier cijfers oplevert. Hierbij wordt dan een offset uit de magneetstrip opgeteld waarna de PIN-code bepaald is.



Figuur 1: PIN-code huidige systeem.

Wanneer de magneetkaart een uniek kenmerk heeft, vindt koppeling tussen de gegevens op de magneetstrip en het kenmerk als in figuur 2 plaats.
Met behulp van de functie g wordt het kenmerk, dat zich op of in de kaart bevindt, vertaald naar een bijbehorend getal.
Dit getal wordt ook gebruikt als input van de DES-functie, waardoor de uiteindelijk berekende PIN-code dus ook afhankelijk is van het kenmerk van de magneetkaart.



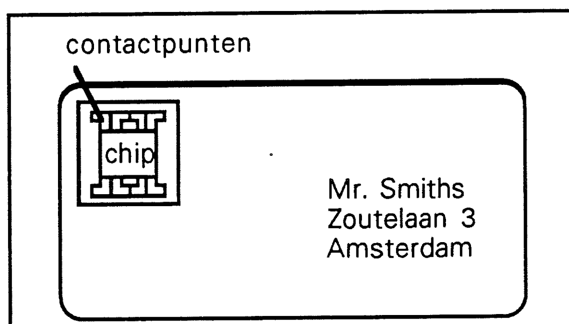
Figuur 2: PIN-code met kenmerk.

4.1 Kenmerken

Natuurlijk zijn er legio andere merkingen te gebruiken. Zo wordt er in België aan gedacht om bij de fabricage van passen een stuk ijzerdraad mee te 'roeren'. Hierdoor levert iedere pas een uniek magnetisch veld. In het Duitse elektronisch betalingsverkeer wordt gebruik gemaakt van magneetkaarten die, na een chemisch bad, een uniek patroon krijgen.
Naast het merken van bankpassen zijn er ook nog andere mogelijkheden die het beschreven scenario bemoeilijken. Zo is het mogelijk om, door een bepaalde manier van magnetiseren, twee gegevenslagen op een magneetkaart te kunnen lezen en schrijven. Dit wordt het 'dual-stripe'-principe genoemd. Door het toepassen van een dual-stripe-manier van magnetiseren, wordt dus in wezen probleem 3, het kunnen kopiëren van een magneetkaart bemoeilijkt omdat apparatuur hiervoor moeilijker te verkrijgen is.
Een andere oplossing is het kiezen van een actief medium zoals de chipkaart.

5 De chipkaart

Alvorens in te gaan op de voordelen van de chipkaart een korte introductie.
De chipkaart is een plastic kaartje waarin een chip is geplaatst. Aan de chip zit een aantal contactpunten; waardoor communicatie met de buitenwereld mogelijk is.



Figuur 3: Chipkaart

Er zijn drie soorten chipkaarten:

- Geheugenkaart, de gebruikte chip functioneert slechts als geheugen.
- Beperkt intelligente chipkaart, de chipkaart heeft slechts een beperkte functionaliteit.
- Intelligente chipkaart, ook wel smartcard genoemd. Kenmerkend voor een smartcard is de aanwezigheid van een complete processor. Er draait een applicatie op de chipkaart. Communicatie met de buitenwereld vindt via die applicatie plaats.

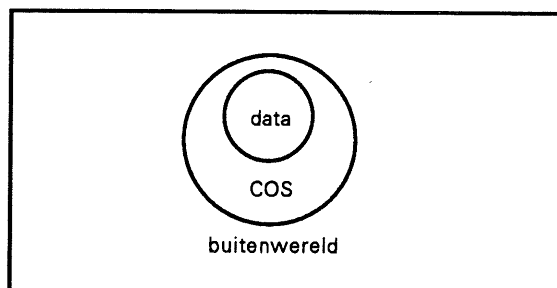
Een voorbeeld van de geheugenkaart is een debet-kaart zoals de telefoonkaart, die in Frankrijk wordt gebruikt voor de openbare telefooncellen. Op de geheugenkaart wordt bijgehouden hoeveel eenheden verbruikt zijn. Het geheugen bestaat uit EPROM[3]. Initieel is het geheugen gevuld met

[2] *DES staat voor Data Encryption Standard en is door het National Bureau of Standards in de Verenigde Staten vastgesteld tot encryptie standaard [MEYE82].*

[3] *EPROM, Erasable Programmable Read Only Memory. Gegevens kunnen door een programma in het geheugen worden geplaatst en gelezen. De gegevens*

enen waarbij iedere één overeenkomt met een telefooneenheid. Na iedere telefoonperiode wordt een één vervangen door een nul. Een nul is echter alleen in een één te wijzigen door het gehele geheugen te wissen.

Om echter te voorkomen dat men zelf een debetkaart kan wissen, zodat de kaart opnieuw gebruikt kan worden, is een bepaalde lokatie in het geheugen gevuld met een 'handtekening', een bepaald getal. De 'write-enable'-lijn van de geheugenplaatsen voor de handtekening wordt, na het zetten van de handtekening, doorgebrand zodat deze niet meer te beschrijven zijn.



Figuur 4: Schil-principe chipkaart.

Wanneer dus het hele geheugen wordt gewist, brengt dit met zich mee dat ook de handtekening gewist is en de debet-kaart zal niet meer door de telefooncellen worden geaccepteerd.
De smartcard bevat een processor en een applicatie die in ROM[4] is gebakken, een databank in EPROM en een werkgeheugen voor berekeningen e.d. in RAM[5].

Op de chipkaart draait een applicatie, deze wordt ook wel het Card Operating System (COS) genoemd. Het COS moet gezien worden als een schil om de databank. Alle handelingen vinden plaats via het COS. Het COS heeft dus een uitvoerende taak maar is in staat controle uit te oefenen op alle handelingen.
Het COS is verdeeld in drie groepen functies:
- File management;
- Cryptografische functies;
- Random getal generatie.

<u>File management</u>
Met behulp van de file management-functies is het mogelijk een file-structuur op te bouwen. Gedacht moet worden aan functies zoals het openen, creëren van directories en bestanden en het lezen en schrijven van bestanden. Gegevens komen beschikbaar door het uitvoeren van leesinstructies op de files. Alleen via de file management-functies is toegang tot de gegevens mogelijk. De belangrijkste functie is de access control-mogelijkheid. Een leescommando van de buitenwereld wordt alleen gehonoreerd als de buitenwereld bevoegd is die gegevens te lezen. Dit kan bijvoorbeeld vergeleken worden met de access control op een mainframe.

<u>Cryptografische functies</u>
Met behulp van deze functies kan encryptie en decryptie plaatsvinden. Gedacht moet worden aan DES of RSA[6] als encryptie-methode. De sleutelgegevens zijn in een sleutelfile opgenomen. Doordat alleen de cryptografische functies geautoriseerd zijn om de sleutel-file te benaderen, is het voor de buitenwereld niet mogelijk de sleutelgegevens uit te lezen.

<u>Random getal generatie</u>
Binnen het Card Operating System is de mogelijkheid aanwezig voor het genereren van een random getal. Dit vindt plaats door het periodiek toepassen van een random-functie. Dit is veelal een functie in de trant van $y_{i+1} = a * y_i + b$, waarbij a en b dusdanig zijn gekozen dat een goede spreiding verkregen wordt. Daar de functie steeds uitgaat van een vorig berekende randomwaarde, moet initieel gestart worden met een startwaarde. Bij de chipkaart wordt veelal als startwaarde de inhoud van een lokatie in het RAM-geheugen gebruikt op het moment dat de chipkaart in de terminal wordt gestopt.
De tijdseenheid is afhankelijk per chipkaart. Iedere chip heeft dus een onbekende en verschilllende tijdseenheid die gebruikt wordt voor het bepalen van een random getal. De tijdseenheid ligt ergens in de buurt van de 10 ms.
Doordat de startwaarde niet bekend is en de tijd tussen iedere generatie niet te achterhalen is, is de waarde van ieder random getal niet voorspelbaar.

Samenvattend kan de chipkaart gezien worden als een medium met de volgende eigenschappen:
- gecontroleerde informatieverstrekking;
- encryptie-mogelijkheden;
- mogelijkheid tot het genereren van een random getal.

5.4 Chipkaart versus hiaten
Met behulp van deze eigenschappen is het mogelijk oplossingen te creëren voor de vier probleemgebieden van de huidige opzet van het elektronisch betalingsverkeer.

blijven in het geheugen staan als de spanning verdwijnt. Uitwissen kan alleen van het volledige geheugen d.m.v. ultra-violet-licht.

[4] *ROM, Read Only Memory. De inhoud moet bekend zijn bij het bakken van de chip.*

[5] *RAM, Random Access Memory. Ook wel volatile of vluchtig geheugen genoemd. Gegevens kunnen door een programma willekeurig in het geheugen worden geschreven en gelezen. Zodra de spanning weg is, zijn ook de gegevens weg.*

[6] *RSA, Rivest Shamir en Adleman, de ontwerpers van een 'public key'-crypto-algoritme [MEYE82].*

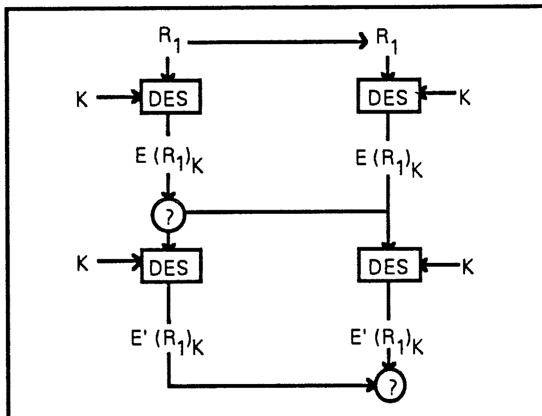
Herkennen
Het wederzijds herkennen is met de chipkaart mogelijk door authenticatie wat in de volgende paragraaf aan de orde zal komen.

Informatiestop
Doordat alle gegevens afgeschermd zijn door het Card Operating System met access control-faciliteiten, is het mogelijk dat de chipkaart besluit geen informatie aan de buitenwereld af te staan. Hierdoor is een informatiestop dus mogelijk.

Kopiëren
Het kopiëren van een chipkaart houdt in dat de masters waarmee de chip gebakken is, dienen te worden nagemaakt. Dit is een complex proces.

6 Authenticatie

In [DUIN88] werd al kort ingegaan op het twee-chipkaarten-principe, het authenticeren van twee partijen door middel van twee chipkaarten. In deze paragraaf zal dieper ingegaan worden op te hanteren protocollen gegeven de genoemde chipkaart-eigenschappen.



Figuur 5: Authenticatie-protocol.

Authenticatie vindt plaats doordat bepaald wordt of zowel de chipkaart van de betaalautomaat als de chipkaart van de klant valide zijn. Hiertoe wordt onderzocht of beide partijen de beschikking hebben over een bepaalde geheime sleutel. Hiervoor kan het 'handshake'-protocol worden gebruikt [KERS87] dat weergegeven is in figuur 5.
Partij A genereert een willekeurig getal R_1, versleutelt het getal met een door beide partijen afgesproken sleutel en verstuurt het vercijferde getal $E(R_1)$ naar partij B. Partij B had al het random getal gekregen, heeft deze ook versleuteld en is nu in staat om te vergelijken of dit overeenkomt met hetgeen door partij A is verzonden.
Op dit moment heeft partij B zekerheid over de authenticiteit van partij A, indien deze correct is, kan het protocol vervolgd worden.

Partij B versleutelt het berekende getal wederom en verstuurd dit getal naar partij A, E'(R1). Partij A had al het getal na één versleuteling, versleutelt deze nog een keer en kan deze uitkomst vergelijken met het door partij B verzonden getal.
Daarna heeft ook partij A zekerheid over de authenticiteit van partij B.
De terminal is dus in staat te bepalen of de betaalpas van de klant valide is. Wanneer dit niet zo is, kan door de terminal de gewenste transactie worden geweigerd.
De betaalpas van de klant is echter ook in staat te bepalen of de terminal valide is. Aangezien de chipkaart een gecontroleerde informatieverstrekking heeft, kan ingeval van een niet-valide terminal, een informatiestop worden afgedwongen.

7 Conclusie

Dat de huidige opzet van het elektronische betalingsverkeer niet optimaal beveiligd is, was reeds in de media aangetoond. Door het toepassen van identificatie van de betaalpas is de beveiliging te verbeteren. Nadeel hiervan is dat dit technisch nog moeilijk te realiseren is.

De chipkaart biedt de mogelijkheid om oplossingen te geven voor de huidige hiaten.
De beveiliging bij de chipkaart is niet alleen gebaseerd op een complexer fabricageproces maar ook op niet-fabricage-aspecten door de intrede van beveiligde geheime sleutels in de chipkaart.

In hoeverre het namaken van chipkaarten mogelijk is, is moeilijk te zeggen. Mogelijkheden op dit terrein ontwikkelen zich meer en meer, mede door de ontwikkeling van faciliteiten op het gebied van de chip-verificatie. Hierbij worden röntgenstralen en optische microscopen gebruikt voor het zichtbaar maken van de chip-layout.

Of de steeds groeiende technologische ontwikkeling in staat zal zijn om in de toekomst de afgeschermde informatie uit een chipkaart te bemachtigen, is eveneens moeilijk in te schatten.

Ten behoeve van de beveiliging van de chipkaart dient dan ook gestreefd te worden naar een beveiligingsconcept met unieke sleutels per chipkaart en per toepassing, zodat analyse van de geheime gegevens niet lonend is maar beperkt tot één chipkaart.

8 Personalia

Jaco Rotteveel is sinds juni 1986 werkzaam bij KPMG Klynveld EDP Audit. In het kader van zijn studie aan de HTS-Informatica te Eindhoven, heeft hij zich bij het afstuderen beziggehouden met Artificial Intelligence. Zowel stage- als

afstudeerwerkzaamheden hebben plaatsgevonden bij KPMG Klynveld EDP Audit. Momenteel werkt hij bij de sectie Software Engineering die zich bezighoudt met research en het ontwikkelen van audit-programmatuur. Tevens wordt hij ingezet bij audits van de EDP Audit-sectie.

9 Literatuur

[DUIN88] Duin, IM van: Elektronisch Funds Transfer, het elektronisch uitvoeren van betalingen, literatuurstudie. Compact 88/1 winter 1988.

[KERS87] Kersten, AG: Smart cards for POS-banking. Siemens AG, 1987.

[MEYE82] Meyer, CH, Matyas, SM: A new dimension in computer data security cryptography.

[NEIS87] Neisingh, AW: Consequenties voor de beheersbaarheid ten gevolge van nieuwe technologische ontwikkelingen. Compact zomer 1987.

BOEKEN

'Odyssee, van Pepsi naar Apple' door J. Sculley
486 pagina's, ISBN 90 274 1915 9, Het Spectrum, 1987.

door: Ing. L.J.M.W. Gielen

Afgelopen jaar is de autobiografie van John Sculley, topman van Apple computers, uitgebracht. Het boek draagt de veelzeggende subtitel: 'van Pepsi naar Apple'. Sculley beschrijft in dit boek zijn carrière bij Pepsi Cola, het vertrek naar Apple en zijn eerste jaren bij Apple. De oorspronkelijke titel 'Odyssee: Pepsi to Apple... a Journey of Adventures, Ideas and Future' geeft goed aan wat de lezer te wachten staat.

Niet lang na de komst van Sculley beleefde Apple een bewogen periode. Sculley vertelt over de ingrijpende reorganisatie in 1985 en hoe zijn aanvankelijke vriendschap met Steve Jobs onmogelijk werd door onoverbrugbare verschillen van inzicht over de toekomst van Apple. Als een rode draad door het boek loopt de tegenstelling tussen Jobs en Sculley.
Onverbloemd schrijft hij over de moeilijke periode bij Apple, over de daling van de verkopen, het sluiten van fabrieken en gedwongen ontslagen, maar daarnaast vertelt hij enthousiast over de hoogtepunten en het succes van Apple.

Het succes van de eerste jaren van Apple wordt door Sculley vrijwel volledig toegeschreven aan marketing-man Mike Markkula die in 1977, toen hij op het punt stond zich als multimiljonair uit het zakenleven terug te trekken, werd overgehaald door Steve Jobs om samen met hem en Steve Wozniak Apple computer op te richten.

Het verschil in mentaliteit tussen de bedrijven Pepsi en Apple wordt door Sculley haarfijn beschreven: Pepsi strak georganiseerd en efficiënt, Apple dynamisch en flexibel.

Het boek, waarin autobiografische hoofdstukken worden afgewisseld met beschrijvingen van marketing-aspecten, is een must voor iedereen die geïnteresseerd is in de geschiedenis van Apple. Het boek is in de Verenigde Staten een bestseller en het in één adem uitlezen is niet voorbehouden aan marketing- of computer-deskundigen alleen.

'Operating Systems, design and implementation' door Andrew S. Tanenbaum
Prentice Hall International Editions, 1988.

door: Ing. J.C. van Winkel RI

Prof. Tanenbaum van de Vrije Universiteit heeft na zijn boeken over computerarchitectuur en computernetwerken, nu ook een boek geschreven over Operating Systems (O.S.). Hij maakt in zijn nieuwe boek gebruik van MINIX, een door hem geschreven UNIX[1]-look-a-like. Aan de hand van dit besturingssysteem (zie elders in deze Compact) maakt hij de lezer duidelijk hoe besturingssystemen opgebouwd zijn en welke functies ze vervullen. Om dit goed te kunnen doen, heeft hij van de bodem af, een besturingssysteem geschreven voor IBM-PC's en compatibelen dat, functioneel gezien, zeer sterk lijkt op UNIX. Omdat MINIX door hem zelf is geschreven, kon hij de broncode opnemen in zijn boek (ruim 12500 regels, 270 van de 720 pagina's van het boek). Op deze wijze kan de theorie ook werkelijk in de praktijk toegepast en geverifieerd worden. MINIX is op diskette beschikbaar bij de uitgever, inclusief broncode. Momenteel wordt op de VU de laatste hand gelegd aan een versie van MINIX voor de ATARI ST-computer.

In zijn boek gaat Prof. Tanenbaum, aan de hand van MINIX en UNIX, in op bekende problemen uit de theorie van besturingsystemen:
Hoofdstuk 1 geeft een algemene introductie over besturingssystemen, met een overzicht van de geschiedenis van computers in het algemeen en besturingssystemen in het bijzonder. In hoofdstuk 2

[1] *UNIX is een geregistreerd handelsmerk van AT&T Bell laboratories.*

gaat Prof. Tanenbaum in op interprocescommunicatie, het beheer van parallel draaiende processen (scheduling) en de complicaties die kunnen optreden bij interprocescummunicatie, zoals deadlocks en starvation. Het derde hoofdstuk gaat over input/output. Hierbij wordt weer uitgebreid ingegaan op deadlocks en wat er tegen deadlocks gedaan kan worden.
Ook de diverse manieren waarop device drivers geschreven kunnen worden, worden uitgediept. Een zeer belangrijk onderdeel van een besturingssysteem is het memory management, en dat wordt in hoofdstuk 4 bijzonder uitgebreid behandeld. Met name virtual memory en de strategieën die mogelijk zijn om in het systeem meerdere processen het geheugen te laten delen, worden te uit en te na behandeld. Hoofdstuk 5 gaat in op de wijze waarop bestanden opgeslagen kunnen worden op direct toegankelijke media zoals diskettes en harde schijven. Uiteraard komt hierbij niet alleen de technische implementatie aan bod, maar wordt ook aandacht besteed aan de mogelijkheden om bestanden die in de computer zijn opgeslagen te beveiligen.
Het laatste hoofdstuk bevat een uitgebreide lijst van toegelichte referenties naar artikelen en boeken. Na de zes hoofdstukken volgen nog vijf appendices met daarin achtereenvolgens een introductie in C (de taal waarin MINIX geschreven is), een introductie in de IBM-PC, een gebruikershandleiding voor MINIX, hoe men zelf MINIX naar eigen behoefte kan aanpassen, en ten slotte de broncode van MINIX zelf.

Omdat bij de behandeling van de diverse onderwerpen steeds vermeld wordt hoe de problemen in MINIX zijn aangepakt, waarbij ook nog verwezen wordt naar de MINIX broncode die achterin het boek is opgenomen, is het geen droog theorieboek geworden. Daarbij komt de levendige stijl van schrijven die Prof. Tanenbaum ook al in zijn computerarchitectuur- en computernetwerkboeken ten toon heeft gespreid, zodat het boek een zeer leesbaar karakter heeft. Het is niet verwonderlijk dat het boek op het moment bezig is hét (leer)boek op het gebied van besturingssystemen te worden.

RISC Architecture door Daniel Tabak
161 pagina's, ISBN 0 471 91302 2, John Wiley & Sons

door: Ing. JC van Winkel RI

Professor Tabak geeft in zijn boek RISC Architecture een beeld van de nieuwe architectuur bij het ontwerpen van processorchips.

Het boek geeft een uiteenzetting over de algemene eigenschappen van de RISC-architectuur. RISC staat voor Reduced Instruction Set Computer. De ontwerpers van processors zijn tot de conclusie gekomen dat het in sommige gevallen gunstig kan zijn, om de instructieset van processors te beperken, maar wel zeer rechtlijnig en consequent op te zetten. Hierdoor kunnen de processoren eenvoudiger worden opgezet, waardoor ze sneller te ontwerpen zijn en ook bij de executie van programma's sneller kunnen zijn.

Professor Tabak geeft in zijn boek een inleiding in RISC-architectuur. Het is zeer verduidelijkend als men wel al in processor-architectuur bekend is, maar de RISC-architectuur nog niet kent. Hoewel de inleiding kort is, is deze duidelijk genoeg om de uiteenzetting van de eigenschappen van een lijst van processors te kunnen begrijpen.

Het voornaamste deel van het boek is gewijd aan de behandeling van een groot aantal moderne RISC-processoren: de Berkeley RISC I en II, de Stanford MIPS, de IBM 801, de Acorn Risc Machine (ARM), de Inmos Transputer en andere. Hij zet de eigenschappen van deze processoren op een rijtje en vergelijkt ze. Een aantal processoren is geen echte RISC, maar wordt toch vaak RISC genoemd. Tabak vertelt waarom deze processoren toch vaak tot RISC-processoren gerekend worden.

Samenvattend kan gezegd worden dat het boek een duidelijk beeld geeft van de huidige stand van zaken bij de moderne RISC-ontwerpen. Ook de voor- en nadelen komen duidelijk aan bod.