Leiden nieuwe ontwikkelparadigma's ook tot betere software?

Danny Greefhorst

De mensheid staat niet stil; we leren continue en proberen te bouwen op ervaringen van anderen om steeds verder te komen. In de software engineering gaat het niet anders. Was er in de jaren zestig nog sprake van een ware softwarecrisis, inmiddels hebben we uitgebreide methoden en technieken tot onze beschikking waarmee we grote complexe systemen kunnen ontwikkelen. Daarbij maken we gebruik van moderne paradigma's, zoals componentgebaseerde en servicegeoriënteerde architectuur. De vraag rijst echter in hoeverre de kwaliteit van de softwaresystemen ook echt is verbeterd. Aangezien ook voor IT-auditors de kwaliteit van software een belangrijk aandachtspunt vormt, is het voor hen zinvol om inzicht te hebben in deze materie. Dit artikel, geschreven door een IT-architect, biedt aanknopingspunten voor de advisering over, dan wel beoordeling van paradigma's in ontwikkeltrajecten.

Inleiding

In de afgelopen veertig jaar is er veel gebeurd in de software engineering, zowel qua methoden en technieken als op het gebied van de gehanteerde paradigma's. Een methode geeft een werkwijze om te komen van eisen tot een werkend systeem. Daarbij wordt gebruikgemaakt van technieken die een model bieden om bepaalde aspecten van het systeem te beschrijven. De basis van een methode is het gehanteerde ontwikkelparadigma, de wijze waarop naar softwaresystemen wordt gekeken. Een systeem met een bepaalde functionaliteit kan worden verdeeld in kleinere samenwerkende eenheden. Paradigma's verschillen in de manier waarop dit gebeurt, en het gehanteerde paradigma bepaalt dan ook in sterke mate de architectuur van het softwaresysteem.

Drs. D. Greefhorst is werkzaam als IT-Architect bij IBM Business Consulting Services. Naast zijn primaire rollen als applicatie architect en enterprise architect voert hij ook regelmatig advies- en reviewtrajecten uit op het gebied van softwareontwikkeling en IT-architectuur.

Het meest eenvoudige paradigma is dat van het monolithische systeem. Een dergelijk systeem heeft in feite geen heldere structuur waarin eenduidige elementen te onderkennen zijn. Het nadeel van een dergelijk paradigma is helder; het realiseren en onderhouden van de resulterende systemen is een ware nachtmerrie. In de loop der jaren is er daarom een aantal verschillende paradigma's bedacht die hierin verlichting brengen. De verschillen tussen deze paradigma's liggen in de soort eenheden die worden onderkend en de manier waarop zij samenwerken. De belangrijkste eenheden die we in al die jaren hebben zien passeren, zijn procedures, objecten, aspecten, componenten en services.

Naast de evolutie van methoden en technieken is er ook een aantal andere belangrijke ontwikkelingen geweest. Zo zijn computers een steeds belangrijker deel van onze samenleving gaan uitmaken. Niet alleen is er dus een explosie aan toepassingen waarvoor software een oplossing moet bieden, ook de verwachtingen van gebruikers van deze toepassingen nemen toe, waardoor softwaresystemen steeds groter en complexer worden. Naast het ontwikkelen van systemen wordt ook het integreren ervan steeds belangrijker. Een laatste belangrijke ontwikkeling is die op het terrein van de technologie, zowel op het

gebied van hardware als op dat van software. Softwaresystemen en de daarbij benodigde hulpmiddelen hebben zich steeds moeten aanpassen aan nieuwe technologieën. Dit alles heeft er dan ook voor gezorgd dat software engineering steeds voor nieuwe uitdagingen staat.

Een vraag die bij dit alles rijst, is in hoeverre methoden en technieken voldoende zijn opgewassen tegen de toegenomen complexiteit van softwaresystemen. Zijn moderne softwaresystemen kwalitatief verbeterd ten opzichte van de eerste monolithische softwaresystemen? Dit artikel probeert een antwoord te geven op deze vraag door de belangrijkste ontwikkelparadigma's van de afgelopen jaren met elkaar te vergelijken. De paradigma's worden daarbij afgezet tegen een kwaliteitsraamwerk, dat kwaliteitseigenschappen van software beschrijft.

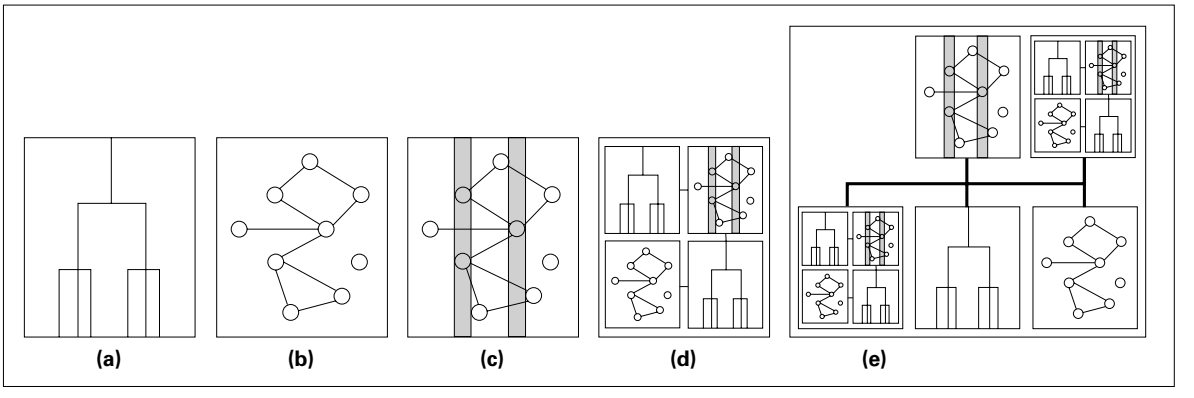
In de rest van dit artikel wordt eerst een overzicht gegeven van de paradigma's, waarna het kwaliteitsraamwerk wordt toegelicht. Daarna worden de hoofdeigenschappen van het raamwerk geprojecteerd op de paradigma's. Het artikel sluit af met een conclusie.

Ontwikkelparadigma's

Deze paragraaf geeft een overzicht van de belangrijkste ontwikkelparadigma's van de afgelopen jaren. Achtereenvolgens zijn dat de procedurele, objectgeoriënteerde, aspectgeoriënteerde, componentgebaseerde en servicegeoriënteerde paradigma's (zie ook figuur 1). Per paradigma worden de concepten, werkwijze, technieken, methoden en programmeertalen beschreven.

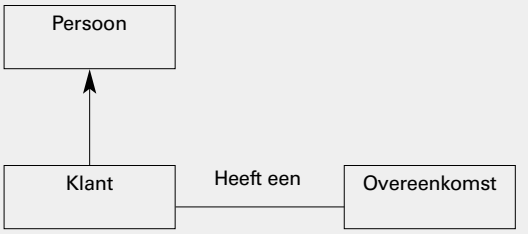
Het 'traditionele' paradigma duiden we hier aan als het *procedurele paradigma*, alhoewel we eigenlijk een aantal verschillende benaderingen en methoden zoals Yourdon, Information Engineering, en SDM op één hoop gooien. Zij hebben echter gemeen dat ze een duidelijk onderscheid maken tussen functies en gegevens. Functies worden geïdentificeerd op het hoogste niveau en worden vervolgens in een aantal slagen gedecomponeerd tot procedures die in procedurele programmeertalen worden geïmplementeerd. Figuur 1a visualiseert deze voor het procedurele paradigma kenmerkende top-down aanpak van de functionele decompositie. Hierbij wordt gebruikgemaakt van technieken als decompositiediagrammen en data-flow diagrammen. Aan de andere kant speelt het modelleren van gegevens een belangrijke rol. Daarbij wordt met name gebruikgemaakt van Entity-Relationship diagrammen. Voorbeelden van procedurele programmeertalen zijn ALGOL, COBOL en C.

Het *objectgeoriënteerde paradigma* is het vervolg op het procedurele paradigma. In objectoriëntatie worden functies en gegevens geclusterd in eenheden (objecten) die zoveel mogelijk overeenkomen met objecten in de werkelijke wereld. Overeenkomstige objecten worden beschreven in een 'klasse', die de gemeenschappelijke methoden (functies) en attributen (gegevens) van alle objecten in die klasse beschrijft. Klassen kunnen eigenschappen van elkaar 'erven', waardoor gemeenschappelijke eigenschappen maar op één plaats hoeven te worden gedefinieerd (zie de box 'objectoriëntatie'). De gedachte is dat klassen gedurende het gehele ontwikkeltraject steeds verder worden verfijnd totdat ze letterlijk in een objectgeoriënteerde programmeertaal kunnen worden opgeschreven. De belangrijkste techniek is het klassendiagram, waarin klassen en hun relaties zijn weergegeven. Figuur 1b visualiseert het netwerk van (min of meer gelijkwaardige) klassen, dat typisch is voor het objectgeoriënteerde paradigma. Dynamische aspecten worden weergegeven in interactiediagrammen. Deze technieken maken



Figuur 1. Systemen conform de procedurele (a), objectgeoriënteerde (b), aspectgeoriënteerde (c), componentgebaseerde (d) en servicegeoriënteerde (e) paradigma's

deel uit van UML (Unified Modeling Language), een taal die inmiddels is uitgegroeid tot de standaardtaal voor het modelleren van objectgeoriënteerde systemen. Ook is er inmiddels een groot aantal objectgeoriënteerde ontwikkelmethoden, zoals Rational Unified Process en SELECT Perspective. Bekende objectgeoriënteerde programmeertalen zijn Smalltalk, C++, Java en C#. De eerste objectgeoriënteerde programmeertaal verscheen overigens reeds in 1967 (SIMULA-67).



Objectoriëntatie

Bij een objectgeoriënteerde aanpak wordt software gemodelleerd als objecten in de werkelijke wereld. Zo geeft het bovenstaande model aan dat er in de werkelijke wereld personen, klanten en overeenkomsten bestaan. Gelijksoortige objecten worden beschreven in een klasse. Alle klantobjecten maken deel uit van de klasse klant. Ook zijn er relaties tussen de verschillende klassen. Een klant heeft een overeenkomst; dat wordt ook wel een associatie genoemd. Een klant is verder een specifiek persoon; dat is een overervingsrelatie. In deze relatie erft de klasse klant (ook wel subklasse genoemd) alle eigenschappen die gelden voor de klasse persoon (ook wel superklasse genoemd). Samen vormen de klasse persoon en klant een zogenaamde 'klassenhiërarchie'.

Het *aspectgeoriënteerde paradigma* [KICK97] is een uitbreiding op het objectgeoriënteerde paradigma. Bij aspectoriëntatie wordt onderkend dat bepaalde aspecten in veel verschillende klassen voorkomen en dat het om die reden nuttig is deze aspecten separaat te beschrijven. Denk daarbij aan systeemeigenschappen zoals logging en beveiliging, maar ook aan functionele aspecten. Zo zou je een nieuwe functie van een systeem als aspect kunnen definiëren, zonder de bestaande programmacode aan te passen. Figuur 1c visualiseert aspecten als verticale banden die meerdere klassen raken. Er bestaat niet echt een standaardwerkwijze om te komen tot aspecten, alhoewel er al veel standaardaspecten gedefinieerd zijn. Standaardtechnieken voor het modelleren van aspecten zijn er evenmin, alhoewel er wel wordt gekeken naar de mogelijkheden om UML hiervoor te gebruiken. Ook bestaan er op dit moment geen echte methoden voor aspectoriëntatie. Aspecten worden beschreven in een eigen taal die vervolgens tijdens ontwikkeling of uitvoering worden toegevoegd aan klassen. Een populaire aspectgeoriënteerde taal is AspectJ, die gebaseerd is op Java.

Een evolutie van objectoriëntatie is het *componentgebaseerde paradigma*, ook wel aangeduid als Component Based Development (CBD). Dit paradigma definieert eenheden (componenten) op een hoger niveau, waarbij de precieze inhoud van een component wordt afgeschermd door zijn interface. Een individuele component kan zelf weer procedureel, objectgeoriënteerd of aspectgeoriënteerd worden gedefinieerd. Dit is te zien in figuur 1d waarin de visualiseringen van al deze paradigma's terugkomen. Hiermee vervangt CBD dus niet zozeer voorgaande paradigma's, maar voegt het een extra abstractieniveau toe. Dit maakt het mogelijk om een systeem in hanteerbare eenheden te definiëren, die vervolgens los van elkaar kunnen worden ontwikkeld. Precieze criteria om te komen tot componenten verschillen per methode, maar gebruikelijk is functies en gegevens te clusteren in respectievelijk proces- en gegevenscomponenten. Er kan voor dit paradigma gebruik worden gemaakt van de componentdiagrammen zoals die beschikbaar zijn in UML. Voorbeelden van componentgebaseerde methoden zijn Catalysis en SELECT Perspective. Deze methoden bouwen voort op het objectgeoriënteerde paradigma. Alhoewel componenten in vrijwel alle programmeertalen kunnen worden ontwikkeld, vragen zij wel extra technologie (middleware) om met elkaar te kunnen communiceren. Componentgebaseerde systemen zijn vaak dan ook gedistribueerd over meerdere fysieke machines, wat mogelijk wordt gemaakt door deze zelfde middleware. Voorbeelden van componenttechnologie zijn COM, CORBA, .NET en J2EE.

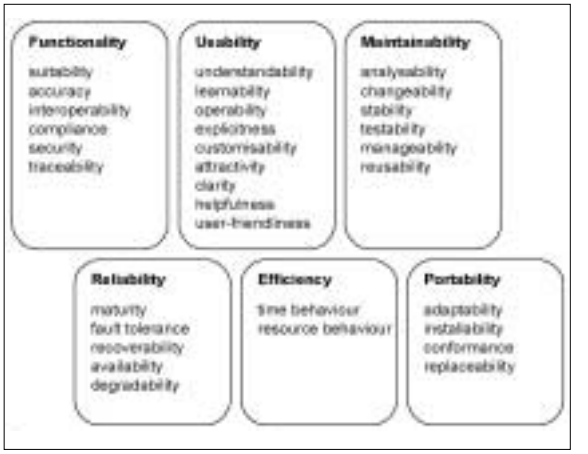
Het laatste paradigma is het *servicegeoriënteerde paradigma*, dat wordt gebruikt voor zowel het integreren van bestaande systemen als het ontwikkelen van nieuwe systemen. Het idee is dat er behoefte is aan een nog hoger abstractieniveau waarop softwaresystemen, of eigenlijk een verzameling van softwaresystemen, worden gedecomponeerd. Daarbij is de precieze inhoud van de eenheden minder relevant, zolang zij maar kunnen samenwerken op basis van services. Een service is een herbruikbaar stuk functionaliteit dat wordt aangeboden door een component of door een softwaresysteem als geheel. Een service wordt afgeleid van een bedrijfsproces, maar is zodanig generiek

dat het ook in andere processen bruikbaar is. Een voorbeeld is de bepaling van de kredietwaardigheid van een klant. UML kan worden gebruikt als techniek voor het modelleren van services, maar ook in XML kunnen services worden beschreven. Er wordt op dit moment op verschillende plaatsen gewerkt aan methoden voor het ontwikkelen van een servicegeoriënteerde architectuur; een standaard op dit gebied is er alleen nog niet. Bestaande objectgeoriënteerde en componentgebaseerde methoden zijn echter relatief eenvoudig uit te breiden met een aantal goede vuistregels om te komen tot goede services. Serviceoriëntatie bouwt net als CBD voort op voorgaande paradigma's en is dus programmeertaal onafhankelijk. Wel geldt ook dezelfde kanttekening dat er specifieke middleware nodig is voor het integreren van services. Dit is vaak Message Oriented Middleware, die berichten gegarandeerd kan afleveren en een asynchroon karakter heeft, waarbij er niet op een antwoord op een bericht wordt gewacht. Ook Web Services standaarden zoals SOAP en WSDL spelen een belangrijke rol. Zoals weergegeven in figuur 1e ontstaat met serviceoriëntatie een landschap van (vaak heterogene) systemen en componenten die met standaard middleware (met dikke lijnen weergegeven) en via services aan elkaar gekoppeld zijn. Servicegeoriënteerde systemen kunnen sterk gedistribueerd zijn, waarbij organisatiegrenzen kunnen worden overschreden.

In het voorafgaande zijn de ontwikkelparadigma's op een neutrale manier beschreven, zonder daarbij aspecten als ervaring, organisatorische inrichting en technische omgeving mee te nemen. In de praktijk zijn het echter juist dergelijke factoren die de kwaliteit van de oplossing sterk beïnvloeden. Denk bijvoorbeeld aan de hoeveelheid ervaring die binnen een organisatie bestaat met een bepaalde ontwikkelstraat, waarvan naast het ontwikkelparadigma onder meer ook hulpmiddelen, fysieke machines, procedures en richtlijnen deel uitmaken. Softwareproducten die worden gemaakt op een nieuwe ontwikkelstraat zullen gemiddeld genomen een lagere kwaliteit hebben.

Kwaliteitsraamwerk

Wat bepaalt nu de kwaliteit van een softwaresysteem? Het is duidelijk dat hier geen eenvoudig antwoord op bestaat. Een kwaliteitsraamwerk biedt ondersteuning bij het beantwoorden van de vraag. Een voorbeeld van een kwaliteitsraamwerk is het Extended ISO 9126 model [ZEIS96] (zie figuur 2), dat een uitbreiding is van de ISO/IEC 9126 standaard. Het model beschrijft 32 kwaliteitseigenschappen waarop softwareproducten kunnen worden beoordeeld, inclusief de bijbehorende



Figuur 2. Extended ISO 9126

indicatoren en meetvoorschriften. De kwaliteitseigenschappen zijn onderverdeeld in zes hoofdeigenschappen: functionality, reliability, usability, efficiency, portability en maintainability. Aangezien dit model een bruikbare manier biedt om ontwikkelparadigma's te vergelijken, is het goed iets dieper op de verschillende eigenschappen in te gaan. Vooral die kwaliteitseigenschappen waarop het gehanteerde ontwikkelparadigma een belangrijke invloed heeft, worden nader belicht.

Functionality heeft betrekking op de specifieke functies die een softwareproduct biedt en de specifieke eigenschappen van deze functies. Onder deze noemer vallen de kwaliteitseigenschappen suitability, accuracy, interoperability, compliance, security en traceability. *Suitability* is de mate waarin het softwareproduct doet wat de gebruiker ervan verwacht. *Accuracy* is de mate van juistheid van de resultaten bij de uitvoering van software en kan worden bepaald aan de hand van onder meer het aantal foute resultaten en de impact daarvan. *Interoperability* heeft betrekking op de mate waarin de software kan interacteren met andere specifieke systemen. Daarbij spelen de benodigde inspanning en de mate waarin de interfaces op elkaar aansluiten een belangrijke rol. *Compliance* geeft aan in hoeverre de software voldoet aan standaarden op diverse gebieden.

Usability gaat over de mate waarin het softwareproduct bruikbaar is. In het bijzonder betreft het de eigenschappen understandability, learnability, operability, explicitness, customisability, attractivity, clarity, helpfulness en user-friendliness. Aangezien het ontwikkelparadigma geen invloed zou mogen hebben op de gebruikerservaring, gaat dit artikel er niet verder op in.

Maintainability geeft aan hoe makkelijk het is om specifieke wijzigingen aan te brengen in de software. Onder maintainability vallen de eigenschappen analysability, changeability, stability, testability, manageability en reusability. *Analysability* gaat over de mate waarin een fout kan worden gelokaliseerd in de software. Het gemak van het daadwerkelijk aanbrengen van een wijziging wordt uitgedrukt met de eigenschap *changeability*. *Stability* betreft het risico dat een wijziging ongewenste bijeffecten heeft. *Reusability* ten slotte, geeft aan hoe makkelijk een stuk software te hergebruiken is in een ander softwareproduct.

Reliability is de mate waarin de software voor een bepaalde tijd op een bepaald niveau kan opereren. Daaronder vallen de eigenschappen maturity, fault tolerance, recoverability, availability en degradability. *Maturity* geeft een indicatie van het aantal fouten dat optreedt bij het uitvoeren van de software. Daarbij spelen begrippen zoals mean time between failures (MTBF) en mean time to failure (MTTF) een rol. *Recoverability* geeft aan hoe snel de software weer kan herstellen van een foutsituatie. *Availability* geeft een indicatie van de tijd dat een softwareproduct beschikbaar is voor gebruik.

Efficiency beschrijft de relatie tussen de performance van de software en de bronnen die daarbij nodig zijn. Enerzijds gaat efficiency dus over tijdgedrag (*time behavior*), waarbij indicatoren als responsetijd en doorvoer relevant zijn. Anderzijds gaat efficiency over de hoeveelheid bronnen die nodig zijn (*resource behavior*). Denk daarbij aan geheugengebruik, netwerkbandbreedte, processorgebruik en schaalbaarheid.

De laatste hoofdeigenschap is *portability*, de mate waarin software kan worden overgebracht naar een andere omgeving. Hieronder vallen de eigenschappen adaptability, installability, conformance en replaceability. De meest voor de hand liggende kwaliteitseigenschap daarbinnen is *adaptability* die aangeeft hoe eenvoudig een softwareproduct in een andere omgeving kan worden gezet. *Replaceability* is de mate waarin een stuk software een ander stuk software kan vervangen.

Vergelijking van de paradigma's

Op basis van het geschetste kwaliteitsraamwerk wordt het mogelijk om meer inzicht te krijgen in de kwaliteit van de ontwikkelparadigma's. Tabel 1 zet de verschillende ontwikkelparadigma's af tegen de hoofdeigenschappen van het kwaliteitsraamwerk. In de rest van deze sectie zullen we per paradigma een dieper gaande analyse geven.

	Procedureel	Object-georiënteerd	Aspect-georiënteerd	Component-gebaseerd	Service-georiënteerd
Functionality	**-**	**±**	**+**	**+**	**++**
Suitability	-	+	+	+	+
Accuracy	-	+	++	+	
Interoperability	-	-	-	+	++
Compliance	-	-	-	+	++
Reliability	**+**	**+**	**+**	**-**	**±**
Maturity	+	+	+	-	±
Availability	+	+	+	-	±
Recoverability	+	+	+	-	-
Efficiency	**+**	**-**	**-**	**±**	**+**
Time Behavior	++	-		±	+
Resource Behavior	+	-	—	±	++
Maintainability	**-**	**+**	**++**	**++**	**++**
Analysability	-	+	±	++	+
Changeability	±	+	++	+	
Stability	±	+	++	+	
Reusability	-	±	±	+	++
Portability	**-**	**±**	**±**	**+**	**++**
Adaptability	-	-	-	+	++
Replaceability	-	±	+	+	+

Tabel 1. Vergelijking ontwikkelparadigma's

Procedureel

Het eerste ontwikkelparadigma dat we analyseren is het procedurele paradigma. Dit is een nogal gesloten paradigma; systemen zijn vaak monolithisch van aard, geschreven in één programmeertaal en niet erg gericht op het voldoen aan open standaarden en het integreren met andere systemen. Ook het (her)gebruiken van software in een andere context is vaak beperkt mogelijk, wat enerzijds wordt veroorzaakt door de taalgebondenheid, en anderzijds door de structuur van de code. Deze structuur wordt gekenmerkt door veel dwarsverbanden, zowel door het globaal gebruik van procedures als van gegevens, waardoor er niet eenvoudig een deel uit te vervangen is. Ook het onderhouden van procedurele code is vaak niet erg eenvoudig. De structuur van de code is zodanig dat het niet eenvoudig is om de te wijzigen code te lokaliseren. Bij het aanbrengen van wijzigingen is het bovendien vaak lastig om te zien wat de invloed van de wijziging is op andere delen van de software. Ook wordt de kans op fouten in de code groter. Een andere observatie is dat een procedurele specificatie niet altijd even eenvoudig is af te stemmen met een gebruiker, waardoor het eindresultaat niet altijd even goed overeenstemt met de verwachtingen van de gebruiker. Dit alles

zorgt ervoor dat het procedurele paradigma niet erg goed scoort op de hoofdeigenschappen functionality, portability en maintainability.

Het procedurele paradigma heeft echter ook een aantal positieve kanten. Deze zijn met name het resultaat van juist weer het monolithische karakter van de software, waardoor het vaak op één fysieke machine draait. Voordeel hiervan is een hogere reliability van het systeem: het is niet afhankelijk van andere machines, die kunnen falen, en kan in geval van eigen falen weer snel worden opgestart. Natuurlijk zorgt ook de grotere hoeveelheid kennis en ervaring met dit type systemen ervoor dat de organisatie een hogere betrouwbaarheid kan garanderen. Voorts heeft het monolithische karakter belangrijke voordelen voor de efficiency van de software. Zo is er geen inefficiënte netwerkcommunicatie nodig en er is ook geen behoefte aan ingewikkelde middleware om de problematiek op te lossen die ontstaat bij het distribueren van een systeem. Bovendien is procedurele code relatief eenvoudig te optimaliseren voor tijdgedrag, waarbij alleen code die strikt nodig is wordt aangeroepen. Dit alles zorgt ervoor dat het procedurele paradigma een goede keuze is voor software waarbij betrouwbaarheid en tijdgedrag van groot belang zijn. Denk daarbij bijvoorbeeld aan embedded omgevingen, waaraan hoge beschikbaarheidseisen worden gesteld en die een zeer voorspelbaar gedrag met minimale vertragingen moeten hebben.

Objectgeoriënteerd

Het objectgeoriënteerde paradigma beoogt een betere structurering van de software te realiseren ten opzichte van het procedurele paradigma. Hierdoor wordt met name de maintainability van de software vergroot. Het idee is dat het onderverdelen van de software in een structuur die zoveel mogelijk overeenkomt met objecten in de werkelijke wereld erg overzichtelijk is. Daardoor is het relatief eenvoudig om bij wijzigingen de plaats in de software die moet worden aangepast, te vinden. Logica is eenduidig gelokaliseerd in één klasse, waardoor een wijziging maar op één plaats hoeft te worden aangebracht.
Kanttekening daarbij is wel dat kennis van de klassenhiërarchie essentieel is voor het begrijpen van de code en het op de juiste wijze kunnen aanbrengen van wijzigingen; een verkeerde wijziging in een superklasse kan leiden tot problemen in alle subklassen. Klassenhiërarchieën zouden dan ook niet te diep moeten zijn. Een ander voordeel dat vaak met objectoriëntatie wordt geassocieerd is hergebruik, wat inhoudt dat een klasse relatief eenvoudig in een andere context zou kunnen worden ingezet. In de praktijk blijkt hergebruik lastiger te zijn doordat klassen toch vaak sterk gekoppeld zijn aan andere klassen, en niet in de laatste plaats aan de gebruikte taal, tool en omgeving.
Als je kijkt naar reliability dan is objectoriëntatie vergelijkbaar met het procedurele paradigma. Dit hangt samen met de constatering dat puur objectgeoriënteerde systemen eigenlijk niet gedistribueerd zijn en dus ook niet afhankelijk zijn van andere systemen. Een voordeel van objectoriëntatie is verder dat objecten een representatie zijn van de wereld van de gebruiker waardoor de kwaliteit van de specificatie toeneemt. Uiteindelijk is hierdoor ook de kans groter dat de software doet wat de gebruiker verwacht. Een laatste voordeel van objectoriëntatie is de betere accuracy, enerzijds door het beter gelokaliseerd zijn van functionaliteit. Ook is een objectgeoriënteerde UML specificatie beter één-op-één te vertalen naar code, waardoor er minder fouten zullen optreden.

Casus 1 (objectgeoriënteerd)

Een organisatie in de uitzendbranche besluit zijn frontoffice te vernieuwen. Er moet een nieuwe applicatie worden ontwikkeld die gebruikmaakt van de nieuwste Java technologie en die, aangezien Java een objectgeoriënteerde taal is, dan ook volledig gebaseerd is op objectoriëntatie. Een groot deel van het ontwikkeltraject blijkt te bestaan uit het ontwikkelen van een raamwerk dat de basis vormt van de applicatie. Dit raamwerk biedt een generieke objectrelationele vertaling, een implementatie van een organisatiespecifieke look-and-feel en veel generieke superklassen waar ontwikkelaars van moeten erven om de specifieke functionaliteiten te ontwikkelen. Het raamwerk blijkt continue te moeten worden aangepast tijdens de ontwikkeling van het systeem. Dit levert voor ontwikkelaars veel extra werk op doordat ze code die zij reeds hadden ontwikkeld, moeten aanpassen aan wijzigingen in de generieke superklassen. Ook valt op dat het ontwikkelen behoorlijk arbeidsintensief is doordat er relatief veel klassen nodig zijn voor de gehele applicatie; in totaal gaat het om vele duizenden klassen. Al met al roept dit vragen op over het ontwikkelen in een objectgeoriënteerde taal. Het lijkt erop dat dergelijke talen veel te veel vrijheidsgraden kennen, waardoor er een raamwerk nodig is dat deze vrijheid weer wegneemt. Dit zorgt ervoor dat objectoriëntatie een weinig efficiënte manier van ontwikkelen is.

Er kleeft ook een aantal nadelen aan het objectgeoriënteerde paradigma. We hebben al wat van de maintainability-eigenschappen genuanceerd. Een verdere nuancering is dat systemen in de praktijk bijna nooit volledig objectgeoriënteerd kunnen worden ontwikkeld; er dient vaak geïntegreerd te worden met een relationele

database waardoor een extra vertaalslag nodig is. Daarnaast maken de pure werkelijke wereld objecten vaak slechts een beperkt deel uit van een objectgeoriënteerd systeem; er zijn ook veel technische klassen nodig die allerlei standaardtaken uitvoeren. Denk daarbij bijvoorbeeld aan standaardklassen voor user-interface logica en persistentie (het mechanisme dat ervoor zorgt dat de toestand van objecten bewaard wordt na afloop van het programma). Een gebrek in objectoriëntatie is de aanname van een gesloten wereld, die inhoudt dat alle behoeften van een systeem kunnen worden vervuld door de objecten binnen het systeem. In de praktijk bevindt zich een groot deel van deze behoeften buiten het systeem, waardoor er extra vertalingen nodig zijn. Standaarden voor objectoriëntatie richten zich met name op de programmeertalen. Goede standaarden voor interoperabiliteit van puur objectgeoriënteerde systemen bestaan in feite niet. Dit alles zorgt ervoor dat objectoriëntatie niet goed scoort qua functionality en portability. Ook is objectoriëntatie niet specifiek gericht op efficiency omdat de vereiste objectallocatie en communicatie een bepaalde overhead geven. Daarnaast zijn objectgeoriënteerde omgevingen niet gebouwd met het oog op schaalbaarheid van de te ontwikkelen systemen. Om nog maar te zwijgen over de overhead waar objectgeoriënteerde talen als Java en Smalltalk mee te maken hebben, omdat bij deze talen de omzetting van broncode naar objectcode tijdens de uitvoering van het programma plaatsvindt – het zijn 'geïnterpreteerde' talen – en niet via compilatie vooraf, en omdat ze geheugen automatisch aanvragen en weggooien. Dit maakt dat een systeem met harde real-time eisen beter niet volledig objectgeoriënteerd kan worden ontwikkeld.

Aspectgeoriënteerd

Strikt genomen is aspectoriëntatie een generieke benadering, maar in de praktijk is het, zoals gezegd, sterk gekoppeld aan objectoriëntatie en de daarvoor beschikbare talen. Dat betekent dat aspectoriëntatie een groot deel van de voor- en nadelen van objectoriëntatie met zich meebrengt. Het belangrijkste specifieke voordeel van aspectoriëntatie is dat aspecten die op meerdere plaatsen terugkomen, op één plaats kunnen worden gedefinieerd. Denk bijvoorbeeld aan het eenmalig definiëren van een audit trail en de condities waaronder deze gebruikt dient te worden. Hierdoor zijn wijzigingen snel aan te brengen en zijn klassen beter herbruikbaar buiten hun oorspronkelijke context. Deze eigenschappen hebben dus een positieve invloed op de maintainability en portability van het systeem. Ook zorgt aspectoriëntatie ervoor dat er minder fouten in de software sluipen doordat handmatige duplicatie van code wordt voorkomen.

Genoemde voordelen moeten wel genuanceerd worden. Met aspectoriëntatie wordt een nieuwe technologie geïntroduceerd die het systeem en het onderhoud daaraan direct weer complexer maakt. Verder brengt het definiëren van een groot aantal aspecten nieuwe uitdagingen met zich mee. Ze moeten namelijk los van elkaar beheerd worden en blijken elkaar soms op onvoorspelbare manieren te beïnvloeden. Naast deze nuanceringen geldt ook dat aspectoriëntatie gebonden is aan de taal, tool en omgeving waardoor een aspect niet zomaar in een andere context inzetbaar is. Het ontbreken van standaarden voor aspectoriëntatie helpt daarbij ook niet. Een ander nadeel dat zich bij sommige aspectimplementaties voordoet, is een negatieve invloed op de efficiency van de software, doordat de definities van de aspecten pas tijdens runtime worden geïnterpreteerd en hun plaats krijgen in de objectcode, wat tijd kost. Al met al is aspectoriëntatie nog een paradigma in ontwikkeling.

Componentgebaseerd

Het componentgebaseerde paradigma biedt een aantal belangrijke voordelen ten opzichte van de hiervoor beschreven paradigma's. Het biedt primair een mechanisme om de complexiteit van een systeem te reduceren door het in beter onderhoudbare componenten onder te verdelen. Aanpassingen zijn daarbij gelokaliseerd in componenten, die onafhankelijk zijn van de implementatie van andere componenten ('ontkoppeling'). Afhankelijkheden tussen componenten zijn namelijk strikt gebaseerd op de beschikbare interfaces. Het vervangen en hergebruiken van componenten wordt daardoor een realistische optie. Daarbij geldt overigens wel een aantal kanttekeningen. In de eerste plaats dient de andere component via dezelfde componenttechnologie te zijn ontwikkeld. Gelukkig bestaan hier oplossingen voor, waardoor interoperability in de praktijk realistisch is, zeker binnen de grenzen van een organisatie. Een andere belangrijke voorwaarde voor hergebruik van componenten is een goede organisatorische inrichting. Zo dienen componenten specifiek voor hergebruik te worden ontwikkeld en vervolgens te worden beheerd door een projectonafhankelijke organisatie-eenheid.
De efficiency van componentgebaseerde systemen is acceptabel. Door het typisch gedistribueerde karakter van componentgebaseerde systemen gaat er wel wat tijd zitten in netwerkcommunicatie, maar componenttechnologie heeft vaak ingebouwde ondersteuning voor schaalbaarheid. Denk aan mechanismen voor dupliceren van componenten, waarbij meerdere identieke 'instanties' van een component worden gecreëerd, zodat verzoeken kunnen worden gespreid over alle beschikbare instanties. Net als bij objecten kunnen gebruikers zich ook goed iets voor-

stellen bij componenten, wat een positief effect heeft op de kwaliteit van de specificatie en uiteindelijk dus de suitability. Verder zijn componentgebaseerde systemen qua accuracy vergelijkbaar met objectgeoriënteerde systemen; functionaliteit is ook gelokaliseerd en componenten kunnen vrijwel geheel uit UML worden gegenereerd.

Casus 2 (componentgebaseerd)

Bij een bank wordt een groot deel van de kredietprocessen, inclusief de bijbehorende IT-voorzieningen, herzien. Dit betekent dat een groot aantal systemen opnieuw wordt ontwikkeld, zoals het kredietaanvraagsysteem, het zekerhedensysteem en het relatiesysteem. Men besluit deze systemen volledig componentgebaseerd te ontwikkelen, zodat de ontwikkelde componenten kunnen worden hergebruikt op andere plaatsen in de organisatie. Concreet is er belangstelling vanuit de Belgische afdeling voor het hergebruiken van het zekerhedensysteem. Na de eerste opleveringen van de componenten blijkt de componentgebaseerde aanpak minder succesvol dan verwacht.
Er zijn met name problemen met de beschikbaarheid van de gehele keten die eerder rond de tachtig procent dan rond de honderd procent lijkt te liggen. Een analyse van de situatie lijkt met name te wijzen op problemen in de beheerorganisatie, welke onvoldoende is uitgerust voor dergelijke nieuwe architecturen.

De nadelen van het componentgebaseerde paradigma liggen in het typisch gedistribueerde karakter van componentgebaseerde systemen. Naast genoemde netwerkcommunicatie zorgt dit er ook voor dat de reliability van het systeem minder groot wordt, doordat bepaalde componenten kunnen uitvallen. Dit wordt nog eens versterkt door het feit dat veel componenten gebruikmaken van synchrone communicatie, waarbij gewacht wordt op het antwoord op een verzoek. Ook het opnieuw opstarten van een gedistribueerd systeem vergt extra tijd, welke deels voor rekening komt van de gebruikte componenttechnologie. Deze laatste helpt overigens wel weer met het verhogen van de beschikbaarheid van het systeem door het redundant uitvoeren van componenten.

Servicegeoriënteerd

Serviceoriëntatie is primair gericht op het op grotere schaal kunnen hergebruiken van functionaliteit en het voorkomen van redundante koppelingen tussen systemen. Met name de context waarbinnen hergebruik mogelijk is, wordt vergroot; een service is binnen de gehele organisatie en potentieel zelfs daarbuiten herbruikbaar. Deze herbruikbaarheid wordt gefaciliteerd door de beschikbare open standaarden, met name op het gebied van Web Services. Deze open standaarden zijn inmiddels breed geïmplementeerd en stellen organisaties in staat allerlei heterogene systemen aan elkaar te verbinden.
Een andere belangrijke bijdrage aan herbruikbaarheid is de grote mate van ontkoppeling van services. Dat wil zeggen dat een systeem dat een service aanroept niet hoeft te ‘weten’ hoe die service is geïmplementeerd, en waar deze zich bevindt. Deze ontkoppeling is nog groter dan bij het componentgebaseerde paradigma doordat de servicedefinitie helemaal onafhankelijk is van de serviceimplementatie. In feite zegt serviceoriëntatie niets over de wijze waarop een service geïmplementeerd is. Ook zijn services vaak herkenbaar voor de gebruiker, wat een positieve invloed heeft op de suitability. Net als bij het componentgebaseerde paradigma geldt ook voor serviceoriëntatie dat hergebruik van services vraagt om een specifieke organisatorische inrichting. Al met al scoort serviceoriëntatie goed op de gebieden functionality, portability en reusability. Voor wat betreft de overige maintainability-eigenschappen en accuracy is serviceoriëntatie afhankelijk van het onderliggende paradigma. Idealiter is dat componentgebaseerd, maar ook procedurele, objectgeoriënteerde en aspectgeoriënteerde systemen kunnen direct services beschikbaar stellen. Dit is tevens een gevaar, want een top-down definitie van services kan leiden tot een functionele decompositie van het systeem conform het procedurele paradigma. Dit paradigma is nu juist niet het meest geschikt voor grote, complexe systemen.

Voor een deel heeft serviceoriëntatie verder ook last van de uitdagingen op het gebied van reliability die horen bij gedistribueerde systemen. Servicegeoriënteerde systemen scoren daarbij wel iets beter dan puur componentgebaseerde systemen doordat er vaak gebruikgemaakt wordt van asynchrone communicatie (‘fire-and-forget’). Bij deze vorm van communicatie hoeft de aangeroepen service niet simultaan beschikbaar te zijn. Deze is echter slechts toepasbaar voor een deel van de services. Ook de efficiency van een servicegeoriënteerd systeem is potentieel beter dan dat van een componentgebaseerd systeem. Dit, doordat services typisch op een hoger granulariteitsniveau (meer functionaliteit omvattend) worden gedefinieerd, waardoor er minder aanroepen noodzakelijk zijn. Services kunnen – doordat zij typisch geen procestoestand onthouden na een aanroep – ook relatief eenvoudig worden opgeschaald. Al met al is serviceoriëntatie een goed paradigma voor het ontwikkelen van grootschalige systemen, en het integreren van heterogene systemen.

Casus 3 (servicegeoriënteerd)

Een grote financiële instelling heeft vanuit de historie een zeer uitgebreid systemenlandschap, met veel dwarsverbanden en functionele redundantie. Dit zorgt ervoor dat wijzigingen lastig zijn door te voeren en dat de IT ontwikkel- en beheerkosten relatief erg hoog zijn. Zo wijzen onderzoeken uit dat financiële instellingen in de Verenigde Staten relatief veel minder geld aan hun IT-voorzieningen besteden. Dit alles leidt ertoe dat het genoemde bedrijf een nieuwe architectuurvisie definieert, gebaseerd op service-oriëntatie. Centraal in deze visie staat een blauwdruk waarin de verschillende domeinen in de organisatie zijn gedefinieerd, en die worden verbonden door een centrale applicatiebus. Deze applicatiebus is een stuk middleware dat berichten routeert tussen applicaties in de verschillende domeinen zodat zij onafhankelijk van elkaar kunnen evolueren. Een doel van niet te onderschatten belang daarbij is het faciliteren van de consolidatie van systemen in de organisatie waardoor kosten worden bespaard en synergie tussen de bedrijfsonderdelen kan worden bereikt. Een aantal jaar na de introductie van de architectuurvisie is er inmiddels een flink aantal herbruikbare services beschikbaar. Een nieuwe bedrijfsstrategie waarbij autonomie van de bedrijfsonderdelen weer belangrijker wordt, zorgt er echter voor dat eerder geformuleerde consolidatiedoelstellingen niet geheel haalbaar blijken te zijn.

Conclusies

In dit artikel is inzicht gegeven in de evolutie van ontwikkelparadigma's en hun relatie met kwaliteitseigenschappen. Daarbij is gesignaleerd dat nieuwere componentgebaseerde en servicegeoriënteerde ontwikkelparadigma's goed scoren op eigenschappen als onderhoudbaarheid, schaalbaarheid, portabiliteit en interoperabiliteit. Daardoor zijn ze beter geschikt voor het ontwikkelen van grotere en complexere systemen. Serviceoriëntatie is vooral ook erg geschikt in situaties waarbij heterogene systemen geïntegreerd moeten worden. Belangrijke eigenschappen als tijdgedrag en betrouwbaarheid zorgen voor extra uitdagingen door het typisch gedistribueerde karakter van systemen die volgens deze paradigma's worden ontwikkeld. Deze eigenschappen vragen dus om extra aandacht, zowel tijdens ontwikkeling als beheer. De performance van systemen wordt voor een groot deel gewaarborgd door nieuwere en snellere hardware. Betrouwbaarheid zal echter moeten worden ingebouwd en worden bewaakt in de beheerfase.

Leiden deze nieuwe ontwikkelparadigma's dan per definitie tot betere systemen? Neen: het antwoord op de vraag welk paradigma in een concrete situatie het meest geschikt is, hangt nog steeds af van de context. Voor een relatief kleinschalig systeem waarbij performance of betrouwbaarheid een belangrijke rol spelen, is het procedurele ontwikkelparadigma bijvoorbeeld nog steeds het meest geschikt. En systemen waarbij een goede representatie van de werkelijke wereld van belang is, kunnen prima object georiënteerd ontwikkeld worden; denk bijvoorbeeld aan simulatiesoftware. Bedenk ook dat servicegeoriënteerde en componentgebaseerde systemen op het laagste niveau uiteindelijk toch procedureel of objectgeoriënteerd zijn. Ten slotte wordt de kwaliteit van systemen ook voor een groot deel bepaald door organisatiefactoren zoals ervaring, organisatorische inrichting en technische omgeving. Een onvolwassen organisatie zal geen kwalitatief hoogwaardige softwareproducten ontwikkelen.

Literatuur

[KICK97] Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C. Lopez, J.M. Loingtier en J. Irwin, (1997), *Aspect Oriented Programming*, European Conference on Object Oriented Programming (ECOOP), Springer-Verlag, LNCS 1241, Berlijn.

[ZEIS96] Zeist, B. van, P. Hendriks, R. Paulussen en J. Trienekens, (1996), *Kwaliteit van softwareprodukten – praktijkervaringen met een kwaliteitsmodel*, Kluwer Bedrijfswetenschappen, ISBN 90-267-2430-6.