*Performance is één van de belangrijkste kwaliteitseisen die we aan een systeem stellen. Als er al expliciet over performance tijdens het ontwerp van een systeem wordt nagedacht blijkt het vertalen van performance-eisen naar architectuur en ontwerp moeilijk. Dit artikel geeft daarom een aantal richtlijnen voor het bereiken van een goede performance.*

**Softwareproductkwaliteit**

Als er iets is waar we het met zijn allen over eens zijn is dat we goede software willen maken. Goede software voldoet aan impliciete en expliciete eisen van de alle betrokken partijen [1]. Naast functionele eisen spelen ook kwaliteitseisen een belangrijke rol. Het is dan ook zaak in een zo vroeg mogelijk stadium de kwaliteitseisen in kaart te brengen, te kwantificeren en te borgen in de architectuur [2]. Alhoewel de focus in dit artikel op software ligt is het goed om te beseffen dat het redeneren over softwarekwaliteit alleen kan als ook de andere aspecten van een systeem worden beschouwd. Een holistische aanpak voor systeemontwerp redeneert vanuit de business eisen en heeft naast software ook aandacht voor hardware en middleware.

Een uitdaging bij het in kaart brengen van kwaliteitseisen is het bepalen van de belangrijkste eisen, aangezien het niet haalbaar is om alle mogelijke kwaliteitseisen te inventariseren en te kwantificeren. Een goed middel voor het vinden van de belangrijkste kwaliteitseisen is het organiseren van een workshop met alle partijen die bij het systeem betrokken zijn.

Performance heeft betrekking op het gebruik van tijd en hulpbronnen door de software en blijkt vrijwel altijd een belangrijke kwaliteitseis. Het is dan ook zaak goed inzicht te hebben in het specificeren van performance-eisen en het borgen van deze eisen in de architectuur en de realisatie van de software. Daarnaast moet er zo vroeg mogelijk een goede indicatie worden verkregen van de performance van een systeem, aangezien het aanbrengen van wijzigingen dan nog relatief het meest eenvoudig is.

Voor het specificeren van performance-eisen is het noodzakelijk na te denken over indicatoren die de performance verder kwantificeren en meetinstrumenten die ingezet kunnen worden om de performance te bepalen [1]. Indicatoren kunnen bijvoorbeeld betrekking hebben op responsetijd, doorlooptijd, doorvoersnelheid, deadlines, gebruikersaantal en resourcegebruik. Het kwantificeren van indicatoren komt neer op het toekennen van getallen uitgedrukt in indicator-specifieke eenheden, ook wel budgetten genoemd. Voor het kunnen toekennen van budgetten is het van belang van te voren aan te geven wat de door de business verwachte volumes zijn. Vaak zijn dergelijke volumes en budgetten niet bekend en kan het helpen om gewoon een aantal voorbeeldscenario´s uit te werken. De totale budgetten kunnen dan worden verdeeld over de verschillende lagen van het systeem en uiteindelijk worden toegekend aan individuele componenten.

De budget- en volumegegevens kunnen in een vroeg stadium van een ontwikkeltraject al in een statische analyse worden gebruikt om potentiële performance problemen in kaart brengen. Naar mate een project meer vordert wordt met queueing modellen en eventueel simulaties een nog beter gevoel voor de performance gekregen. Als een eerste versie van de software eenmaal draait kan er worden gemeten en kunnen eerdere schattingen worden vervangen door metingen, uiteraard rekening houdend met de mate waarin de ontwikkelomgeving representatief is voor de productieomgeving.

Een belangrijk onderdeel voor performance-gedreven ontwerp is het hebben van goede richtlijnen voor het borgen en verbeteren van de performance van de software. Zulke richtlijnen moeten al toepasbaar zijn op architectuurniveau aangezien daar de belangrijke beslissingen over de software worden vastgelegd. Geïnspireerd door de patronen-cultuur [3, 4, 5, 6] beschrijft dit artikel daarom veelvuldig toegepaste oplossingen voor performance problemen: performance patronen.

**Performance patronen**

Patronen voor het verbeteren van de performance van software zijn onder te verdelen in tien categorieën (zie tabel 1). Omdat de ruimte in dit artikel beperkt is zullen per categorie slechts een aantal voorbeeldpatronen worden behandeld. Voorbeelden uit de praktijk worden gebruikt om het gebruik van

het patroon te illustreren. Merk op dat per situatie moet worden bepaald in hoeverre een patroon toepasbaar is aangezien het klakkeloos toepassen van patronen niet automatisch tot een goede oplossing leidt.

| Categorie | Voorbeeld patronen en anti-patronen |
|---|---|
| Voorkom overhead | batching<br>value object |
| Bereid je voor | preallocatie<br>preprocessing<br>pooling |
| Doe tegelijk iets anders | multithreading<br>kleine en korte locks<br>asynchrone communicatie |
| Wees niet onnodig flexibel | meta-objectmodel<br>flexibele datatypes<br>dynamische aanroepmechanismen |
| Wees optimistisch | optimistic locking<br>tijdelijk uitschakelen integriteitscontroles |
| Doe het zo optimaal mogelijk | hergebruik objecten<br>delegeer naar hardware<br>gebruik snelle algoritmen |
| Doe niet meer dan nodig | lazy evaluation<br>partial processing |
| Doe niet meer dan je kan | beperk inkomende verzoeken<br>staging |
| Scheid tijd-kritische delen | scheid real-time processen<br>datawarehousing |
| Hou het zelf in de hand | directe databasetoegang<br>gebruik lager niveau programmeeromgevingen |

Tabel 1 Performance patronen

*Voorkom overhead*

De basis voor veel performanceproblemen ligt in het feit dat bepaalde functies relatief veel overhead opleveren. In het bijzonder zijn netwerk- en diskcommunicatie vaak een belangrijke botteneck. De belangrijkste patronen voor het voorkomen van overhead richten zich dan ook op deze categorieën. Voor het beperken van netwerkcommunicatie is het belangrijk zowel te kijken naar het aantal maal dat gegevens worden gecommuniceerd, als naar de hoeveelheid verzonden gegevens. De eerste wordt beïnvloed door de vertraging van het netwerk, terwijl de tweede gerelateerd is aan de doorvoersnelheid van het netwerk. Een belangrijk patroon voor het reduceren van het aantal gegevensuitwisselingen is "batching". Het patroon stelt voor gegevensuitwisselingen te groeperen in "batches"; groepen van gegevens die toch altijd samen worden gebruikt. Een specifieke variant van dit patroon is het "value object" patroon [7] waarbij benodigde gegevens worden ingepakt in een speciaal (Java) object.

Bij een reisbureauketen is besloten om de bestaande DOS-gebaseerde applicatie te vervangen door een Windows applicatie. De nieuwe applicatie wordt gebouwd in een 4GL omgeving waarbij snel schermen kunnen worden gedefinieerd en met "data-aware controls" direct kunnen worden gerelateerd aan databasevelden. Wat in eerste instantie een goede manier lijkt om snel een applicatie te ontwikkelen blijkt later desastreuze gevolgen te hebben voor de performance van de applicatie. Zo wordt er voor de communicatie van de applicatie vanuit de vestigingen gebruik gemaakt van een WAN waarnaar een ISDN verbinding ligt van 64Kb. De "slimme" controls lijken niet gemaakt voor netwerken met een dergelijk grote latency. In het bijzonder blijken er voor één scherm enkele tientallen gegevensuitwisselingen nodig te zijn. Het gevolg is dat gebruikers soms wel enkele tientallen seconden

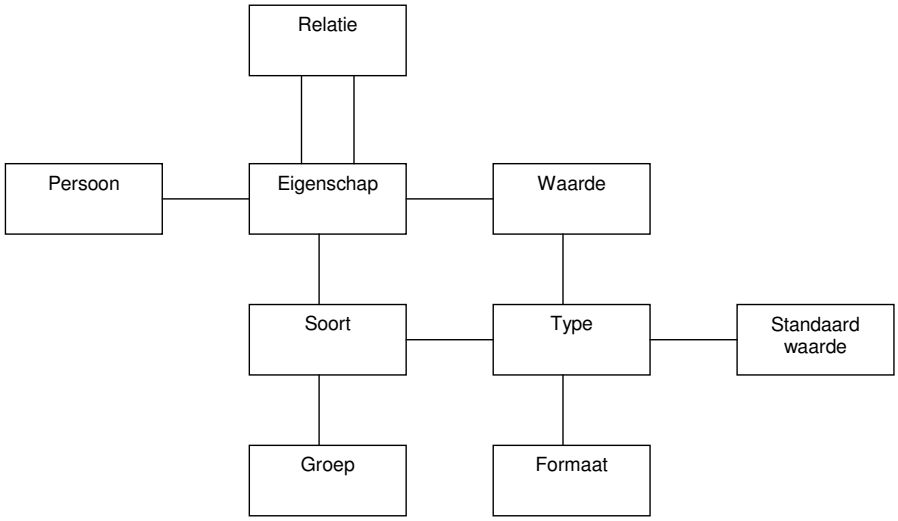
op een antwoord moeten wachten. Achteraf had deze applicatie beter als web-applicatie kunnen worden ontwikkeld, waarbij een scherm in één keer als HTML pagina wordt verzonden.

*Bereid je voor*

Een groot deel van de performanceproblemen kunnen worden opgelost door een aantal voorbereidende maatregelen te treffen voordat het echte werk begint. Denk hierbij bijvoorbeeld aan het alvast alloceren en initialiseren van objecten en het alvast voorbewerken van uit te voeren code. Een groot deel van dit werk wordt uit handen genomen door tools zoals (pre/JIT) compilers die code kan “inlinen”, “unrollen” en voorcompileren. Meer op architectuurniveau is het mogelijk om van objecten die veel gelijktijdig worden gebruikt alvast een aantal instanties te creëren. Dit patroon, wat ook wel bekend staat onder de naam “pooling”, is bekend geworden door databasemiddleware, applicatieservers en webservers. Zo kan een gemiddelde webserver een aantal HTTP-processen alvast opstarten zij klaar staan om verzoeken van meerdere gebruikers af te handelen.

*Doe tegelijk iets anders*

Een groot deel van de activiteit in software kan parallel plaats vinden. Het is daarom zonde om te wachten op een bepaalde activiteit en verstandig om meerdere computers, processen of threads naast elkaar het werk uit te laten voeren. Er moet dan wel voor worden gezorgd dat deze processen een gelijke hoeveelheid werk hebben en dat zij elkaar zo min mogelijk in de weg zitten. Voor dat laatste is het bijvoorbeeld gebruikelijk om “locks” zo klein mogelijk (bijvoorbeeld per record en niet per tabel) en zo kort mogelijk vast te houden. Een ander belangrijk patroon is dat van asynchrone communicatie, waarbij een component niet wacht op het antwoord van een verzoek. Dit patroon is de grootste reden voor de populariteit van “message oriented middleware”, tegenover synchrone middleware zoals object request brokers.

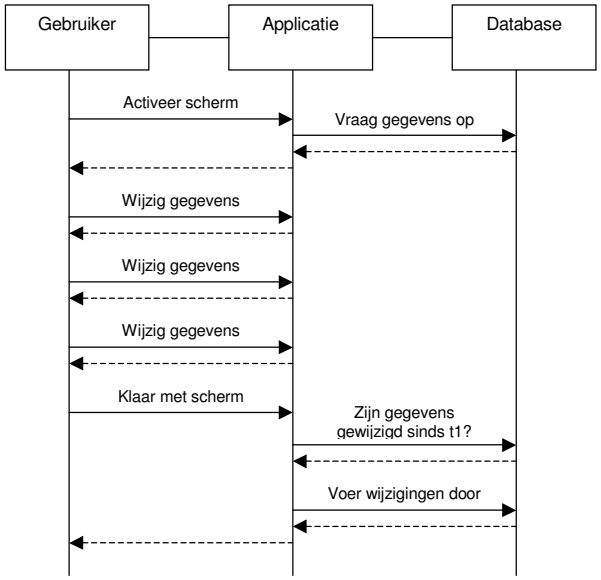



Figuur 1 Voorbeeld van het meta-objectmodel anti-patroon

*Wees niet onnodig flexibel*

Natuurlijk is het belangrijk om te kijken in hoeverre de software in de toekomst nog kan wijzigen of zal moeten opschalen, maar het inbouwen van extreem veel flexibiliteit heeft ook zo zijn prijs. Het motto is dus; bouw alleen die variabiliteit in de software in waarvan vrijwel zeker is dat deze binnen afzienbare termijn kan wijzigen. Er zijn in de praktijk veel voorbeelden van anti-patronen die te veel flexibiliteit nastreven en daardoor de performance sterk negatief beïnvloeden. Een veel voorkomend anti-patroon is die van een objectmodel dat eigenlijk meer een metamodel is (zie bijvoorbeeld figuur 1). In het extreme geval bevaten zo’n model klassen als “object”, “relatie” en “attribuut”, waardoor eigenlijk vrijwel de gehele

wereld gemodelleerd kan worden. De performance-impact ligt erin dat voor het creëren van een domeinobject op basis van zo'n model vaak veel relaties moeten worden gelegd, die richting database vaak worden vertaald naar dure join operaties. Flexibiliteitsmechanismen geboden door ontwikkelomgevingen en middleware hebben ook vaak een negatieve invloed op performance. Zo zijn flexibele datatypes zoals COM variants en CORBA any's prima voor generieke operaties, maar blijkt het versturen en in- en uitpakken van zulke datatypes vaak veel performance overhead op te leveren. Een ander anti-patroon in dergelijke omgevingen is het gebruik van dynamische aanroepmechanismen zoals COM Automation of Java reflectie, aangezien deze vaak om veel extra indirecties vragen.




Figuur 2 Optimistic locking patroon

*Wees optimistisch*

Een hoop tijdswinst blijkt te behalen door er gewoon vanuit te gaan dat een functie goed gaat en pas achteraf te controleren of dit ook echt het geval is. Het belangrijkste voorbeeld van deze categorie is het "optimistic locking" patroon (zie bijvoorbeeld [8]) waarbij gegevens in een database tijdens een gebruikerstransactie niet worden gelocked in de database (zie figuur 2). Pas bij het doorvoeren van de transactie in de database wordt er kort gelocked. Hierdoor wordt het mogelijk om meerdere gebruikerstransacties tegelijkertijd van dezelfde gegevens gebruik te laten maken en hoeven gebruikers dus niet op elkaar te wachten. Uiteraard moeten conflicten wel aan de hand van tellers, tijdstempels, toestanden of condities worden bepaald. Een andere database-gerelateerd patroon in deze categorie is het tijdelijk uitzetten van de database referentiële integriteit tijdens het uitvoeren van een batch transactie. Referentiële integriteitscontroles blijken erg veel overhead op te leveren en het belangrijkste is dat de database aan het eind van de transactie weer in een consistente toestand is.

*Doe het zo optimaal mogelijk*

Als bepaalde functionaliteit moet worden uitgevoerd dan is het natuurlijk verstandig om te kijken hoe het zo efficiënt mogelijk zou kunnen. Zo moet het opnieuw creëren van objecten zoveel mogelijk worden vermeden en gegevens op een plaats worden bewaard waar ze snel toegankelijk zijn. Een andere oplossingsrichting voor optimalisatie is het laten uitvoeren van rekenintentieve taken zoals compressie, encryptie en rendering door specifiek daarvoor bedoelde hardware. Een open deur lijkt misschien het toepassen van snelle algoritmen, maar het is verbazingwekkend in hoeveel belangrijke code nog inefficiënte algoritmen worden toegepast.

Bij een uitzendbureau wordt een geheel nieuwe multi-tier Java applicatie ontwikkeld. Natuurlijk hoort bij

zo'n applicatie ook een geheel eigen look-and-feel. Er wordt dan ook veel werk gestoken in het opzetten van een raamwerk hiervoor. Een onderdeel van deze look-and-feel is een component dat bepaalt op welk volgend component focus wordt gelegd. Als de performance van dit component sub-optimaal blijkt zorgt het vervangen van de gehanteerde bubble-sort door een quick-sort voor het sorteren van gebruikersinterface elementen, toch snel enkele seconden tijdswinst op te leveren. Door het resultaat van deze sorteerslag ook nog eens voor een volgende aanroep te bewaren lijkt een gebruiker haast niet meer te hoeven wachten op het verplaatsen van de focus op een ander gebruikersinterface-element.

*Doe niet meer dan nodig*

Veel werk in applicaties gebeurt terwijl er eigenlijk helemaal geen noodzaak toe is. Om onnodige berekeningen te beperken kan het "lazy evaluation" patroon worden toegepast waarbij gegevens pas worden berekend op het moment dat het nodig is. Als de de hoeveelheid op te leveren gegevens ook moeten worden beperkt dan is het het "partial processing" [6] patroon te overwegen. In dit patroon berekent en levert de aanroep van een remote operatie alleen dat resultaat op wat op dat moment nodig is. Als meer gegevens nodig zijn dan worden die in een vervolgaanroep opgeleverd.

*Doe niet meer dan je aan kan*

Performanceproblemen kunnen worden veroorzaakt doordat er teveel gevraagd wordt van de software waardoor verzoeken op elkaar gaan wachten. Doel is dus om dergelijke wachtrijen te voorkomen door beperkingen te leggen aan de inkomende verzoeken. Die beperkingen kunnen betrekking hebben op het aantal gelijktijdige gebruikers, het aantal en de omvang van verzoeken of de hoeveelheid resources die bepaalde afnemers nodig hebben. Hiervoor zouden historische gegevens kunnen worden gebruikt. Een interessant patroon ter voorkoming van deze categorie van problemen is het "staging" patroon, waarbij een tussenliggend component wordt geïntroduceerd die een verzamelbak van inkomende verzoeken bijhoudt en deze in een voor een component behapbare frequentie naar het doelcomponent stuurt. Message oriented middleware zoals WebSphere MQ biedt een natuurlijk mechanisme voor het realiseren van zo'n staging area. De kracht van zulke middleware ligt erin dat het een ontkoppeling biedt tussen zendende en ontvangende partijen.

*Scheid tijd-kritische delen*

Er is vaak een onderscheid tussen tijdkritische en niet-tijdkritische delen in de software. Het is dan ook van belang dat tijdkritische delen niet gestoord worden door niet-tijdkritische delen en het fysiek scheiden van dergelijke delen in afzonderlijke componenten is dan ook aan te bevelen. Een sprekend voorbeeld is het scheiden van hard real-time processen van overige processen in technische software. De real-time componenten worden hierbij in een laag niveau programmeertaal gerealiseerd, terwijl andere delen prima in Java kunnen worden ontwikkeld. Een ander patroon in deze categorie is "datawarehousing" waarbij operationele databases worden ontlast van managementrapportages door het inrichten van een separaat datawarehouse.

*Hou het zelf in de hand*

Als het er echt op aan komt en een omgeving je teveel afschermt van de details waardoor performance negatief wordt beïnvloed is het zaak het heft in eigen hand te houden. Zo zagen we al eerder wat er kan gebeuren als je databasetoegang door "data-aware controls" laat regelen. In zo'n geval is het zaak zelf de toegang tot de database te verzorgen. Ook het dichter op fysieke hardware gaan zitten door het gebruik van lager niveau programmeertalen kan een belangrijk patroon zijn. Zo kan een derde generatie programmeertaal veel optimalere code opleveren als bijvoorbeeld een geïinterpreteerde SmallTalk omgeving. Nog meer op de hardware is het gebruik van assembly voor het direct aanspreken van de CPU operaties.

**Conclusies**

Het is belangrijk in een vroeg stadium al over de performance van een systeem na te denken. Een end-to-end aanpak voor systeemontwerp die begint met eisen uit de business speelt hierin een belangrijke rol. Architecten moeten goed begrijpen wat de invloed van een bepaalde keuzen op de performance van een systeem is. De bijdrage van dit artikel is het beschrijven van categorieën van patronen en anti-patronen om een performance-bewustzijn te kweken en concrete handvatten te geven voor verandering.

**Referenties**

[1] B. van Zeist, et.al.: Kwaliteit van software producten, praktijkervaring met een kwaliteitsmodel, Kluwer Bedrijfsinformatie, 1996.

[2] D. Greefhorst, J. Bosch: Kwaliteitgedreven software-architectuur, Informatie, Ten Hagen Stam, Jaargang 43, mei 2001.

[3] E. Gamma et al.: Design patterns, Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

[4] F. Buschmann et al. : Pattern-Oriented Software Architecture - A System of Patterns, John Wiley & Sons, 1996.

[5] D. Schmidt et al.: Pattern-Oriented Software Architecture - Patterns for Concurrent and Networked Objects, John Wiley & Sons, 2000.

[6] T.J. Mowbray, R.C. Malveau: CORBA Design patterns, John Wiley & Sons, 1997.

[7] N. Kassem et al.: Designing Enterprise Applicatons with the Java 2 Platform, Enterprise Edition, 2000.

[8] IBM ITSO: *Design and Implement Servlets, JSPs, and EJBs for IBM WebSphere Application Server*, IBM RedBook SG24-5754-00, Augustus 2000.

*Drs. Danny Greefhorst is werkzaam als I/T Architect bij IBM Global Services*