Met de ontwikkeling van service oriëntatie tot dominant ontwikkelmodel zullen ontwikkelaars in alle opzichten hun blik moeten verruimen om te voorkomen dat zij de aansluiting met het nieuwe model gaan missen. In dit artikel zullen allereerst de verschillen tussen de ontwikkelstijlen toegelicht worden. Daarna worden drie best practices gepresenteerd voor het ontwikkelen van servicegeoriënteerde applicaties. Hierbij zal niet zozeer worden ingegaan op bijzondere implementatie aspecten (deze verschillen per platform), maar wel op de principes.

achtergrond

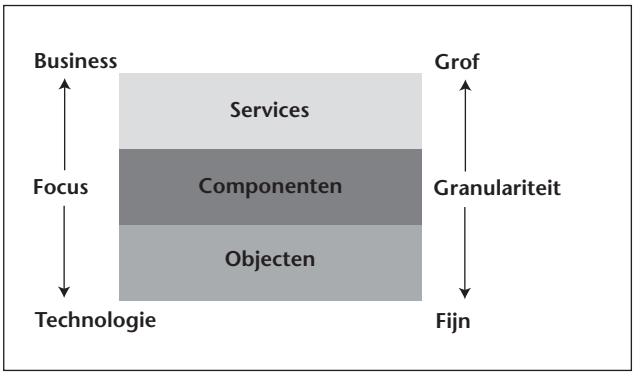
Nieuwe concepten, nieuwe dimensies

Blikverruiming door focus op services

Ontwikkelaars hebben de laatste jaren hard hun best gedaan om zich de concepten van component-based development (CBD) en objectoriëntatie (OO) als ontwerpmodellen voor software eigen te maken. Nu zullen zij ook de onderliggende concepten van de Service Oriented Architecture (SOA) goed moeten begrijpen en kunnen toepassen in de software die zij ontwikkelen.

SERVICES, COMPONENTEN EN OBJECTEN Vanaf de opkomst van objectgeoriënteerde talen als Smalltalk, Ada en C++ begin jaren negentig en pas echt sinds 1997 toen de taal Java het levenslicht zag, is objectoriëntatie het dominante ontwerpmodel geworden voor software systemen. Met de lancering van het .NET framework in 2000, wordt getracht om de wereldwijd drie miljoen Visual Basic ontwikkelaars objectgeoriënteerd te laten werken. Met het eerste lustrum van het .NET framework in zicht, kunnen we constateren dat objectoriëntatie zelfs voet aan de grond heeft gekregen binnen de Visual Basic community. Daarnaast is er de afgelopen jaren een sterk accent komen te liggen op het ontwerpen en ontwikkelen van op componenten gebaseerde systemen. Veel organisaties hebben de afgelopen jaren veel tijd en geld geïnvesteerd om de ontwikkelafdelingen op niveau te krijgen voor wat betreft objectoriëntatie en component-based development. En nu is er ineens de Service Oriented Architecture als *new kid on the block*. Wat betekent dit nieuwe gedachtegoed voor al deze inspanningen?

Wordt de in OO en CBD gemaakte software ineens *legacy* en is de investering in opleiding in OO- en CBD-denken overbodig geworden door SOA? Nee, zeker niet. Hierna zal worden uitgelegd dat service oriëntatie een dimensie toevoegt aan de mogelijkheden. Geheel in lijn met één van de oogmerken van service oriëntatie (het verder benutten van reeds gedane IT investeringen) blijken services, objecten en componenten prima samen te gaan. In figuur 1 is de plaats van services ten opzichte van componenten en objecten weergegeven, zoals we deze in veel situaties aantreffen.



FIGUUR 1. Services, componenten en objecten bij een service provider.

De services maken gebruik van de onderliggende componenten, die weer opgebouwd zijn uit objecten. Naar buiten toe werken de services volledig volgens de gangbare standaarden en protocollen (waarover verderop meer), intern zijn de meeste services opgebouwd volgens principes van objectoriëntatie en CBD. Het verschil tussen objecten, componenten en services is gelegen in de focus en de granulariteit. Waar services dicht bij de ondersteuning van business liggen, zullen objec-

ten dichter tegen de techniek aanleunen. Services zijn gedefinieerd in termen die door de business begrepen worden, en zijn tevens gemodelleerd naar bedrijfsprocessen, in tegenstelling tot bijvoorbeeld objecten en componenten.

Granulariteit (korreligheid) betreft de mate van detail waarmee een interface wordt aangeboden. Over het algemeen is de granulariteit van objecten en componenten fijner dan de granulariteit van services. Objecten en componenten bieden vaak zeer gedetailleerde en specialistische functionaliteit, terwijl services wat meer generieke functionaliteit bieden. Objecten zijn overwegend kleine data-eenheden gecombineerd met bijbehorende processen of *methods*. Componenten bieden een aanvullende abstractielaag en verpakking van meerdere objecten, waardoor zij minder korrelig zijn. Componenten worden soms ook wel gezien als een vroege incarnatie van services.

Services, componenten en objecten kunnen dus volledig complementair zijn. Het is echter niet zo dat services altijd bovenop componenten of objecten worden geïmplementeerd. We kennen bijna allemaal de voorbeelden van legacy-systemen of standaardpakketten voor ERP en CRM die via services ontsloten of geïntegreerd worden.

Verschillen met OO en CBD Als we voor wat betreft dataverkeer, koppeling en *state* naar de componenten en objecten kijken, zien we dat dataverkeer geschiedt door aan *method calls* argumenten mee te geven en wellicht als returnwaarden data terug te krijgen. Hierbij worden datatypen gebruikt (zoals strings, integers of samengestelde typen) voor het dataverkeer als argumenten en returnwaarden. Het aanroepen van deze *method calls* zou redelijk ad hoc zou kunnen verlopen en een method call wordt doorgaans direct beantwoord. De objecten en componenten 'draaien' meestal op één en hetzelfde platform.

In een gedistribueerde omgeving zien we vaak dat ontwikkelaars zich veel moeite moeten getroosten bij het uitrollen van clientcomponenten die communiceren met een servercomponent. Dit komt omdat goed rekening gehouden moet worden met mogelijke veranderingen in de servercomponent. Vaak geldt dat een aanpassing in code op de server ook een aanpassing op de client noodzakelijk maakt. Dit noemen we ook wel *tight coupling*. Tot slot zien we dat componenten en objecten doorgaans een *statefull* karakter hebben.

Wat opvalt in de terminologie rondom service-oriëntatie, is dat er niet meer gesproken wordt van *client* en *server*, maar van *consumer* en *provider*. Concreet hebben we te maken met een consument en een aanbieder van een dienst. Deze terminologie lijkt triviaal, maar is juist bedoeld om te breken met het beeld van dat er binnen een gedistribueerde, servicegeoriënteerde omgeving een clientrol en een serverrol is. Bij een op diensten gebaseerde architectuur is het juist zo, dat services zowel de rol van consument, als van aanbieder op zich kunnen nemen. In het ene geval vraagt de service een bepaalde dienst aan een andere service, in het andere geval biedt het zelf een service. Vergelijk het met de bekende peer-to-peer fileshare programma's als KaZaA of eMule: het ene moment worden er bestanden vanaf jouw machine gedownload door anderen, het volgende moment downloadt jij zelf bestanden van de machine van een ander.

Kenmerkend voor de link tussen serviceproviders en serviceconsumers is dat de communicatie tussen de twee door middel van berichten gebeurt. De communicatie komt veelal via een internetverbinding tot stand, via standaard-communicatieprotocollen. Er wordt hierbij dus bewust een systeemgrens gepasseerd. Bovendien zullen aan beide kanten van deze link waarschijnlijk verschillende bedrijven voor 'consumer en provider software' zorgen. Het berichtenverkeer tussen consumer en provider zal evenwel een vaste, vooraf gedefinieerde structuur hebben qua inhoud en communicatiepatroon. Gezien de onbepaalde tijdsduur dat een *request* van een *reply* wordt voorzien zal de communicatie een asynchroon karakter hebben. De serviceprovider zal niet voor alle servicerequests gegevens kunnen bewaren tijdens het afhandelen ervan. De service heeft dan dus een stateless karakter. Samenvattend kunnen we stellen services de volgende zaken impliceren:

- De *communicatie* tussen consumer en provider is *expliciet* gemaakt
- Deze communicatie verloopt door middel van *vooraf gedefinieerde berichten*.
- De data-uitwisseling in deze *berichten* is altijd verpakt.
- De communicatie heeft een *asynchroon* karakter.
- De *koppeling* tussen de provider en consumer is *los* kan separaat worden ontwikkeld.
- Tevens zijn de *grenzen* tussen consumer en provider *expliciet* gemaakt.
- Consumer en provider zijn *platformonafhankelijk*.
- Services hebben een *stateless* karakter.

Services	Componenten
Communicatie expliciet	***Communicatie impliciet***
Via berichten -> ingewikkelder	Via method calls -> eenvoudig
Data zijn verpakt in berichten	Data zijn typen
Asynchroon karakter	Synchroon karakter
Loosely coupled	***Tightly coupled***
'Client' en 'server' los te ontwikkelen	Vaak 'client' en 'server' tegelijk uitrollen
Grenzen expliciet	Grenzen lostrekken
Platformonafhankelijk	1 platform (os)
Stateless karakter	***Statefull karakter***

Tabel 1. De verschillen tussen services en componenten.

Voor de ontwikkelaar betekent dit concreet, dat hij of zij bekend moet zijn met de implicaties zoals hiervoor beschreven, en dat er gevoel moet zijn voor de verschillen met het ontwikkelen van componenten. Het goed ontwikkelen van services vereist andere kennis en vaardigheden dan tot dusver nodig was. Waarschijnlijk zal er een verschuiving plaatsvinden naar meer kennis van het orkestreren van services . De ontwikkelaar heeft daarbij een goed begrip nodig van de concepten van service-oriëntatie.

BEST PRACTICES VOOR ONTWIKKELAARS In Software Release Magazine 8/2004 is toegelicht dat de volgende concepten ten grondslag liggen aan service-oriëntatie:

- Ontkoppeling *(loose coupling)*
- Service contracten
- Asynchroniciteit

In de uiteenzetting van de verschillen tussen componenten, objecten en services hierboven hebben we deze elementen ook teruggezien. Om als ontwikkelaar optimaal te kunnen werken met bovenstaande concepten zijn er de volgende best practices:

1. Gebruik van patterns
2. "Contract first!"
3. Open standaarden

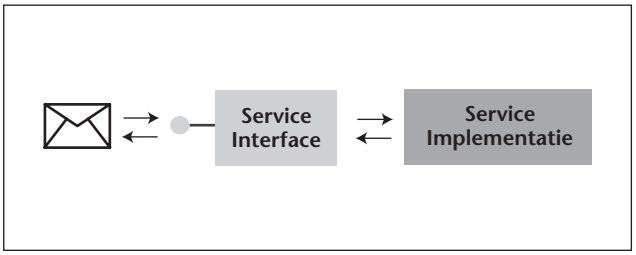
GEBRUIK VAN PATTERNS Patterns zijn handige hulpmiddelen voor ontwikkelaars en architecten, omdat ze:

- een simpel, werkend mechanisme beschrijven
- een gemeenschappelijke vocabulaire bieden voor ontwikkelaars en architecten
- de mogelijkheid bieden dat oplossingen beschreven worden als combinaties van patterns
- hergebruik van architectuur-, ontwerp- en implementatiekeuzes mogelijk maken.

De afgelopen jaren zijn er in Software Release Magazine een aantal artikelen verschenen die de toegevoegde waarde van patterns beschrijven. Het goede nieuws is dat er ook voor service oriëntatie een aantal patterns zeer bruikbaar zijn. Wij zullen hier een aantal patterns voor het voetlicht brengen die de concepten van service oriëntatie ondersteunen. Deze patterns richten zich vooral op concepten als *loose coupling* en asynchroniciteit, die voor 'klassieke' OO- en CBD ontwikkelaars lastig zullen zijn.

Service interface pattern

De losse koppeling in een SOA-architectuur wordt door een aantal maatregelen bewerkstelligd. Ten eerste hebben we al gezien dat de communicatie door middel van berichten geschiedt, zodat de programmatuurcode niet direct wordt benaderd. Ten tweede zal de implementatie van een service zich altijd achter een interface of façade bevinden en niet direct worden aangesproken. Dit wordt het *'service interface'* pattern genoemd. Het stelt ons in staat om wijzigingen in de eigenlijke implementatie van een service aan te brengen zonder de consumers daarmee te belasten. Tevens kan de consumer een andere provider kiezen die dezelfde service-interface geïmplementeerd heeft zonder al te veel aanpassingen te doen. Alleen de verwijzing naar de provider hoeft dan te worden aangepast. Het 'deel de klasse' (OO) versus 'deel de interface' (SOA) gaat hier nog een stapje verder dan in de Microsoft COM wereld.



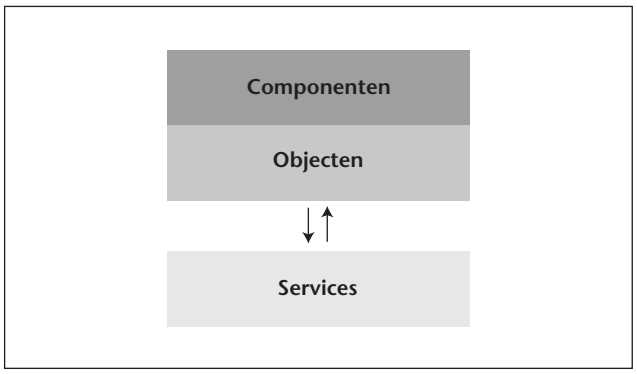
FIGUUR 2. Services worden altijd benaderd via een service interface.

De provider kan bovendien langs deze weg een compleet andere technologie gebruiken om de services te implementeren. De communicatie vindt natuurlijk wel plaats via standaardgebaseerde protocollen. Binnen een SOA zijn dit bij voorkeur open standaarden. Om de koppeling nog verder los te maken, zal alles uit de service wat met de implementatie te maken heeft, moeten worden verwijderd: alle gegevens, referenties, gedrag en niet-standaard datatypen welke specifiek zijn voor de implementatie of gebruikte onderliggende technologie.

Een stap verder gaat nog het samenstellen van een service uit meerdere services, zodat de service nog meer een het karakter krijgt van een business service. Deze kleinere deelservices worden niet noodzakelijk direct blootgesteld aan de buitenwereld. De consumer hoeft dan niet meerdere kleinere services te kennen en te benaderen. De interne kleinere servicecomponenten kunnen tevens gemakkelijker worden aangepast, hetgeen de agility weer laat toenemen.

Service gateway pattern

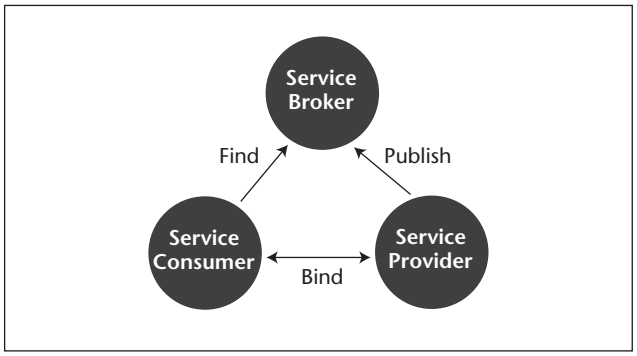
Bij het *'service gateway'* pattern wordt het gebruik van een service hier geëncapsuleerd in objecten, die weer door componenten worden gebruikt. In een SOA zullen diverse samenstellingen van deze twee combinaties van componenten, objecten en services worden gebruikt aan de provider-kant en aan de kant van de consumer. Het service gateway pattern wordt geïmplementeerd aan de consumer-kant, en is de evenknie van het service interface pattern dat aan de provider-kant geïmplementeerd wordt.



FIGUUR 3. Encapsuleren van services in objecten en componenten aan de consumer kant.

Service broker pattern

Nog een stap verder brengt ons het pattern van de *'service broker'*. Het gebruik van een service broker stelt de provider in staat om te bepalen naar welke onderliggende service een bepaald verzoek wordt gerouteerd. Deze onderliggende service is ofwel gespecialiseerd in het verwerken van bepaalde verzoeken, ofwel is op dat moment beschikbaar om een verzoek te verwerken. De service broker staat dan tussen de consumer en meerdere service providers in. Dit pattern toont het plaatje dat vaak getekend wordt om uit te leggen hoe een service oriented architecture er uitziet (zie figuur 4). Via standaardprotocollen als UDDI worden de service consumer en service provider door de service broker aan elkaar gekoppeld. De vergelijking met de Gouden Gids voor webservices is al regelmatig gemaakt, dus deze laten wij hier verder achterwege.

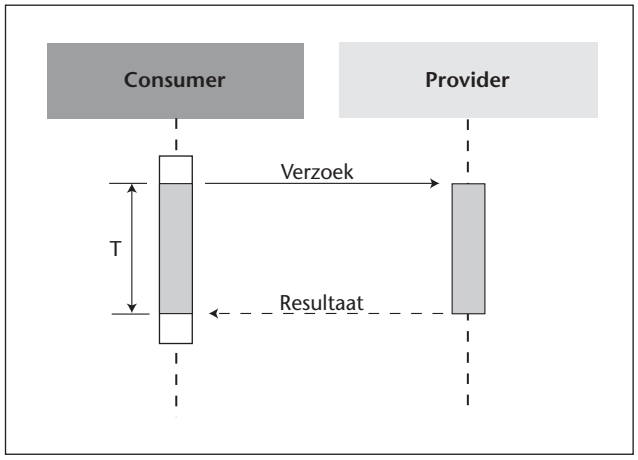


FIGUUR 4. De service broker.

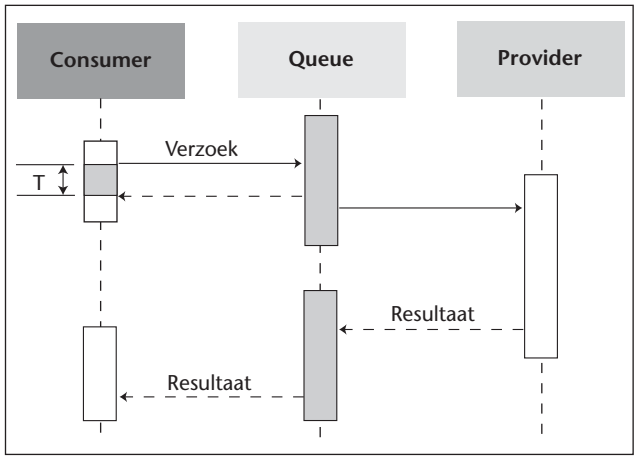
Patterns voor asynchroon programmeren

De meeste ontwikkelaars zijn bekend met en bedreven in het synchroon programmeren. Je roept een functie aan, en je blijft wachten op het resultaat van deze functie. In figuur 5 wordt dit schematisch weergegeven. Gedurende de periode dat de provider (functie) de verwerking uitvoert, blijft de consumer/client wachten (de periode T in de figuur). We zien ook dat de Consumer hetzelfde verzoek doet. Nu plaatst de Consumer echter alleen het verzoek in de Queue en wacht niet op het resultaat. De tijd T die daarvoor nodig is, is kleiner dan in het synchrone scenario. Het Consumer-proces kan nu met andere werkzaamheden verder gaan.

De Queue zorgt ondertussen dat het verzoek bij de provider aankomt en zal de Consumer daarvan op de hoogte stellen, wanneer het resultaat teruggestuurd is vanaf de Provider. Indien de Provider het verzoek snel kan afhandelen en het resultaat snel afhandelt zal het resultaat ook snel weer bij de Consumer terug zijn. De totale verwerkingstijd zal dan niet veel schelen met het bovenstaande synchrone scenario. Als de Provider niet snel kan reageren, kan de Consumer intussen met andere dingen bezig zijn. Het geringe performanceverlies en de extra complexiteit van de Queue wegen dan op tegen het niet blokkeren van het Consumer-proces.

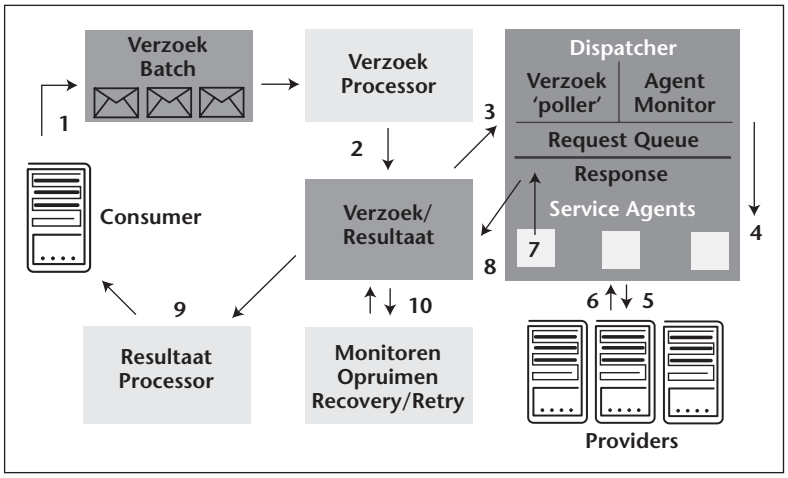


FIGUUR 5. Synchrone verwerking van een verzoek.



FIGUUR 6. Asynchrone verwerking van een verzoek

GROTE BOZE BUITENWERELD Het asynchrone karakter zullen we verder benaderen vanuit het perspectief van een ontwikkelaar die een service of een aantal services wil gaan gebruiken. Het gebruik van services zal worden geëncapsuleerd in zogenaamde service agents of service gateways. Dit zijn componenten die zijn gespecialiseerd in het gebruiken van services en waarin alle 'plumbing', die noodzakelijk is voor het benaderen van de services, is weggestopt. De service agent weet hoe de service benaderbaar is en kan met de service



FIGUUR 7. Voorbeeld van een asynchroon gebruiken van diverse services.

communiceren. In de ideale wereld zal je bij een request direct een response krijgen. Maar een beetje ontwikkelaar is zich bewust van de grote boze buitenwereld en zal zijn maatregelen willen treffen tegen bijvoorbeeld de volgende issues:

- is mijn vraag überhaupt bij de provider terecht gekomen?
- als ik niet meteen antwoord krijg maar op een later tijdstip, op welke vraag krijg ik dan welk antwoord?
- hoe lang moet of mag ik wachten op een response?

Als ik binnen een gestelde termijn geen antwoord krijg, wil ik de vraag nog een keer stellen. Het mag dan natuurlijk niet zo zijn dat ik twee keer een antwoord krijg dat verschillend is, of dat mijn aanvraag twee keer wordt verwerkt. Door een ID mee te sturen met de vraag en datzelfde ID ook in mijn antwoord terug te vinden, kan ik vraag en reply met elkaar matchen. De provider zal ook met behulp van het ID aan de slag kunnen. Hij kan kijken of de vraag met dat ID al verwerkt is en tevens bijhouden welk antwoord op welke vraag is gegeven. In dit scenario dienen de consumer en de provider er wel voor te zorgen dat voor een bepaalde message met een bepaald ID de inhoud hetzelfde is ofwel de message moet *idempotent* zijn. Als ik niet meteen antwoord krijg op een vraag kan ik ook op de consumer kant de vraag bewaren. Als ik de vraag voorzien had van een ID, kan ik de vraag later terugzoeken, indien er een antwoord gekomen is. Zo zijn vraag en antwoord technisch gezien met elkaar in overeenstemming te brengen. Tevens weet de consumer dan welke vragen er gesteld zijn.

Vaak zal de service agent een time-out mechanisme bevatten om de 'consumer' applicaties niet onnodig lang te laten wachten of in de echte asynchrone wereld helemaal niet eens wachten op antwoord (zie ook de uitleg hierboven). Figuur 7 toont een architectuur waar de consumer voorbereid is op het asynchroon ontvangen van antwoorden. Ook is de consumer in deze architectuur in staat om te zien op welke vragen antwoord is gekomen. Meerdere services worden benaderd via diverse service agents. In figuur 7 gebeurt het volgende:

1. Een consumer applicatie doet een aantal verzoeken in een batch.
2. De verzoekprocessor zal de verzoeken opslaan in de verzoek/resultaat-database. Per verzoek wordt naast de inhoud van het verzoek zelf, een referentie ID meegegeven en tevens welke service agent het verzoek moet verwerken.
3. De dispatcher zal in de verzoek/resultaat-database kijken welke nieuwe verzoeken er zijn.
4. Nieuwe verzoeken worden aan de beoogde service-agents aangeboden. Deze service agents zouden op meerdere servers actief kunnen zijn voor extra schaalbaarheid.
5. De service agents sturen de verzoeken naar de beoogde providers.
6. De providers sturen hun resultaat terug.
7. De service agents geven de ontvangen resultaten terug aan de dispatcher.
8. De dispatcher zet het resultaat in de verzoek/resultaat database.
9. De resultaat processor geeft de resultaten terug naar de consumer.
10. Een monitorproces controleert of er onbeantwoorde verzoeken zijn en kan retry's initiëren. Het monitorproces kan ook opruimwerkzaamheden uitvoeren.

"CONTRACT FIRST!" De basis voor een SOA blijft het heen en weer sturen van berichten. De berichten zijn voor wat betreft inhoud en formaat volgens een vooraf gedefinieerd schema bepaald, en zij vormen de elementen van het contract. Het contract is het samenspel van messages tussen service providers en service consumers om een bepaalde taak uit te voeren: de choreografie. In het vorige artikel uit deze reeks (Software Release Magazine nr. 2, 2005) is ruimschoots aandacht besteed aan de standaardisering rondom het samenspel van messages (compositie, choreografie en orkestratie).

Voor beide aspecten geldt de aloude vuistregel: ontwerp eerst de interface en het contract, doe daarna pas de implementatie (internationaal kennen we deze regel ook wel als "contract first!"). De diverse toolkits voor webservices moedigen het aan om een service contract te laten genereren op basis van de class. Feitelijk doe je hiermee hetzelfde als wat gedaan wordt bij de meer 'traditionele' gedistribueerde technologieën als RPC en DCOM: interface en implementatie zijn nauw gekoppeld, en niet los van elkaar te wijzigen of uit te breiden. Een wijziging in de implementatie leidt ook hier tot de noodzaak om de interface te wijzigen. Bovendien bevordert het eerst maken van een interface het goed nadenken over deze interface. Indien het contract gegenereerd

wordt, dan wordt er over het algemeen niet zolang nagedacht over de vormgeving van de interface. Tevens is het maar de vraag in hoeverre het gegenereerde contract nog goed leesbaar is.

OPEN STANDAARDEN De concepten van service oriëntatie kunnen in meerdere of mindere mate gefaciliteerd worden door een juiste toepassing van open standaarden. De bekendste en meest gebruikte open standaarden rondom messaging en SOA zijn uiteraard XML, SOAP, UDDI en WSDL. Daarnaast zijn er echter nog legio andere standaarden die het ontwikkelen van servicegeoriënteerde applicaties vereenvoudigen of dit tenminste beogen. Het grote probleem waar veel ontwikkelaars echter mee geconfronteerd worden, is het bepalen van welke standaarden wel, en welke standaarden niet relevant zijn.

Met name rondom de zogenaamde WS-Specifications (Web Service Specifications, ook wel *wizees* genoemd) zijn er erg veel keuzes te maken. De WS-Specifications zijn SOAP extensies op SOAP 1.1 en SOAP 1.2, en adresseren specifieke onderdelen van webservices (al dan niet als incarnatie binnen een SOA) die de toepassing van webservices meer robuust moeten maken. We kunnen hierbij denken aan specificaties voor beveiliging (WS-Security, WS-SecureConversation), processturing (WS-Coordination en WS-Transaction) en zogenaamde reliable messaging (WS-ReliableMessaging en WS-Reliability). Veel *wizees* vinden hun oorsprong in het gedachtegoed rondom CORBA, een architectuur en specificatie die vooral in Java-kringen veel gebruikt wordt voor het creeren, distribueren en beheren van gedistribueerde objecten in een netwerk. In dit opzicht is het opmerkelijk, dat behalve IBM, ook Microsoft zich een warm pleitbezorger van de *wizees* toont.

Voor ontwikkelaars is het echter een lastige keuze welke specificatie wel, en welke niet moet worden toegepast. Het zijn er erg veel, en het is haast ondoenlijk om je alle *wizees* eigen te maken. Bovendien wijzigen de wizees zeer regelmatig, zowel voor wat betreft de dominante standaard voor een probleemgebied, als de inhoud van de betreffende specificatie. Als vuistregels hanteren wij daarom:

- Code-generatoren en toolkits (bijvoorbeeld de uitstekende WSE 2.0 toolkit voor .NET) ontwikkelen zich razendsnel, en bevatten meestal vrij vroeg de meest bruikbare en meest toegepaste specificaties.
- De meeste webservices hebben slechts een fractie van de beschikbare specificaties nodig. Wees kritisch over welke specificaties wel, en welke niet gevolgd worden! Als minimale set zou je bijvoorbeeld WS-I BP (WS-Interoperability Basic Profile) kunnen gebruiken. Deze set borgt dat je de meest gangbare *wizees*, en vooral degenen die gaan over interoperabiliteit, toepast.

CONCLUSIE SOA geeft een nieuwe dimensie aan systeemontwikkeling met enorme nieuwe mogelijkheden voor het koppelen van systemen die op verschillende besturingssystemen of locaties draaien en die ook los van elkaar kunnen worden ontwikkeld. In de diverse verhandelingen over de SOA ligt de nadruk meestal op de voordelen voor de business. Om deze beloftes te kunnen inlossen dienen de ontwikkelaars zich bewust te zijn van een aantal aspecten van SOA die een belangrijke rol spelen bij het maken van SOA-software.

Naast de projectmatige, technische en architecturale investeringen, zijn er ook nog investeringen voor de ontwikkeling van kennis en ervaring van de ontwikkelaars die dit alles voor elkaar moeten boksen. Er wordt van de ontwikkelaars immers nogal wat gevraagd; zij zijn degenen die hun blik volledig moeten verruimen. Zij moeten verder kijken dan:

- object of component: services staan centraal.
- het specifieke project of de applicatie waar men aan werkt: hoe bouw je anders herbruikbare services?
- de bekende manier van ontwikkelen: synchroon met method calls kan niet meer.
- het platform waarvoor men voorheen ontwikkelde: het nieuwe platform heet SOA.

Beslissers dienen hier zeker rekening mee te houden, om te voorkomen dat grote ambities op strategisch of operationeel niveau sneuvelen, omdat de ontwikkelaars die het moeten waarmaken niet in de gelegenheid zijn gesteld deze paradigmawijziging te maken, of de kans niet hebben gekregen deze wijziging toe te passen, bijvoorbeeld in de vorm van extra tijd voor analyse en ontwikkeling.

Hein Vader is managing consultant bij Capgemini, en is werkzaam als software architect en senior software engineer. Loek Bakker is als senior consultant werkzaam bij Capgemini, en is gespecialiseerd in architectuur en integratievraagstukken.