Met de komst van Internet en applicatieservers ontstaat nieuwe aandacht voor gedistribueerde systemen. Het distribueren van systemen heeft veel voordelen, maar is ook complexiteitsverhogend. Hoe kunnen dergelijke systemen voldoen aan eisen die een organisatie stelt? Architectuur beschrijft fundamentele keuzen en is de plaats waar (kwaliteits)eisen worden geborgd. Dit artikel beschrijft dergelijke fundamentele keuzen en de relatie van deze keuzen met middleware.

Inleiding

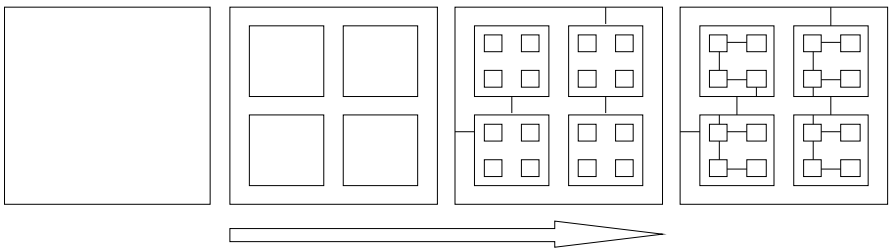
Gedistribueerde systemen zijn systemen die opgesplitst zijn in componenten die zich op fysiek verschillende machines bevinden. Er zijn veel goede redenen om systemen te distribueren. Zo heeft bijvoorbeeld het splitsen van een systeem in client en server componenten een positief effect op de schaalbaarheid, betrouwbaarheid, herbruikbaarheid en onderhoudbaarheid. Gedistribueerde systemen verhogen echter ook de complexiteit; systemen moeten worden opgesplitst en verbonden met middleware waardoor programmacode complexer wordt.

De komst van Internet heeft een nieuwe impuls aan gedistribueerde systemen gegeven. De belangrijkste reden voor het succes van Internet technologie is dat hiermee toepassingen niet meer expliciet bij gebruikers hoeven te worden geïnstalleerd. Naast middleware voor het distribueren van een gebruikersinterface, typisch de rol van een web-server, is er ook behoefte aan robuuste middleware-omgevingen om bedrijfslogicacomponenten te faciliteren. Aan dergelijke middleware worden typisch hogere eisen gesteld met betrekking tot transactieverwerking, beveiliging, schaalbaarheid, en betrouwbaarheid. Het is deze categorie van middleware die ook wel wordt aangeduid als applicatieserver.

Applicatieservers zijn erop gericht zoveel mogelijk complexiteit uit de programmacode te halen door automatisch benodigde eigenschappen aan componenten toe te kennen. Bijbehorende ontwikkelomgevingen benadrukken de complexiteitsreductie door automatisch code te genereren op basis van specificaties in bijvoorbeeld UML. De zekerheid die hierdoor wordt geboden is echter voor een deel schijn. Zonder eerst goed na te denken over architectuur kan een systeem nooit aan alle (kwaliteits)eisen voldoen.

Architectuur

Architectuur beschrijft de fundamentele keuzen die gemaakt moeten worden voordat aan het ontwerp en de bouw van een systeem wordt begonnen. Een architectuur is dan ook de plaats waar kwaliteitseisen moeten worden geborgd. Fundamentele keuzen in een systeem zijn de onderverdeling van het systeem in componenten en het relateren van deze componenten aan elkaar en aan de omgeving (zie figuur 1). Alhoewel de woorden “systeem” en “componenten” misschien al snel aan een softwaresysteem en softwarecomponenten doen denken is hun definitie breder. Zo kan een organisatie ook worden gezien als een systeem en kunnen in deze organisatie verschillende soorten componenten worden onderkend, denk bijvoorbeeld aan bedrijfsprocessen, producten, medewerkers, netwerken en computersystemen.



Figuur 1 Architectuur beschrijft de fundamentele keuzen om te komen tot componenten en relaties

Figuur 2 geeft de verschillende architectuurperspectieven weer waarmee naar een organisatie kan worden gekeken [Wagter2001]. De business architectuur belicht een organisatie en haar bedrijfsprocessen en geproduceerde producten. De informatie-architectuur beschrijft de gegevens en applicaties in een organisatie. De technische architectuur geeft weer hoe de IT infrastructuur de organisatie faciliteert, middels hardware, platformen en middleware. Een software-architectuur beschrijft alle perspectieven waarin software een rol speelt, de applicatie- en middleware perspectieven in het bijzonder. Elk perspectief kan op

verschillende abstractieniveaus worden beschreven, maar een pragmatisch onderscheid is dat tussen algemene principes, beleidslijnen en modellen. Voorbeelden van algemene principes zijn “een multi-channel organisatie worden” of “streven naar straight through processing”. Bij beleidslijnen kan gedacht worden aan richtlijnen zoals “hergebruiken voor kopen voor bouwen”, maar ook aan een IT-middelen beleid waarin standaard keuzes voor hardware, platformen en middleware zijn gemaakt. Modellen beschrijven componenten en hun relaties binnen een perspectief zoals een procesmodel of een gegevensmodel. Zij zouden eigenlijk alleen moeten worden gemaakt als ze projecten ondersteunen.

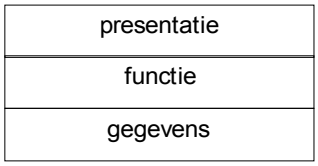
	Business Architectuur			Informatie Architectuur		Technische Architectuur		
	Product	Proces	Organisatie	Gegevens	Applicatie	Middleware	Platform	Netwerk
Algemene principes								
Beleidslijnen								
Modellen								

Figuur 2 Perspectieven in de architectuur van een organisatie

In de rest van dit artikel zullen een aantal belangrijke architectuurprincipes worden beschreven tezamen met hun invloed op software-architectuur.

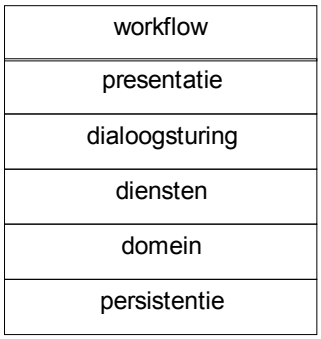
Scheiding van verantwoordelijkheden

Het basisprincipe van architectuur is "scheiding van verantwoordelijkheden" en is voor alle perspectieven en niveau's van architectuur geldig. Scheiden van verantwoordelijkheden betekent dat een systeem moet worden opgesplitst in componenten met duidelijk gedefinieerde verantwoordelijkheden [Wirfs-Brock1990]. Zo zal bijvoorbeeld de verantwoordelijkheid voor individuele bedrijfsprocessen moeten worden belegd bij verschillende afdelingen in een organisatie. De invloed van het organisatieperspectief op de applicatieperspectief zorgt ervoor dat deze bedrijfsprocessen ook in verschillende softwarecomponenten moeten worden gerealiseerd. Een goede scheiding van verantwoordelijkheden zorgt ervoor dat een systeem eenvoudiger te begrijpen, te ontwikkelen en aan te passen is. De reden hiervoor is dat sneller duidelijk is waar bepaalde verantwoordelijkheden zijn (of zouden moeten zijn) gedefinieerd. Aangezien zo’n 70% van het aanbrengen van wijzigingen gerelateerd is aan het opsporen van de plaats waar de wijziging moet worden aangebracht, kan door dit principe de kosten van software-onderhoud aanzienlijk worden verkleind. Een goede scheiding van verantwoordelijkheden is tevens de basis voor het verdelen van werk, waarbij componenten los van elkaar kunnen worden ontworpen en geïmplementeerd. Merk echter op dat dit principe ook te ver worden doorgevoerd waardoor het zijn doel voorbij streeft en juist leidt tot minder onderhoudbare software. Uiteindelijk is een goede scheiding van verantwoordelijkheden een kwestie van ervaring.



Figuur 2 Drie-lagenmodel

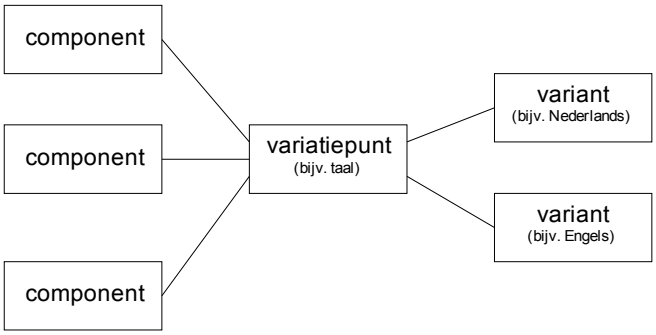
Een belangrijke scheiding van verantwoordelijkheden binnen het applicatieperspectief is de scheiding tussen functionele lagen. Het bekendst is het zogenaamde drie-lagen model (figuur 2) waarin onderscheid wordt gemaakt tussen een presentatielaag, functielaag en gegevenslaag. Deze lagen zijn verantwoordelijk voor respectievelijk interactie met de gebruiker, (bedrijfs)functionaliteit en opslag van bedrijfsgegevens. Figuur 4 geeft een uitgebreider lagenmodel weer, die veel gelijkenis vertoont met de gelaagde bedrijfsobjecten beschreven in [Prins1997]. In dit model wordt een workflowlaag onderkend die de geautomatiseerde aansturing van bedrijfsprocessen verzorgt. De workflow wordt typisch gestuurd op basis van de verschillende toestanden die bepaalde bedrijfsobjecten hebben. Processen waarbij meerdere actoren betrokken zijn worden opgesplitst in atomaire use-cases per actor die zich in de dialoogsturingslaag bevinden. Vanuit de dialoogsturingslaag worden individuele schermen in de presentatielaag aangestuurd. Bedrijfslogica wordt opgesplitst in een dienstenlaag en een domeinlaag. In de domeinlaag bevinden zich bedrijfsobjecten zoals “klant” en “overeenkomst”. De dienstenlaag beheert de relaties tussen bedrijfsobjecten en voert bedrijfsobject-overstijgende diensten uit. De opslag van de bedrijfsobjecten wordt verzorgd in de persistentielaag.



Figuur 4 Uitgebreider lagenmodel

Variatie

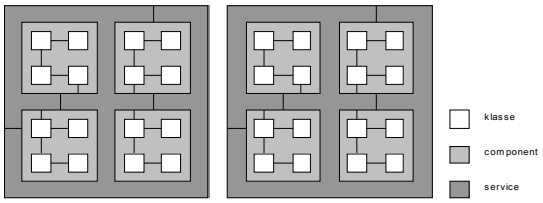
Na een eerste onderverdeling in lagen kunnen componenten binnen de lagen worden onderscheiden. Een belangrijke scheiding van verantwoordelijkheden wordt veroorzaakt door de aanwezigheid van variatiepunten. Variatiepunten zijn eigenschappen die kunnen variëren, bijvoorbeeld de taal die in een applicatie wordt gebruikt. Gekoppeld aan variatiepunten zijn de verschillende varianten die kunnen worden onderscheiden, in het geval van taal bijvoorbeeld Nederlands en Engels. Variatie kan in een bepaalde versie van een applicatie zitten, maar kan ook ontstaan uit wijzigingen die in de toekomst worden voorzien. Algemeen principe is het variatiepunt en de varianten te encapsuleren in afzonderlijke componenten (zie figuur 3) . De component met het variatiepunt verbergt de variatie voor afnemende componenten en is de plaats waar de keuze voor een bepaalde variant wordt gemaakt. Het kiezen of later ontwikkelen van een variant kan zo worden gelokaliseerd, wat de onderhoudbaarheid verhoogt. Het denken in variatiepunten en varianten vraagt ook om het denken over het moment in tijd waarop variatiepunten en varianten worden gedefinieerd en aan elkaar worden gekoppeld. Deze tijdstippen hoeven niet aan elkaar gelijk te zijn en kunnen variëren van het moment van ontwerp tot het moment dat de software wordt uitgevoerd. Zo kan bijvoorbeeld een variatiepunt al in een ontwerp worden gedefinieerd, terwijl de keuze voor een variant pas tijdens installatie wordt gemaakt.



Figuur 3 Verbergen variatie achter component met variatiepunt

Koppeling en cohesie

Sterk gerelateerd aan het scheiden van verantwoordelijkheden is het redeneren over relaties tussen componenten. Het principe is dat componenten zo veel mogelijk ontkoppeld (gescheiden) zouden moeten zijn van andere componenten, maar zelf een hoge mate van cohesie (samenhang) vertonen [Stevens1974]. De mate van koppeling tussen componenten wordt bepaald door het aantal relaties tussen componenten, de complexiteit en de intensiteit van deze relaties, zowel functioneel als technisch. Door het reduceren van de koppeling tussen componenten kunnen ze beter worden begrepen en los van elkaar worden ontwikkeld en hergebruikt. Cohesie heeft te maken met de binding tussen elementen in een component. Elementen in een component zouden zoveel mogelijk moeten bijdragen aan dezelfde functie, aangezien zij anders zelf ook herbruikbare componenten zouden zijn. Het herbeleggen van verantwoordelijkheden in een verzameling componenten en relaties kan leiden tot een lagere koppeling en een hogere cohesie, maar kent zijn grenzen. Uiteindelijk zullen componenten moeten worden geclustered tot groepen van componenten die sterk samenhangen maar weinig relaties met andere groepen van componenten hebben. In een object-georiënteerde ontwikkelwijze betekent dit het clusteren van klassen tot componenten. Deze componenten kunnen weer worden geclustered tot grotere componenten, die bijvoorbeeld services genoemd zouden kunnen worden. Belangrijk is dus om te beseffen dat er meerdere granulariteiten van componenten zijn (zie figuur 4).



Figuur 4 Componenttypes van verschillende granulariteit

Naast het aantal relaties tussen componenten en de intensiteit van deze relaties zijn er ook andere vormen van koppeling:

- Lokatie: als componenten aan elkaars lokatie gekoppeld zijn dan is het relatief moeilijk hun lokatie te veranderen, bijvoorbeeld om redenen van load balancing of fail-over. Componenten kunnen zijn gekoppeld aan eenzelfde proces, node of netwerk.

- Tijdstip: als componenten aan elkaars tijdstip van beschikbaarheid gekoppeld zijn dan verkleint dat de beschikbaarheid en betrouwbaarheid van het aanroepende component.

- Kennis: als componenten gekoppeld zijn aan kennis van andere componenten dan kunnen deze andere componenten moeilijker evolueren of migreren zonder afnemende componenten aan te passen. Er zijn verschillende aspecten waar andere componenten kennis over kunnen hebben zoals

het fysieke adres, de naam, de programmeertaal, de componenttechnologie, de interne opbouw, de signatuur van de interface, het gegevensmodel, het gegevensformaat en de hoeveelheid gevraagde of opgeleverde gegevens van een component. Aanwezigheid van kennis van een aspect bij afnemende componenten maakt het wijzigen van dit aspect erg onderhoudsintensief.

Bovenstaande dimensies van koppeling zijn gerelateerd aan de realisatie van de verschillende granulariteiten van componenten (zie ook tabel 1). Op het laagste niveau kan worden gesproken over klassen, zoals beschikbaar in een object-georiënteerde programmeertaal. Alhoewel klassen los van elkaar zijn gedefinieerd zijn zij over het algemeen sterk aan elkaar gekoppeld doordat zij veel kennis over andere klassen hebben, en gekoppeld zijn aan de lokatie en het tijdstip van andere klassen. Op een iets hoger niveau kunnen fysieke componenten worden geïdentificeerd, zoals CORBA- of EJB componenten. Dit zijn typisch verzamelingen van samenhangende klassen die zich op fysiek verschillende nodes in een netwerk kunnen bevinden. Componenten communiceren op basis van een interface met elkaar. Zij hebben dus geen kennis van het fysieke adres, de programmeertaal en de interne opbouw van andere componenten. Daarnaast zijn zij ook ontkoppeld van de lokatie en mogelijk zelfs van het tijdstip van andere componenten. Op een nog hoger niveau bestaan services die typisch een verzameling van fysieke componenten zijn. Gebruikers van een service hoeven niets te weten over het bestaan ervan. Zij hoeven alleen kennis te hebben van bedrijfsbrede domeinmodellen die beschrijven welke gegevens en bedrijfsoperaties er beschikbaar zijn. Services kunnen zich verder ook op fysiek andere netwerken begeven, waarbij dan ook wel wordt gesproken over Web Services.

Koppeling→ *Componenttype↓*	**Lokatie**	**Kennis**	**Tijdstip**
Klasse	Proces Node Netwerk	Fysieke adres Naam Interne opbouw Signatuur interface Gegevensmodel Gegevensformaat Programmeertaal	Zelfde tijdstip
Component	Netwerk	Naam Signatuur interface Gegevensmodel Gegevensformaat Componenttechnologie	Zelfde tijdstip Ander tijdstip
Service			Zelfde tijdstip Ander tijdstip

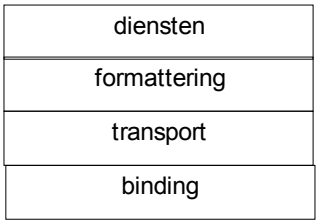
Tabel 1 Componenttypes en ontkoppeling

Koppeling heeft ook een sterke invloed op herbruikbaarheid, zoals het verleden aantoont. Daar waar object-oriëntatie in de jaren tachtig als het middel voor hergebruik werd gezien, veranderde deze visie in de jaren negentig. Het werd duidelijk dat objecten te klein, specifiek en gekoppeld zijn om ze los te kunnen herbruiken en componenten zouden de eenheid van hergebruik moeten worden. Inmiddels zijn we in een fase aanbeland waarbij sterk wordt getwijfeld aan de beloftes van component gebaseerd ontwikkelen. Alhoewel componenten interfaces bieden is er voor hun gebruik toch kennis over hun naam, precieze interface en gebruikte technologie nodig. Een service-gebaseerde benadering, die uitgaat van sterk ontkoppelde en relatief grote eenheden die ook voor de business betekenis hebben lijkt de nieuwe richting voor hergebruik te worden. In de vorm van Web Services wordt zelfs on-line hergebruik van diensten van andere bedrijven mogelijk.

Middleware en ontkoppeling

Middleware speelt een belangrijke rol bij het realiseren van architectuurprincipes; het kan worden gezien als een middel om componenten op bepaalde aspecten van elkaar te ontkoppelen. Het goed om te beseffen dat middleware typisch uit een aantal lagen bestaat (zie figuur 5). De onderste laag is de bindinglaag die een component en zijn aanroepen bindt aan de middleware. De transportlaag is verantwoordelijk voor

het verzenden van een verzoeken tussen componenten. Bovenop de transportlaag zijn er afspraken over de inhoud van een verzoek, waarbij er onderscheid bestaat tussen aanroep- en bericht-gebaseerde middleware. De bovenste laag van middleware bestaat uit toegevoegde waarde diensten.



Figuur 5 Middleware lagen

In tabel 2 is aangegeven hoe middleware componenten op bepaalde aspecten kan ontkoppelen. Ontkoppeling van lokatie wordt verzorgd door de transportlaag doordat het gegevens over bepaalde grenzen heen kan transporteren. Een queueingdienst zorgt ervoor dat berichten via een tussenliggende queue lopen waardoor componenten van elkaars tijdstip ontkoppeld zijn. Naming-, directory- en routeringdiensten ontkoppelen componenten van kennis over het fysieke adres en de naam van andere componenten. Transformatiediensten verbergen gegevensmodel en formaat van andere componenten. Een belangrijke mate van ontkoppeling wordt verzorgd door bindingsmechanismen. Deze ontkoppelen componenten van kennis over interne opbouw, programmeertaal, technologie en signatuur van andere componenten. Deze laatste vorm van ontkoppeling wordt verzorgd door dynamische aanroepmechanismen, die kan worden geboden door de middleware maar ook zelf middels een interface kan worden gedefinieerd. Flexibele datatypes of gegevensformaten zoals hashtables of XML zorgen ervoor dat componenten geen kennis hoeven te hebben van de precieze hoeveelheid gegevens die een component nodig heeft of oplevert. Onderscheidend is deze vorm van ontkoppeling echter niet aangezien alle middleware dergelijke voorzieningen expliciet of impliciet biedt; XML versturen kan met alle middleware.

Koppelingsaspect↓	**Ontkoppeling door**	**Middlewarelaag**
Lokatie	communicatiemechanisme	transport
Tijdstip	queueing dienst	diensten
Kennis		
- fysiek adres	namingdienst, directorydienst, routeringdienst	diensten
- naam	routeringdienst	diensten
- interne opbouw	binaire binding	binding
- programmeertaal	binaire binding	binding
- componenttechnologie	adapters	binding
- signatuur interface	dynamische invocatie	binding
- gegevensmodel	transformatiedienst	diensten
- **gegevensformaat**	transformatiedienst	diensten
- hoeveelheid gegevens	flexibele datatypes of flexibel gegevensformaat	formattering

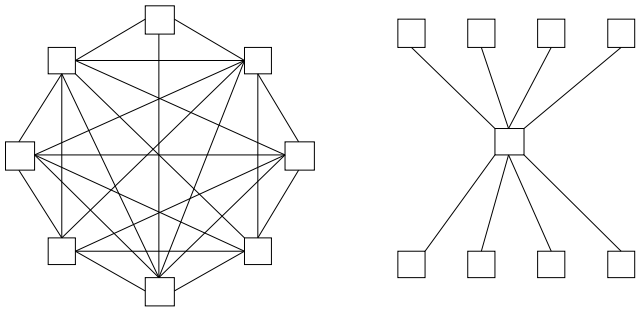
Tabel 2 Componenttypes en ontkoppeling

Soorten middleware

Op basis van tabel 2 is het mogelijk verschillende soorten middleware te positioneren op het gebied van ontkoppeling. Hierbij is het zinnig onderscheid te maken in aanroep- en bericht-gebaseerde middleware. De meest eenvoudige vorm van aanroep is tussen objecten en componenten binnen een proces, bijvoorbeeld binnen een propgrammeertaal. Deze vorm van communicatie is zeer snel maar vraagt om componenten die fysiek bij elkaar geplaatst zijn. Een iets verdere vorm van ontkoppeling wordt bereikt door zogenaamde interprocess communicatie (IPC). Uiteraard heeft deze ontkoppeling ook invloed op performance omdat berichten tussen processen moeten worden vertaald. Een nog verdere mate van ontkoppel-

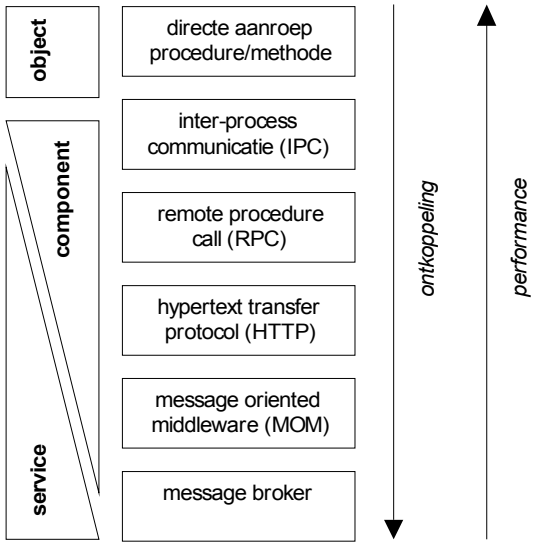
ing biedt remote procedure call (RPC) middleware, waarbij componenten zich ook op fysiek andere nodes kunnen bevinden. Naming en directorydiensten worden vaak met dit soort middleware meegeleverd. RPC middleware heeft een negatieve invloed op performance en betrouwbaarheid, omdat altijd alle componenten waarvan een systeem afhankelijk is beschikbaar moeten zijn. Een goede richtlijn is dan ook om nooit een RPC-koppeling te maken met systemen die een lagere beschikbaarheid hebben. Aan de andere kant wordt de schaalbaarheid van systemen wel vergroot doordat mogelijkheden tot replicatie ontstaan. Voorbeelden van RPC middleware (in de breedste zin van het woord) zijn Java RMI, DCE en CORBA.

De tweede categorie van middleware is bericht-gebaseerde middleware. Het meest triviale voorbeeld is HTTP; een minimale vorm van middleware die wordt gebruikt als infrastructuur voor het world-wide-web. Dit protocol onkoppelt een component van het lokale netwerk waarin het zich begeeft doordat in veel firewalls een opening voor dit protocol is gemaakt. Doordat dit echter niet voor alle firewalls geldt en een aantal firewalls ook op basis van inhoud doorgang van berichten verhinderen is het voordeel van HTTP minder groot. Het beste voorbeeld van bericht-gebaseerde middleware is zogenaamde message oriented middleware (MOM), die vaak ook diensten voor queueing bieden. Zoals eerder aangegeven is het deze mogelijkheid tot queueing die componenten ontkoppelt van elkaars tijdstip van beschikbaarheid. Een laatste categorie van middleware is die van message brokers, die een ultieme mate van ontkoppeling tot doel hebben door point-to-point verbindingen te vervangen door een centrale broker. Deze mate van ontkoppeling is haalbaar doordat zij over transformatie- en routeringsdiensten beschikken, die berichten transparant kunnen vertalen en naar de juiste services toesturen. Vaak wordt hiervoor dan ook een figuur vergelijkbaar met figuur 6 getoond. Door message brokers te combineren met message oriented middleware worden ook de ontkoppelingsvoordelen gecombineerd.



Figuur 6 Situatie voor en na het gebruik van een message broker

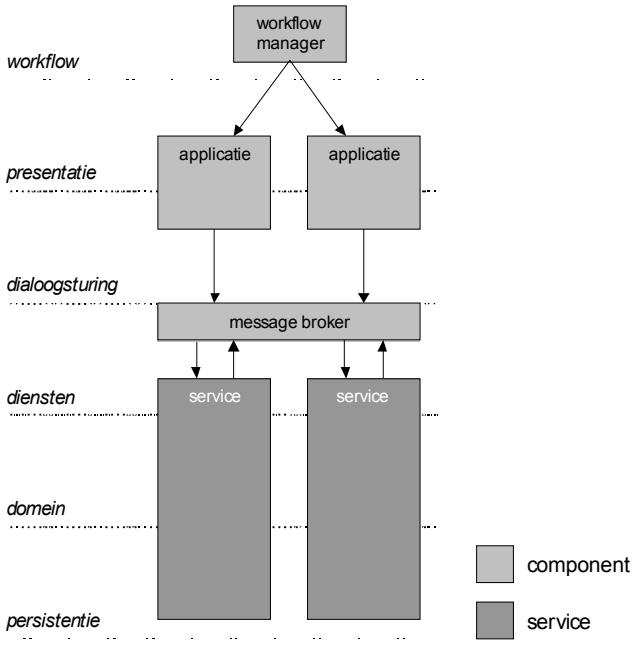
De standaarden (zoals XML, SOAP en UDDI) die voor de realisatie van Web Services kunnen worden gebruikt passen eigenlijk niet geheel in de opsomming van middleware omdat ze niet een eigen transportlaag afdwingen. Dit betekent dat ze gebruik kunnen maken van de ontkoppeling die andere middleware biedt. Daarnaast bieden ze ook middels UDDI een naming en directory servce en speelt XML een belangrijke rol bij Web Services waardoor ontkoppeling van de hoeveelheid invoer/uitvoer gegevens mogelijk is. Figuur 7 geeft een overzicht van de verschillende soorten middleware en de mate waarin zij geschikt zijn voor verschillende granulariteiten van componenten. Ook aangegeven in de figuur is de mate waarin zij componenten ontkoppelen en hun invloed op performance van communicatie indien op antwoord wordt gewacht.



Figuur 7 Communicatiemiddleware in perspectief

Service-gebaseerde architectuur

Op basis van eerder geschetst lagenmodel, het streven naar een service-gebaseerde architectuur en de bijbehorende keuzen voor middleware geeft figuur 8 een typische architectuur voor een organisatie weer. Applicaties richten zich op het ondersteunen van de gebruikerstaken en worden aangestuurd door een workflow manager. Alle bedrijfslogica is gedefinieerd in services, waarin mogelijk meerdere lagen zijn geïmplementeerd. Het gebruik van services door applicaties of door andere services verloopt via een message broker die ervoor zorgt dat er geen kennis over de services nodig is voor hun gebruik. Services kunnen dan ook met verschillende technologieën zijn gerealiseerd; door het definiëren van adapters kunnen zij vrij eenvoudig op de message broker worden aangesloten. Applicatieservers bieden de services automatisch eigenschappen voor ondermeer het afhandelen van transacties, beveiliging, load-balancing, fail-over. Het is verstandig voor de implementatie van de message broker eerst een bedrijfsgegevensmodel op te stellen. Dit voorkomt het definiëren en onderhouden van onnodige transformaties.



Figuur 8 Componenten en services in het lagenmodel

Conclusies

Voor het ontwikkelen van goede gedistribueerde systemen is een kwaliteitgedreven software-architectuur van belang aangezien daar fundamentele keuzen in worden beschreven. Een dergelijke architectuur beschrijft de verdeling van een systeem in componenten, hun relaties onderling en met de omgeving. Om te komen tot een goede software-architectuur is het van belang na te denken over het scheiden van verantwoordelijkheden, gewenste variatie en koppeling en cohesie. Een startpunt voor het scheiden van verantwoordelijkheden is een model waarin functionele lagen worden gedefinieerd. Voor het realiseren van relaties tussen componenten moet gekozen worden voor een soort middleware die past bij de gewenste mate van ontkoppeling.

Referenties

[Wagter2001] R. Wagter, M. Van den Berg, J. Luijpers, M. Van Steenbergen.: *DYA: snelheid en samenhang in business en ICT-architectuur*,Tutein Noltenius, 2001.

[Wirfs-Brock1990] R. Wirfs-Brock, B. Wilkerson, L. Wiener: *Designing object-oriented software*, ISBN 0-13-629825-7, Prentice Hall, 1990.

[Prins1997] R. Prins, A. Blokdijk, N.E. Van Oosterom: *Family traits in business objects and their applications*, IBM Systems Journal, No. 36, 1997.

[Stevens1974] W. P. Stevens, G. J. Myers, and L. L. Constantine: *Structured design*, IBM Systems Journal, 13(2):115-- 139, 1974.

Drs. Danny Greefhorst is werkzaam als IT Architect bij IBM Global Services en bereikbaar onder greefhorst@nl.ibm.com