Java 2, Enterprise Edition

Evaluatie van een ontwikkelplatform voor enterprise-applicaties

DOCTORAALSCRIPTIE	
Auteur:	M.V.N. Seijbel
Studie:	Bestuurlijke Informatica
Instelling:	Erasmus Universiteit Rotterdam Faculteit der Economische Wetenschappen Vakgroep Informatica
Scriptiebegeleiders:	Dr. Ir. J. van den Berg Drs. M.T.H. Polman

Inhoudsopgave

Lijst met figuren

Voorwoord

Na reeds jaren gewerkt te hebben in de ICT-branche, was het tijd om af te studeren. Met deze scriptie rond ik de studie *Bestuurlijke Informatica* af, waarmee er ook officieel een einde komt aan mijn studententijd.

Sinds ik in 1996 te maken kreeg met gedistribueerde technologie, heeft het onderwerp mijn interesse gekregen en niet meer verloren. Programma's die niet gebruikt worden door mensen maar door software, dat idee fascineerde me. Het onderzoek naar een ontwikkelplatform als J2EE heeft me vooral doen inzien dat de praktijk een inhaalslag gemaakt heeft op de theorie. De ideeën en concepten op het gebied van gedistribueerde technologie waren er al langer, maar met een concrete implementatie van die ideeën en concepten in een ontwikkelplatform wordt de technologie pas echt interessant.

Deze scriptie zou niet tot stand gekomen zijn zonder de hulp van een aantal mensen. Een woord van dank is dan ook op zijn plaats. Allereerst wil ik graag mijn begeleiders Jan van den Berg en Mark Polman bedanken voor hun suggesties en sturing bij deze scriptie. Ondanks de onregelmatige voortgang waren ze altijd bereid zich opnieuw in de scriptie te verdiepen. Verder bedank ik de collega's bij ORTEC en alle vrienden die het belang van mijn scriptie hebben ingezien en zich 'begripvol' hebben opgesteld. Twee collega's wil ik in het bijzonder bedanken. Op de eerste plaats Minghui Wang voor zijn hulp bij de implementatie van de testprojecten en daarnaast Ton van Welie voor zijn rol als onvermoeibare motivator. Tenslotte hulde aan mijn ouders, die me altijd mentaal en financieel gesteund hebben tijdens mijn studie.

Milan Seijbel,
Rotterdam, 14 mei 2003

1 Inleiding

1.1 Management samenvatting

Ondernemingen zijn er in de afgelopen jaren steeds meer achter gekomen dat bepaalde eisen op het gebied van marketing, dienstverlening en procesbeheersing alleen te realiseren zijn met behulp van gedistribueerde technologie [A1]. Dit komt mede door de populariteit en de volwassenheid van het World Wide Web. Het heeft zich ontwikkeld van een informatie distributieplatform tot een applicatieplatform. Corporaties kunnen diensten verlenen aan hun klanten die verbonden zijn met het Internet. Misschien nog wel belangrijker is het feit dat een corporatie een willekeurige persoon in de wereld, die verbonden is met het Internet, als een potentiële klant kan zien. Met de komst van het Internet kunnen bedrijven naast de business-to-consumer commercie ook de business-to-business integratie stroomlijnen [C1].

Dankzij deze drijfveren is de wereld van applicatieservers en bijbehorende tools uitgegroeid tot een miljardenindustrie [B21]. Met de opkomende populariteit van 'web services' is het eind nog niet in zicht. Marktonderzoeken geven aan dat de totale waarde van de 'web services'-markt zal toenemen tot ruim twintig miljard dollar in 2007 [B14].

Het ontwikkelen van enterprise-applicaties[1] is echter niet eenvoudig. Deze applicaties moeten duizenden, soms tienduizenden gebruikers tegelijkertijd bedienen. Dit stelt hoge eisen aan de performance, schaalbaarheid en betrouwbaarheid van een applicatie. Vaak vertegenwoordigen ze kritieke bedrijfsprocessen en kan het uitvallen van zo'n applicatie ernstige gevolgen hebben. Enterprise-applicaties op het Internet moeten vanuit een concurrentieoogpunt ook buiten kantooruren te gebruiken zijn. Omdat het Internet toegankelijk is voor iedereen, moeten deze applicaties de gegevens van het bedrijf en haar klanten beschermen en beveiligd zijn tegen onbevoegd gebruik. Door de toegenomen concurrentie, moeten bedrijven enterprise-applicaties ook nog goed, snel en goedkoop kunnen ontwikkelen en onderhouden [C1].

In de afgelopen jaren zijn een aantal ontwikkelplatformen gekomen, die een architectuur definiëren voor gedistribueerde applicaties. Deze platformen implementeren veel theoretische concepten, zoals transactiemanagement, beveiligingsmodellen en resourcemanagement, zodat een organisatie zich kan richten op het maken van de functionaliteit van de applicatie. Vandaag de dag zijn er twee grote spelers in de markt wat betreft ontwikkelplatformen voor enterprise-applicaties. Aan de ene kant is er het platformonafhankelijke Java 2, Enterprise Edition platform van SUN Microsystems en aan de andere kant is er het taalonafhankelijke .NET-platform van Microsoft. Samen hebben ze de ruime meerderheid van de applicatieserver-markt in handen [B14]. De

[1] Onder 'enterprise-applicaties' verstaan we in dit document 'gedistribueerde applicaties van ondernemingen'

onderzoeksgroep Gartner voorspelt dat in ieder geval voor 2008 de strijd om deze markt niet beslist zal zijn.

Op dit moment is J2EE het meest volwassen ontwikkelplatform voor gedistribueerde applicaties [C7]. Het is ruim vijf jaar operationeel en heeft meer dan vijftig aanbieders achter zich gekregen, waaronder grote namen als IBM, BEA en Oracle. Maar is het ook een goed ontwikkelplatform? Dit document beschrijft en toetst het J2EE-model aan de hand van algemene criteria, waaraan enterprise-applicaties moeten kunnen voldoen.

1.2 Probleemstelling

Onze interesse gaat uit naar de ontwikkeling van gedistribueerde applicaties met behulp van het Java 2, Enterprise Edition platform[2]. De centrale probleemstelling van deze scriptie kan als volgt geformuleerd worden:

Voldoet Java 2, Enterprise Edition als ontwikkelplatform voor enterprise-applicaties?

Om deze kwalitatieve vraag te kunnen beantwoorden, splitsen we de probleemstelling op in twee vragen. Biedt J2EE de vereiste basisfunctionaliteit voor het ontwikkelen van enterprise-applicaties en welke toegevoegde waarde levert het hierbij?

- ***Basisfunctionaliteit***
  Applicaties moeten voldoen aan relevante criteria, of "design goals". Deze "design goals" zijn afhankelijk van het applicatiescenario en niet voor alle toepassingen relevant. Zo zal een webapplicatie die vrij te gebruiken is en geen privé informatie bevat, geen hoge eisen stellen aan een criterium als 'beveiliging'.
  De basisfunctionaliteit van een ontwikkelplatform bestaat uit het bieden van de mogelijkheid om applicaties te ontwikkelen die aan hun "design goals" voldoen. Het J2EE-platform dient het mogelijk te maken om enterprise-applicaties te ontwikkelen die voldoen aan de "design goals" van gedistribueerde applicaties. Deze criteria zijn concreet en goed meetbaar.

- ***Toegevoegde waarde***
  Naast de verplichte basisfunctionaliteit speelt de toegevoegde waarde van een ontwikkelplatform nog een rol bij het totaaloordeel. In tegenstelling tot de basisfunctionaliteit is dit onderdeel moeilijker meetbaar. De volgende vragen komen naar voren bij het J2EE-platform:

  - *Welk abstractieniveau biedt het J2EE-platform bij het ontwikkelen van enterprise-applicaties?*

[2] Dit document bespreekt versie 1.3 van het Java 2, Enterprise Edition model en blikt vooruit op de voornaamste vernieuwingen in versie 1.4

- *Hoe flexibel is dit ontwikkelplatform voor de verschillende soorten applicaties?*
- *Hoe groot is de draagkracht van J2EE in de markt?*
- *Wat zijn de toekomstige verwachtingen rondom J2EE?*

1.3 Methodiek & structuur

Om een ontwikkelplatform voor enterprise-applicaties te kunnen evalueren, moeten eerst de criteria waaraan enterprise-applicaties moeten kunnen voldoen, in kaart gebracht worden. Ze dienen als uitgangspunt voor de verdere indeling van dit rapport. Eerst volgt een introductie van het J2EE-platform. Daarna bespreken we in detail de achterliggende theorie van het J2EE-model aan de hand van de genoemde criteria. Het rapport sluit af met een empirische toetsing, om te kijken of het J2EE-model ook in de praktijk haar belofte waar kan maken. Hiervoor gebruiken we een financiële toepassing *Index Tracking*.

Hoofdstuk 2 geeft een inleiding in de wereld van gedistribueerde systemen. Het beschrijft de behoefte aan dergelijke systemen en recente ontwikkelingen op het gebied van gedistribueerde technologieën. Daarna wordt het J2EE-model geïntroduceerd. Het hoofdstuk sluit af met een criterialijst voor enterprise-applicaties.

Hoofdstukken 3-7 gaan dieper in op de theorie achter J2EE aan de hand van de in hoofdstuk 2 genoemde criteria. Elk hoofdstuk bespreekt een criterium en de mogelijkheden om aan dit criterium te voldoen binnen het J2EE-model.

Hoofdstuk 8 beschrijft het *Index Tracking* model. Het begint met de uitleg van de beleggingsvorm Index Tracken. Vervolgens bepreekt het de mogelijkheid om Index Tracking als enterprise-applicatie te implementeren. Het hoofdstuk sluit af met de modellering van de applicatie.

Hoofdstuk 9 en 10 toetsen het J2EE-model aan de hand van de besproken theorie. Hierbij wordt het Index Tracking model gebruikt als voorbeeldimplementatie. Hoofdstuk 9 beschrijft de opzet van de testprojecten en hoofdstuk 10 bespreekt de ervaringen met J2EE aan de hand van deze testprojecten. Naast eigen ervaringen bevat hoofdstuk 10 ook ervaringen van anderen met het J2EE-platform. Deze informatie komt uit de literatuur.

Hoofdstuk 11 geeft een kwalitatieve conclusie van het J2EE-model op basis van de bevindingen uit de theorie en de praktijk. Tevens biedt dit hoofdstuk discussiepunten voor een eventueel verder onderzoek.

2 Gedistribueerde systemen

2.1 Inleiding

Een *gedistribueerd systeem* is een collectie van autonome computers, die met elkaar verbonden zijn over een netwerk en zich als een integrale, gegevensverwerkende faciliteit naar een gebruiker presenteert [A1]. Gedistribueerde systemen maken gebruik van distributed computing technologie. Een synoniem voor distributed computing is e*nterprise computing*[3] [A13].

Gedistribueerde systemen zijn over het algemeen complexer dan stand-alone oplossingen en daarom duurder om te implementeren en te onderhouden. Er moet dus een concrete behoefte zijn voor het gebruik van gedistribueerde systemen. Paragraaf 2.2 gaat dieper in op de drijfveer achter gedistribueerde systemen.

De technologie achter enterprise computing heeft zich steeds verder ontwikkeld. Er is een cyclisch verband tussen de behoefte aan gedistribueerde systemen en de technologische ontwikkeling van gedistribueerde systemen. Ervaring met de ontwikkeling en het gebruik van een technologie leidt tot het formuleren van nieuwe concepten en eisen. Deze eisen leiden op hun beurt weer tot de ontwikkeling van nieuwe systemen [A2]. Paragraaf 2.3 beschrijft de recente ontwikkelingen op het gebied van gedistribueerde technologieën.

Java 2, Enterprise Edition (J2EE) is een platform voor het ontwikkelen van gedistribueerde applicaties met de Java-programmeertaal [A7]. Het slaat een brug tussen de theorie en de praktijk, door het aanbieden van onderliggende diensten aan de applicatieontwikkelaar. Deze diensten implementeren concepten en algoritmes uit de theorie met betrekking tot resourcemanagement, beveiliging, concurrency, transactiemanagement, fouttolerantie en dergelijke [B4]. Met behulp van deze diensten wordt een applicatieontwikkelaar deels ontlast van de complexiteiten van enterprise computing en kan hij zich meer richten op de applicatielogica. In paragraaf 2.4 introduceren we het J2EE-platform.

Een gedistribueerd systeem heeft door haar architectuur een aantal karakteristieken of eigenschappen dat een centrale stand-alone systeem niet kent, zoals schaalbaarheid [A2]. De uitdaging zit in het goed omgaan met deze eigenschappen. Dit leidt tot zogenaamde *design goals* of criteria van gedistribueerde systemen. Het hoofdstuk sluit af met het formuleren van een reeks criteria, waaraan een enterprise-applicatie moet kunnen voldoen. We zijn geïnteresseerd in de toegevoegde waarde die het J2EE-model biedt bij het realiseren van deze criteria, en beschrijven globaal de middelen, of *sturingselementen* die J2EE hiervoor aanbiedt.

[3] In dit document zullen de woorden ‘enterprise’ en ‘gedistribueerd’ door elkaar gebruikt worden in verschillende contexten.

2.2 De behoefte aan gedistribueerde systemen

Enterprise computing begon ruim dertig jaar geleden en werd gebruikt door militaire, academische en onderzoeksinstanties. In de jaren tachtig vond er een verschuiving plaats naar commerciële organisaties. De recente behoefte aan gedistribueerde systemen is onder te verdelen in zakelijke drijfveren en technische mogelijkheden [A1].

2.2.1 Technische mogelijkheden

De techniek maakt het mogelijk om aan bepaalde behoeften te kunnen voldoen. Andersom geldt dit natuurlijk ook. Zonder de benodigde techniek kan aan bepaalde behoeften niet worden voldaan. We bespreken hier een aantal technische ontwikkelingen die belangrijk zijn geweest voor de populariteit van gedistribueerde systemen.

Het World Wide Web is niet langer alleen een distributieplatform voor statische informatie. Het is uitgegroeid tot een applicatieplatform. Desktop computers kunnen op het Internet deel uitmaken van een gigantisch gedistribueerd systeem.

Het is makkelijker om heterogene systemen te integreren. Het Internet is hier een mooi voorbeeld van. Miljoenen computers die allemaal verschillende hardware- en softwareplatformen hebben, werken succesvol samen.

De netwerksnelheden blijven groeien. In de jaren tachtig werd de klassieke client-server architectuur[4] populair door de toegenomen snelheid binnen een Local Area Network (LAN). Nu ligt de nadruk meer op het versnellen van *Wide Area Networks (WAN)*. Wide area netwerken beslaan een groot geografisch gebied, zoals een land of een continent.

2.2.2 Zakelijke motivaties

Bedrijven ontdekken steeds meer dat bepaalde eisen op het gebied van marketing, dienstverlening en procesbeheersing alleen te realiseren zijn met behulp van gedistribueerde technologie. We bespreken een aantal zakelijke drijfveren voor het adopteren van gedistribueerde systemen.

De ontwikkeling van het World Wide Web tot een platform voor applicaties, heeft het mogelijk gemaakt dat elke computer die aan het Internet is gekoppeld, gezien kan worden als een client van een gedistribueerd systeem. Ondertussen is het Internet veilig en betrouwbaar genoeg om als generiek platform te dienen voor het leveren van allerlei enterprise-applicaties. Het Internet is principe bereikbaar vanuit de hele wereld. Dit maakt het mogelijk voor elke corporatie om een willekeurige persoon in de wereld, die verbonden is met het Internet, als een potentiële klant te zien.

Door de globalisatie van het bedrijfsleven is de concurrentie ook toegenomen. Bedrijven moeten in staat zijn om snel in te spelen op marktbehoeften. Het gaat om het uitbrengen van nieuwe of aangepaste productlijnen, het oprichten van distributiekanalen in onverkende markten en in het algemeen het aanpassen van diensten op de wensen van de markt.

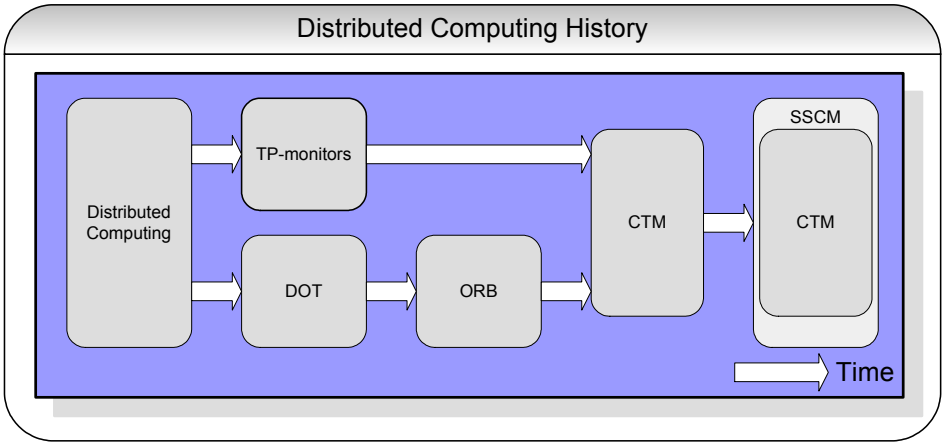
Dienstverlening aan klanten is een ander aandachtspunt. Klanten eisen steeds meer dat corporaties ze als één entiteit behandelen. Ze willen niet binnen een

[4] Zie paragraaf 2.3.1

grote corporatie met allemaal afzonderlijke afdelingen te maken hebben. Bovendien willen klanten meteen bediend worden, ook al vereist een interactie samenwerking tussen departementen.
Werknemersfaciliteiten zijn ook van cruciaal belang. Verkopers die veel reizen willen op afstand kunnen werken. Anderen willen de mogelijkheid hebben om thuis buiten kantooruren te werken. Hiervoor zijn communicatiefaciliteiten en applicaties nodig.
Kostenbeheersing is een algemene reden voor het gebruik van gedistribueerde oplossingen. Procesbeheersing en verspreiding van informatie gaan makkelijker met behulp van intranet- en Internetoplossingen, zoals e-mail, newsgroups en groupware. Het aanbieden van een Internet front-end applicatie voor diensten is een relatief goedkope en gebruikersvriendelijke oplossing. Voorbeelden hiervan zijn virtuele winkelcentra en applicaties voor Internetbankieren.
Al deze factoren vereisen een integratie van de afzonderlijke computerfaciliteiten in departementen en divisies.

2.3 Historie gedistribueerde technologieën

Gedistribueerde systemen dateren al van de zestiger jaren. Dat wil niet zeggen dat de technologie hierna heeft stilgestaan. De eisen aan dergelijke systemen veranderen in de tijd, evenals de toepassingsmogelijkheden. Door deze vernieuwde aandachtspunten verandert ook de onderliggende technologie [A2]. Figuur 2-1 geeft een overzicht van de recente ontwikkelingen op het gebied van gedistribueerde technologieën. Hier is te zien dat de ontwikkeling zich in eerste instantie splitst en later weer samenkomt. De komende paragrafen beschrijven de objecten uit de figuur.

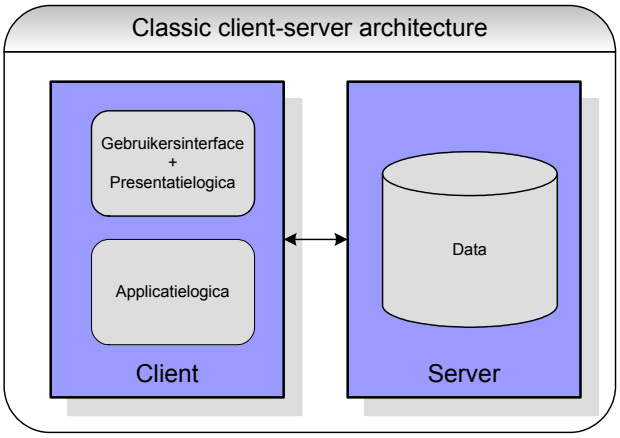


Figuur 2-1: ontwikkeling van gedistribueerde technologieën

2.3.1 Distributed Computing

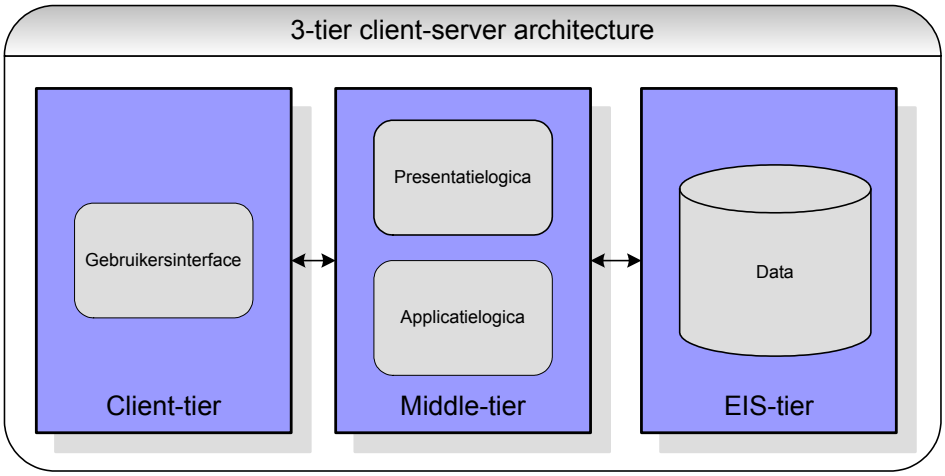
Distributed computing heeft te maken met het lokaliseren van data op afgelegen lokaties en het gebruiken van businesslogica in separate processen. Gedistribueerde systemen zijn gebaseerd op een *tierstructuur* [A7]. Een *tier* is een laag met een gemeenschappelijke functie die diensten verleent aan

gebruikers van de tier. De tierstructuur is een functionele scheiding en geen fysieke. Het kan echter wel voorkomen dat tiers zich op verschillende machines bevinden.



Figuur 2-2: Klassieke client-server architectuur

In het begin van de jaren negentig maakten veel informatiesystemen gebruik van de klassieke 2-tier client-server architectuur. De gegevens staan op een server en de client (vaak een desktop computer) benadert deze data [A16]. De ervaring leerde dat het ontwikkelen en onderhouden van een flexibel gedistribueerd systeem erg moeilijk is, wanneer je gebruik maakt van het 2-tier client-server model. Het veranderen van de businesslogica betekende een nieuwe installatie op elke client-machine.



Figuur 2-3: 3-tier architectuur

De *3-tier architectuur* of de *multitier-architectuur* is het meest gebruikte model bij gedistribueerde systemen. De eerste tier is de *client-tier*. Deze zorgt voor de representatie van het systeem naar de gebruiker toe door middel van de gebruikersinterface. De middelste tier bevat de applicatielogica. Hier zijn de "business regels" geïmplementeerd. De presentatielogica voor de (verschillende) clients staat ook op deze tier. Tenslotte bevat de derde tier de

gegevens en de mogelijkheid om deze te verschaffen. Deze tier wordt de *EIS-tier* of *Backend-tier* genoemd. EIS staat voor *Enterprise Information Systems.*
Het voordeel van deze opzet is dat de data op een centrale plaats staat, de performance kan worden geoptimaliseerd in de tweede tier (bijvoorbeeld door het plaatsen van extra servers) en de gebruiker een voor hem op maat gemaakte presentatie krijgt van de centrale data. Veranderingen in de applicatielogica hoeven alleen op de server doorgevoerd te worden.
De meeste ontwikkelingen op het gebied van gedistribueerde systemen hebben plaatsgevonden in de middelste tier. In dit hoofdstuk richten we ons daarom ook op deze tier.

2.3.2 Transaction Processing Monitors

Transaction Processing monitors (TP monitors) bestaan reeds zo'n dertig jaar en zijn ontwikkeld tot krachtige, snelle serverplatforms voor applicaties van kritieke bedrijfsprocessen. Een TP-monitor kan als een besturingssysteem van een applicatie gezien worden, omdat het de totale omgeving van de applicatie beheerst [A3]. Zo draagt de TP-monitor onder andere zorg voor transacties, resourcemanagement en fouttolerantie. Omdat TP-monitors al zo lang bestaan, is de achterliggende theorie door en door getest.
Transaction Processing monitors zijn gebaseerd op de traditionele 3-tier architectuur: de businesslogica in TP-systemen is geschreven met behulp van procedurele programmeertalen. Ze zijn dus niet object georiënteerd. Het aanroepen van een methode gebeurt met behulp van *Remote Procedure Calls (RPC)*. RPC is een mechanisme waarmee clients procedures van applicaties op de server kunnen aanroepen en waarbij het lijkt alsof deze procedures lokaal worden uitgevoerd.

2.3.3 Gedistribueerde objecttechnologie

De meest recente ontwikkeling binnen distributed computing is het gebruik van *distributed objects*[5]. Deze objecten kunnen een unieke identiteit en status hebben. Met behulp van *Distributed Object Technologies (DOT)* kunnen deze objecten gebruikt worden door client applicaties op andere machines. Bovendien is de software flexibel, uitbreidbaar en herbruikbaar dankzij het objectgeoriënteerde karakter. De basis van distributed object systems is een *Remote Method Invocation (RMI)* protocol. RMI vervangt het "procedurele" RPC en maakt de lokatie van objecten transparant voor clients. Het belangrijkste verschil is dat het aanroepen van een objectmethode gebeurt op een objectinstantie, en dat geldt niet voor een applicatieprocedure. Enkele voorbeelden van gedistribueerde objecttechnologie zijn CORBA, DCOM en JAVA-RMI [A3]. Paragraaf 4.3.3 gaat dieper in op de werking van het RMI-protocol.
De gedistribueerde objecttechnologie komt voort uit de 3-tier architectuur, die ook bij TP-monitors gebruikt wordt. Voorheen bestonden gedistribueerde systemen vaak uit terminals op de presentatie-tier, een applicatie geschreven in een procedurele taal zoals COBOL of PL/1 op de middle-tier, en een database

[5] De literatuur spreekt bij enterprise-applicaties ook wel van *businessobjecten*, wanneer businesslogica in objecten gestopt wordt.

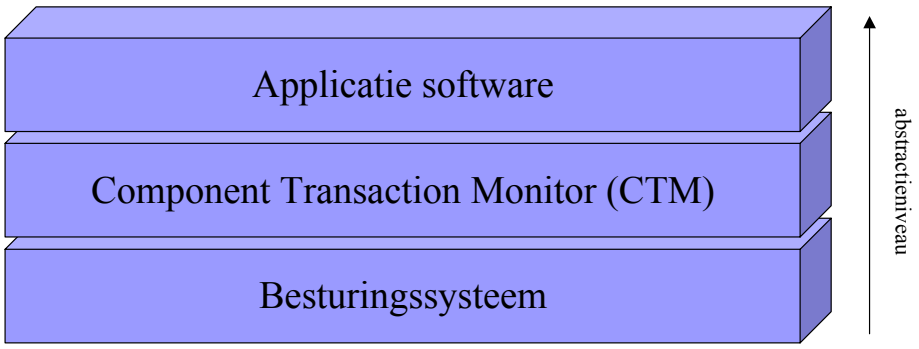
als EIS-tier. Met behulp van de gedistribueerde objecttechnologie het mogelijk om een geavanceerde grafische user-interface te combineren met businessobjecten en een relationele database. Ook kunnen businessobjecten op meerdere servers gezet worden, om een *n-tier* architectuur te creëren [A3].

2.3.4 Object Request Brokers

De besproken distributed object technologie bevat voornamelijk het protocol dat de communicatie regelt tussen gedistribueerde objecten. Een distributed object technologie kan gebruik maken van een *Object Request Broker*. Deze vormt de ruggengraat van de communicatie door clients te helpen bij het lokaliseren van en communiceren met andere objecten. De ORB kan nog meer diensten leveren. Zo gebruiken ze over het algemeen een *naming*-systeem voor het lokaliseren van de objecten, zorgen ze voor gedistribueerde *garbage collection* en *reference passing*. Echter, ze vormen geen volledig besturingssysteem voor een applicatie, zoals een TP-monitor. Het gebruik van diensten als transactiemanagement, concurrency, resourcemanagement, persistentie en beveiliging is niet automatisch en komt voor de rekening van de applicatieontwikkelaar. Dit vereist een grondige kennis van het systeem [A3].

2.3.5 Component Transaction Monitors

Een *Component Transaction Monitor (CTM)* is een gecombineerde vorm van de TP-monitor en de ORB-technologie. Zo heeft een CTM het beste van twee werelden: het is een dynamisch gedistribueerd object geörienteerd systeem, dat standaard alle TP-monitor diensten aanbiedt aan een businessobject. Dit zorgt ervoor dat applicatieontwikkelaars zich kunnen richten op de businesslogica in plaats van op de systeemarchitectuur, terwijl ze gebruik maken van de voordelen van OO-technologie.



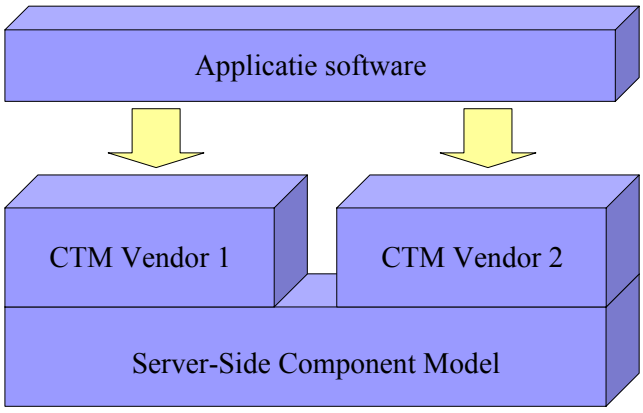
Figuur 2-4: Component Transaction Monitor (CTM)

Component Transaction Monitors vormen de omgeving waarbinnen server-side componenten draaien, net zoals TP-monitors dat doen voor procedurele applicaties. De behoefte aan CTM's is erg groot. Allerlei industrieën op het gebied van relationele databases, applicatieservers, webservers, CORBA en TP-monitors, hebben bijgedragen aan de ontwikkeling van CTM's. Hierbij

verschilt het scala aan diensten per CTM, mede doordat elke tak van de industrie zich richt op haar eigen behoeften.

2.3.6 Componentenmodellen

Om te zorgen dat verschillende objecten transparant met elkaar kunnen communiceren, moeten ze voldoen aan een standaard. Een *componentenmodel* definieert deze standaard. Het fungeert als een soort blueprint. Deze specificatie schrijft onder andere voor hoe om te gaan met het aanmaken van componenten, het beheren van geïnstantieerde componenten en het beschikbaar maken van componenten aan clients. Het gebruik van een componentenmodel garandeert dat een component dat voldoet aan deze standaard, zonder directe aanpassing gebruikt kan worden op een ander systeem dat hetzelfde componentenmodel ondersteunt.
Een speciale vorm van een componentenmodel is een *server-side componentenmodel,* dat een standaard specificeert voor gedistribueerde businessobjecten. Het verschil is dat een "standaard" componentenmodel gebruikt wordt voor componenten binnen een proces, terwijl een server-side componentenmodel componenten beheert tussen verschillende processen. Het gedistribueerde karakter staat dus centraal.



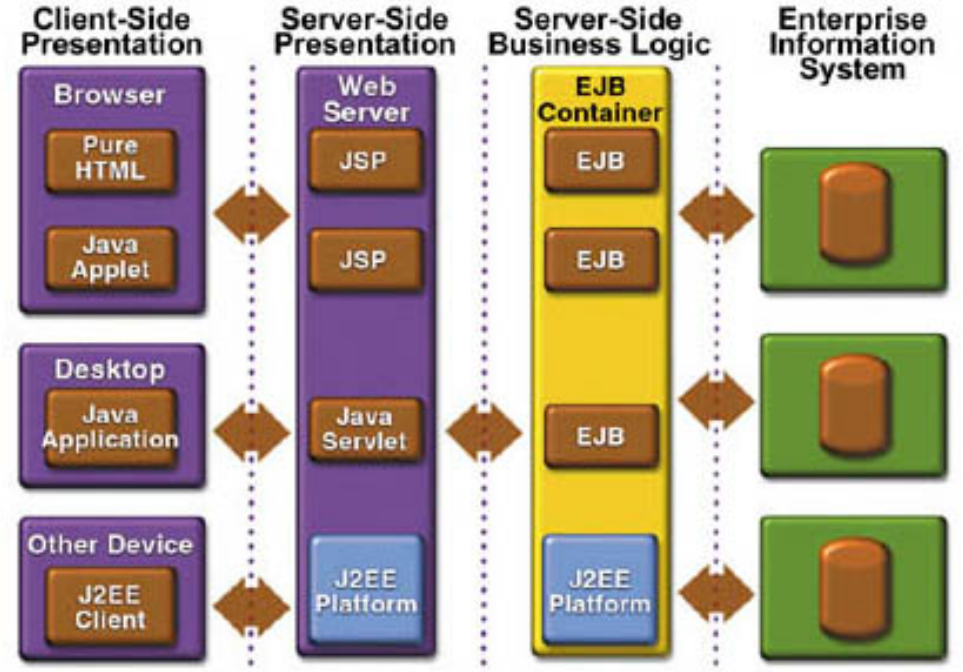
Figuur 2-5: Het Server-Side Componentenmodel

Component Transaction Monitors implementeren een server-side componentenmodel. Businessobjecten kunnen gebruikt worden door een CTM, zolang ze voldoen aan de modelspecificatie. Applicatieontwikkelaars kunnen dan, zonder directe aanpassingen aan hun componenten, overstappen op een CTM van een andere aanbieder die hetzelfde server-side componentenmodel implementeert.

2.4 Java 2 Enterprise Edition

Java 2 Enterprise Edition (J2EE) definieert een platform voor het ontwikkelen van gedistribueerde multitier enterprise-applicaties in de Java-programmeertaal [A3, C4]. Figuur 2-6 schetst een overzicht van het totale model. Er zijn vier tiers te onderscheiden: de client-tier, de presentatie-tier, de businesslogica-tier

en de EIS-tier[6]. De client-tier en de EIS-tier vormen geen onderdeel van de J2EE-specificatie, maar de interfaces naar deze tiers wel. De volgende vier paragrafen geven een korte inleiding over de componenten op de verschillende tiers. Paragraaf 2.4.5 beschrijft de mogelijke J2EE-applicatiescenario's.



Figuur 2-6: Het J2EE-applicatiemodel

2.4.1 Client-tier

Een enterprise-applicatie kan wel of geen webkarakter hebben. Wanneer het om een webapplicatie gaat, bestaat de client uit een browser. De browser ontvangt webpagina's van de server in HTML-, XML- of WML-formaat. Wireless Markup Language is ook te gebruiken met bijvoorbeeld mobiele telefoons. Een HTML-document kan een Java Applet bevatten. Dit is een Java-programma dat beperkte rechten kent en direct in de browser kan draaien, mits er een Java plugin is geïnstalleerd [B3].
Wanneer het geen webapplicatie betreft, dan bestaat de client uit een applicatie. Zo'n applicatie kan zijn geschreven in verschillende programmeertalen, zoals Java, C++, Delphi, VB, etc. Applicaties kunnen direct de businesslogica benaderen, of via de presentatie-tier (zie ook paragraaf 2.4.5).
J2EE-applicaties ondersteunen een zogenaamd *thin client model*. Zo'n client houdt zich alleen bezig met de presentatie naar de gebruiker en het afvangen van de invoer van de gebruiker. De server zorgt voor het benaderen van databases en externe systemen en het uitvoeren van de businesslogica. De client is geen onderdeel van de J2EE-specificatie.

[6] In sommige literatuur worden de presentatielogica-tier en de businesslogica-tier als één tier beschouwd (de middle-tier).

2.4.2 Presentatie-tier

De webcomponenten zijn een "lijmlaag" tussen de client en de businesslogica. Ze bevinden zich in de *webcontainer* op de webserver. Het is mogelijk dat de webserver en de applicatieserver dezelfde machine zijn. Er zijn twee typen web componenten: *Servlets* en *JavaServer Pages*. Servlets zijn Java klassen die enerzijds de gebruikersinvoer verwerken en anderzijds de presentatie voor de gebruiker samenstellen. JavaServer Pages zijn een uitbreiding op het Servlet-model, dat het genereren van HTML- of XML-documenten vereenvoudigt. JavaServer Pages zijn pagina's in tekstformaat, die uiteindelijk worden gecompileerd naar Servlets. Paragraaf 3.2.5 geeft meer informatie over de werking van JavaServer Pages en Servlets.

2.4.3 Businesslogica-tier

Businessobjecten in J2EE heten enterprise beans, of beans in het kort. Ze maken onderdeel uit van de Enterprise JavaBeans specificatie. SUN Microsystems eigen definitie van *Enterprise JavaBeans (EJB)* luidt:

> "*The Enterprise JavaBeans architecture is a component architecture for the development and deployment of component-based distributed business applications. Applications written using the Enterprise JavaBeans architecture are scalable, transactional and multi-user secure. These applications may be written once, and then deployed on any server platform that supports the Enterprise JavaBeans specification.*" [A3]

Echter, Richard Monson-Haefel hanteert in zijn boek een eenvoudigere definitie:

> "*Enterprise JavaBeans is a standard server-side component model for component transaction monitors."* [A3]

Enterprise JavaBeans is volgens de definitie dus een *standaard server-side componentenmodel*[7]. Het specificeert een standaard voor Component Transaction Monitors. Enterprise JavaBeans vormt het hart van het J2EE-framework. Alle businesslogica wordt gestopt in enterprise beans. *Beans* zijn zelfstandige businessobjecten die binnen een EJB-container opereren. De EJB-container vormt de component transaction monitor[8] voor de beans. Deze kunnen zich dan volledig richten op de businesslogica, terwijl de diensten (services) van de container zorgen voor de afhandeling van systeemzaken. Merk op dat Enterprise JavaBeans niet meer is dan een specificatie, een *blueprint.* Aanbieders kunnen hun component transaction monitor implementeren volgens de EJB-specificatie. Paragrafen 3.2.3 en 3.2.4 gaan dieper in op het EJB-model en de aangeboden diensten.

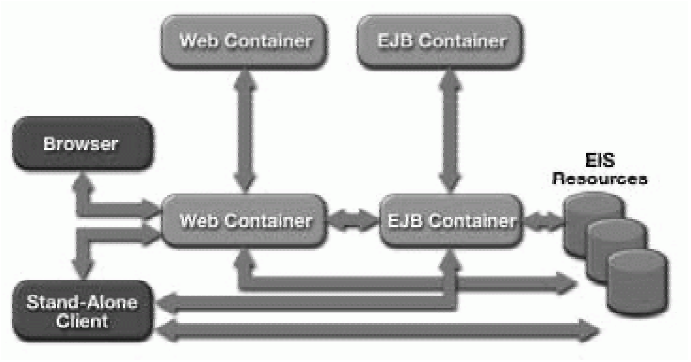
[7] Zie paragraaf 2.3.6 Componentenmodellen
[8] Zie paragraaf 2.3.5 Component Transaction Monitors

2.4.4 EIS-tier

De EIS-tier bevat het informatiesysteem dat de data infrastructuur biedt, die noodzakelijk is voor de bedrijfsprocessen van een organisatie [A7]. Voorbeelden van informatiesystemen zijn relationele databases, enterprise resource planning (ERP) systemen, TP-systemen of "legacy" database-systemen. De EIS-tier is geen onderdeel van de J2EE-specificatie.
De businesslogica laag van een J2EE-applicatie communiceert met een relationele database door gebruik te maken van JDBC (Paragraaf 4.4.1). Communicatie met zogenaamde *legacy systemen* gaat met behulp van *connectors* (paragraaf 4.4.2).

2.4.5 J2EE Applicatiescenario's

De J2EE-specificatie verplicht niet tot het gebruik van een specifieke architectuur. Het programmeermodel is flexibel genoeg om meerdere applicatiescenario's te ondersteunen. De tier-integratie vormt het hart van het J2EE-programmeermodel [A7]. Figuur 2-7 geeft een overzicht van de mogelijke applicatiescenario's voor een J2EE-product. We bespreken een aantal applicatiescenario's aan de hand van de figuur.



Figuur 2-7: J2EE-applicatiescenario's

- ***Web-browser scenario's***
  De interactie tussen een web-browser client en de webcontainer gaat met behulp van HTML-pagina's of Java-applets. Voor de wat eenvoudigere applicaties kan een 3-tier "web-centric" architectuur gebruikt worden. In dit model verzorgt de webcontainer zowel de presentatie- als de businesslogica. In het zogenaamde 'multitier-scenario' zorgt de webcontainer voor de presentatielogica en de EJB-container voor de businesslogica. De webcontainer benadert in het multitier-scenario ook niet direct de data resources, waardoor de backoffice-functionaliteit vrijwel geïsoleerd is van de gebruikersinterface.

- ***Stand-alone client scenario's***
  Een stand-alone clients zijn applicaties geprogrammeerd in een programmeertaal zoals Java. Er zijn drie verschillende scenario's mogelijk met stand-alone clients.
  Clients kunnen direct gebruik maken van de EJB-componenten in de businesslogica tier. Dit is de standaard 3-tier client-server architectuur. Het

is ook mogelijk voor stand-alone clients om te communiceren met de webcontainer. De webcontainer stuurt dan HTML-pagina's of XML-data aan de client In het laatste geval zet de client dit om in de gewenste presentatievorm. Merk op dat de communicatie met de businesslogica laag via de webcontainer verloopt. Het laatste scenario beschrijft het klassieke 2-tier client-server model, waarbij de client direct het informatiesysteem benadert.

- ***Business-to-Business scenario's***
  In een *Business-to-Business (B2B)* scenario vindt er interactie plaats tussen webcontainers of EJB-containers. Dit kan communicatie zijn tussen verschillende samenwerkende bedrijven over het Internet, of binnen een organisatie over het intranet. Voor web-based e-commerce oplossingen is de communicatie tussen webcontainers de meest natuurlijke oplossing. Ze sturen XML-data over een HTTP-protocol en hoeven zo niet nauw geïntegreerd te zijn. Directe communicatie tussen EJB-containers vereist een meer geïntegreerde architectuur en is daarom meer geschikt voor intranet doeleinden. De komst van web services[9] zal de integratie tussen applicaties verder vergemakkelijken.

2.5 Toetsingscriteria

Gedistribueerde applicaties hebben te maken met een aantal criteria of "design goals". Of applicaties deze "design goals" (kunnen) realiseren, hangt af van het ontwerp van de systeemcomponenten [A2]. Om J2EE te kunnen toetsen als ontwikkelplatform voor enterprise-applicaties, stellen we eerst een lijst op met criteria, waaraan enterprise-applicaties moeten kunnen voldoen.

2.5.1 Lijst met criteria

Hieronder volgt de lijst met relevante criteria ("design goals") voor gedistribueerde applicaties. Het J2EE-model moet de mogelijkheid bieden om enterprise-applicaties te creëren, die aan deze criteria voldoen. Deze lijst is gebaseerd op de ontwerpdoelen en eigenschappen van gedistribueerde systemen, die worden genoemd in het boek "Distributed Systems" van G. Coulouris [A2]. Het boek is echter al wat gedateerd en nogal theoretisch. Daarom is uit de praktijk ook het criterium "realisatie & onderhoud" opgenomen. Bovendien is het criterium "consistentie" komen te vervallen. Dit is geen ontwerpdoel, maar een vereiste.

- ***Realisatie & Onderhoud***
  Een relatief nieuw fenomeen in de computerwereld is de realisatietijd. Markten kunnen snel veranderen en daarmee de vraag naar bepaalde gespecialiseerde diensten. Wanneer de oplevering van een softwareproject te lang op zich laat wachten, kan de vraag ernaar afgenomen of zelfs verdwenen zijn. Vaak blijft het niet bij een initiële oplevering. Vernieuwend onderhoud en correctief onderhoud zijn onlosmakelijk

[9] Zie paragraaf 4.3.6

verbonden met applicatieontwikkeling. Onderhoudbaarheid is dan een belangrijk criterium.

- ***Openheid***
  In een gedistribueerd systeem kunnen heterogene componenten van verschillende aanbieders samenwerken, door gebruik te maken van open, standaard interfaces. Met dit soort interfaces is het ook makkelijker om nieuwe componenten aan een systeem toe te voegen. Dit kunnen nieuw ontwikkelde of bestaande componenten zijn.

- ***Performance & Schaalbaarheid***
  Een goede performance van een systeem is belangrijk voor gebruikers. Gedistribueerde systemen hebben te maken met schaalvergrotingen; het aantal gebruikers dat gelijktijdig gebruik maakt van het systeem kan toenemen. De performance mag niet drastisch afnemen wanneer het aantal gelijktijdige gebruikers toeneemt.

- ***Betrouwbaarheid***
  Computersystemen kunnen storingen vertonen. De betrouwbaarheid van een systeem hangt af van de mate van fouttolerantie: de mate waarin storingen in goede banen geleid worden. Systeemgegevens mogen niet corrupt raken als gevolg van een storing. Het zou nog beter zijn als gebruikers niets merken van een opgetreden fout. Het uitvallen van sommige systemen kan catastrofale gevolgen hebben. Voor dit soort systemen is een "stand-by server" gewenst in het geval van een storing.

- ***Beveiliging***
  Alleen gerechtigde gebruikers mogen van het systeem gebruik maken en dan alleen van de functionaliteiten waar ze toe bevoegd zijn. Gegevens in het systeem mogen niet "uitlekken" naar onbevoegden en datacommunicatie mag niet 'vervalst' worden.

2.5.2 J2EE sturingselementen

De mate waarin aan de gestelde criteria kan worden voldaan, hangt af van de inrichting en mogelijkheden van de gedistribueerde architectuur. J2EE biedt een aantal sturingselementen, die helpen bij het invullen van de eigenschappen van een gedistribueerd systeem, zodanig dat het de gestelde criteria tegemoet kan komen. Deze sturingselementen zijn onder te verdelen in vier hoofdcategorieën [A3, A8, B13].
De eerste bevat vooraf geïmplementeerde diensten, zoals een gedistribueerd transactiemechanisme of automatische resource-handling. Met behulp van de door J2EE gespecificeerde *Application Program Interfaces (API)* kunnen deze diensten gebruikt worden. Dit ontlast de ontwikkelaar van zogenaamde "low-level" systeemprogrammering, zodat deze zich kan richten op de applicatielogica.
Ook bestaan er *design patterns* voor het J2EE-model. Design patterns zijn gedefinieerd door SUN als:

> "*A pattern describes a proven solution to a recurring design problem, placing particular emphasis on the context and forces surrounding the problem, and the consequences and impact of the solution*" [B13].

Het zijn bewezen oplossingen voor steeds weer terugkerende problemen, die in dit geval optreden bij het ontwerpen van gedistribueerde applicaties. Appendix B: Core J2EE Patterns bevat een overzicht van de voornaamste J2EE design patterns.
Naast design patterns zijn er *best practices*. Dit zijn adviezen van experts op basis van hun ervaring over wat wel en wat niet te doen op het gebied van gedistribueerde applicatieontwikkeling en J2EE. Waar design patterns meer een richtlijn vormen voor ontwerpkeuzes, zijn dit algemene adviezen die op elk willekeurig onderdeel van het realisatietraject betrekking kunnen hebben.
Tenslotte definieert of ondersteunt J2EE bepaalde interface standaarden als het gaat om bijvoorbeeld databasebenadering of webtoegang. Applicaties communiceren via deze standaard API's met specifieke diensten. Het verschil met de eerder genoemde diensten die door J2EE aangeboden worden, is dat de diensten waarmee gecommuniceerd wordt geen onderdeel uitmaken van het J2EE-model, de API zelf wel.
We hanteren in de volgende hoofdstukken de terminologie die binnen het J2EE-model gebruikt wordt. Dit is overigens de minimale set van diensten die een J2EE-server moet bieden. Leveranciers zijn vrij om extra functionaliteiten te implementeren. Dat valt buiten de strekking van dit document.

2.5.3 Aanpak theorie

De hoofdstukken 3 tot en met 7 bespreken de theorie achter het J2EE-model aan de hand van de genoemde criteria in paragraaf 2.5.1. Elk hoofdstuk behandelt een item van de lijst met toetsingscriteria. De hoofdstukken kennen een gemeenschappelijke indeling. Allereerst volgt een inleiding die uitlegt wat voor concrete eisen er gesteld worden aan dit onderdeel. De rest van het hoofdstuk bespreekt de middelen (sturingselementen) die het J2EE-platform aanbiedt om aan dit criterium te voldoen. Voor zover mogelijk groepeert elk hoofdstuk deze middelen per tier, waarbij de client-side presentatie-tier en de server-side presentatie-tier samengenomen worden.

3 Realisatie & Onderhoud

3.1 Inleiding

Tegenwoordig kennen veel applicaties een dienstverlenende rol, waarin informatievoorziening centraal staat. Mede dankzij de toenemende populariteit van het Internet is de behoefte aan snelle applicatieoplevering ontstaan, omdat nu vele aanbieders dezelfde diensten kunnen leveren. Ook moeten aanbieders snel kunnen inspelen op veranderingen in de markt, om de diensten 'up to date' te houden en zo niet achter te raken op de concurrentie. Bovendien spelen ontwikkelkosten een rol bij de realisatie, zeker in de huidige economische malaise. De tijdswinst kan vaak niet gehaald worden uit de uitbreiding van het ontwikkelteam [C1].
De benodigde software voor de diensten wordt vaak niet door de aanbieders van deze diensten zelf ontwikkeld. Zij nemen de software af van een andere partij. Het verleden heeft uitgewezen dat realisatietijd als enige criterium voor een softwareproduct niet voldoende is. Een slecht onderhoudbare applicatie levert alsnog ontevreden klanten op.
Realisatie en onderhoudbaarheid zijn een maatstaf voor de benodigde tijd en inspanningen met betrekking tot software ontwikkeling, met een gekozen technologie, zoals J2EE. *Realisatie* bevat het initiële ontwikkel- en operationaliseringproces. *Onderhoud* begint na de realisatie en kan verder onderverdeeld worden in correctief onderhoud en vernieuwend onderhoud.

3.1.1 Criteria

De techniek moet ondersteuning bieden bij het snel realiseren van een enterprise-applicatie, die tevens een goede onderhoudbaarheid kent.
De doorlooptijd van een softwareproject, ook wel *time to market* genoemd, geeft aan hoe snel een organisatie in staat is om aan een marktvraag te voldoen. Deze doorlooptijd is cruciaal voor het succes van het project en hangt voornamelijk af van de gebruikte technologie. De technologie mag door haar beperkingen of ongemakken de doorlooptijd niet in gevaar brengen.
Na een initiële oplevering moeten vernieuwingen van informatie en functionaliteit snel, regelmatig en op een betrouwbare manier plaats kunnen vinden. Als deze wijzigingen tot gevolg hebben dat de applicatie opnieuw ontworpen moet worden, dan spreken we van een slecht onderhoudbare applicatie [A7].

3.1.2 Realisatie en onderhoud binnen J2EE

Bijna elk boek, white paper of webartikel over J2EE begint uit te leggen dat J2EE dé standaard oplossing is voor het snel en goedkoop ontwikkelen van robuuste, schaalbare enterprise-applicaties. Realisatie lijkt dan ook centraal te staan binnen het J2EE-model. Uit de theorie zijn een aantal concepten te halen die kunnen helpen bij een snelle realisatie.
Om te beginnen biedt J2EE een *enkelvoudig component-based applicatiemodel*, wat de eenvoud van de applicatie architectuur ten goede

komt. Dit komt omdat alle componenten binnen dit applicatiemodel volgens standaard richtlijnen worden ontworpen, gecombineerd en gedeployed. Dit in tegenstelling tot heterogene, gedivergeerde applicatiemodellen, waar de afstemming van applicatieonderdelen een complexe taak is. Het onderhoud is ook eenvoudiger, omdat componenten onafhankelijk van elkaar kunnen worden vernieuwd en vervangen.
Het J2EE-platform biedt binnen haar applicatiemodel een hoger abstractieniveau doordat J2EE-componenten gebruik kunnen maken van een aantal primaire diensten die een CTM aanbiedt, waardoor ontwikkelaars zich kunnen richten op de applicatielogica.
Het J2EE-componentenmodel vereist dat de componenten in Java geschreven worden. Nu kan dit als een beperking gezien worden, ten opzichte van architecturen waarbij er meerdere talen gebruikt kunnen worden. Echter, Java is een goed doordachte robuuste OO-taal, met hetzelfde abstractieniveau als C++ en een kleinere foutgevoeligheid. Tevens is het platformafhankelijk en staat er een grote 'community' achter. Het biedt hiermee zonder twijfel toegevoegde waarde op het gebied van realisatie en onderhoud.
Door het componentkarakter van het J2EE-platform kan er makkelijker expertiseverdeling plaatsvinden binnen het ontwikkelproces, wat een parallelle uitvoering van taken mogelijk maakt. Zo ontwerpen grafische ontwerpers de JSP-pagina's, definiëren domein experts de businesslogica, stellen Java-programmeurs het totale applicatiegedrag vast, en plaatsen implementatoren de applicatie in een operationele omgeving. De J2EE-specificatie definieert een rollenmodel ter bevordering van de expertiseverdeling.
Het plaatsen of *deployen* van een applicatie behoeft weinig codewijzigingen. Veel van de benodigde code voor een specifieke operationele omgeving kan automatisch worden gegenereerd door tools en bovendien maakt J2EE gebruik van deployment descriptors. Deze descriptors bieden de mogelijkheid om het globale applicatiegedrag en eigenschappen van afzonderlijke componenten met betrekking tot transactiemanagement, beveiliging en benodigde resources, op een declaratieve manier in te stellen.
Doordat J2EE een specificatie is, zitten organisaties niet vast aan een specifieke implementatie. Ze kunnen zelf kiezen welke J2EE-implementatie, tools en standaard componenten ze aanschaffen. Door deze vrijheid kunnen ze aansluiting zoeken bij de reeds aanwezige producten of expertise.

3.2 Realisatie en onderhoud binnen J2EE

3.2.1 Softwarevrijheid

Voor het daadwerkelijk ontwikkelen van een Java 2, Enterprise Edition applicatie kan beginnen, moet er nog wat vooronderzoek plaatsvinden op het gebied van software en vereiste kennis.
Zoals gezegd in hoofdstuk 2 is J2EE een specificatie en geen implementatie. Hierdoor is er niet één maar zijn er meerdere J2EE-server implementaties van verschillende aanbieders op de markt waaruit een organisatie uit kan kiezen. Deze implementaties kunnen verschillen in snelheid, functionaliteit, integratiemogelijkheden met andere producten, en server configuraties [A7].
Zonder een goede ontwikkelomgeving lijkt het maken van een J2EE-applicatie onbegonnen werk. Het ontwerpen en aanpassen van JSP-pagina's en enterprise

beans is vele malen makkelijker met de nodige grafische ondersteuning. Debugmogelijkheden zijn ook onmisbaar in zo'n omgeving, liefst integraal. Deployment tools zijn geen vereiste, maar maken het leven van een J2EE-ontwikkelteam wel makkelijker.
Doordat J2EE een componentenarchitectuur kent, kan bepaalde functionaliteit geïsoleerd, gestandaardiseerd, verpakt en hergebruikt worden. Dit leidt tot een groeimodel voor kant-en-klare 3rd-party componenten. Denk bijvoorbeeld aan rekenmodules, GUI templates of andere veel voorkomende functionaliteit.
Naast software is er ook de nodige voorkennis vereist. Alleen al vanwege de omvang van diensten in J2EE (zie Appendix A: J2EE Required API's) lijkt het een onmogelijke taak om aan de ontwikkeling te beginnen wanneer de ontwikkelaars volledig onbekend zijn met J2EE-technieken, de programmeertaal Java en gedistribueerde applicatieontwikkeling in het algemeen. Verder zijn experts van mening dat het waarschijnlijk niet haalbaar is om een goed onderhoudbare en schaalbare J2EE-applicatie te ontwikkelen zonder het gebruik van J2EE design patterns [A8].

3.2.2 Java-programmeertaal

Het J2EE-model is gebaseerd rondom Java. In tegenstelling tot het .NET framework van Microsoft waar een meerdere programmeertalen gebruikt kunnen worden, moeten J2EE-componenten in Java worden geschreven. *J2EE-componenten* zijn applicatie clients, applets, connectors, Servlet- en JSP-componenten, en EJB-componenten. De hype die halverwege de jaren negentig rondom Java ontstond, is slechts deels terecht. Java is een goede, maar geen geweldige programmeertaal [A5]. Hieronder staat een aantal eigenschappen van Java, dat het tot een goede programmeertaal maakt.
SUN Microsystems bracht begin 1996 Java op de markt. Java borduurt voort op de syntax van C++, een gerespecteerde programmeertaal die zijn strepen al verdiend had. Echter, Java is meer dan een "C++ dialect". De ontwerpers hadden eens goed naar de tekortkomingen van C++ gekeken en met name de foutgevoeligheid ervan. Ze voegden wat kenmerken aan de Java-programmeertaal toe die het maken van meest voorkomende bugs onmogelijk maakt. Zo kent Java een automatische garbage collectie, die het handmatig beheren van geheugen overbodig maakt. Ook rust de verantwoordelijkheid van pointerverwijzingen niet langer op de schouders van de ontwikkelaar. Alles in Java is een object, op de basistypen zoals getallen na. Tenslotte is de complexiteit van multiple inheritance vervangen door het begrip *interfaces*, dat afstamt van "Objective C" en is Java een *strongly typed* programmeertaal. Dit betekent dat de compiler veel gemaakte fouten afvangt, die in andere talen pas tijdens run-time naar voren komen[10].
De verbeteringen in de laatste jaren hebben te maken met enorme aanpassingen in de Java-bibliotheken. Foundation classes zijn redelijk nieuw en volgen daarom netjes de meeste "moderne" design patterns. De ondersteuning voor bijvoorbeeld netwerkprogrammering, serialisatie, multi-threading en run-time type informatie is uitgebreid en makkelijk te gebruiken. Het heeft bijdragen aan de elegantie van Java.

[10] Bijvoorbeeld het verwarren van een assignment met een test op gelijkheid.

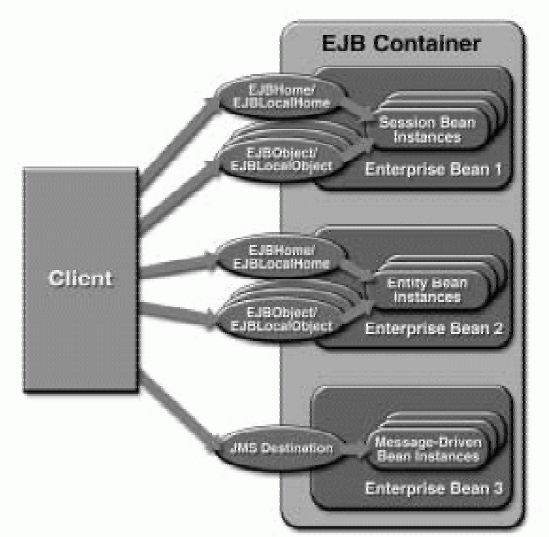
Doordat Java een geïnterpreteerde taal is, kunnen bepaalde beveiligingen worden toegepast. Het is niet mogelijk voor een Java-programma om geheugen buiten haar procesruimte te manipuleren, of om de runtime stack te laten overlopen. Internetapplicaties kunnen niet zonder toestemming van de gebruiker lokale bestanden benaderen. De prijs die betaald moet worden voor deze interpretatie is snelheid. Een programma dat wordt gecompileerd naar direct uitvoerbare code draait sneller dan wanneer het nog een interpretatieslag krijgt. Echter, met de komst van *Just In Time (JIT) compilers* is deze achterstand grotendeels weggewerkt. Deze compilers compileren na het interpreteren delen van een programma naar direct uitvoerbare code en cachen dit voor hergebruik. Een ander voordeel van interpretatie is de portabiliteit. Zo is Java platformonafhankelijk[11] en bovendien zijn de specificaties van data typen op elk platform gelijk.
Een laatste pluspunt van de Java-programmeertaal is de enorme community die erachter zit. Door haar omvang is veel informatie te verkrijgen via boeken, periodieken en op het Internet. Dit trekt de aandacht van bedrijven waardoor toekomstige ontwikkelingen sneller gaan, er meer kant-en-klare componenten verschijnen en de kans klein is dat Java opeens ophoudt te bestaan, door gebrek aan draagkracht.
Samengevat is Java een goed doordachte taal is die hetzelfde abstractieniveau kent als C++, wat de ontwikkelmogelijkheden ten goede komt. De foutgevoeligheid van code echter veel kleiner geworden in vergelijking met C++. Door de grote community staan de ontwikkelingen binnen Java niet stil en door de relatief nieuwe foundation classes wordt een ontwikkelaar van een hoop standaardfunctionaliteit voorzien, zonder zich in oude, achterhaalde code structuren te moeten begeven.

3.2.3 Component Model Architectuur

De componenten op de applicatieserver heten *enterprise beans*, of eenvoudigweg *beans*. Beans zijn objectgeoriënteerde componenten die business methoden bevatten. Ze vormen de businessobjecten van een enterprise-applicatie. Er zijn drie soorten beans: entity beans, session beans en message-driven beans. *Entity beans* modelleren “objecten” uit de werkelijkheid. Hun toestand wordt permanent in een database opgeslagen. *Session beans* handelen processen en taken af. Voor deze taken kunnen ze gebruik maken van de data in entity beans. Session beans komen in twee soorten. *Stateless* session beans houden geen toestand van variabelen bij tijdens de interactie tussen een client en de bean. Dit houdt in dat deze beans geen gebruik maken van klassenvariabelen en dat elke methode aanroep wordt uitgevoerd vanuit dezelfde begintoestand. *Stateful* session beans houden wel een toestand bij door middel van klassenvariabelen. Het verschil met entity beans is dat deze informatie niet wordt opgeslagen in een database en dus blijft bestaan zolang het object bestaat. Tenslotte zijn in EJB 2.0 M*essage-driven Beans (MDB)* toegevoegd. Paragraaf 4.3.5 beschrijft de werking van message-driven beans en de Java Message Service.
De omgeving waarbinnen de beans opereren heet de *container*. Dit is een component transaction monitor op de EJB-server dat optreedt als

[11] Zie paragraaf 4.3.1 Platformonafhankelijkheid

tussenpersoon voor clients, beans en de EJB-server. Clients communiceren nooit direct met bean instanties. De container maakt een EJB-object aan, dat de local en remote interfaces van de bean bevat. Figuur 3-1 geeft een schematisch overzicht van de client-view met betrekking tot verschillende soorten enterprise beans. Het componentenmodel legt de interacties tussen de beans en de container vast. Het beschrijft de interfaces waarmee ze communiceren, dat gezien kan worden als een *bean-container contract*. Op die manier kan de container de beans goed beheren en de primaire diensten uitvoeren op alle beans in de container.



Figuur 3-1: Enterprise JavaBeans

De interfaces tussen de EJB-container en de EJB-server zijn niet concreet gedefinieerd. Het is onduidelijk wat precies de verantwoordelijkheden zijn van de server en van de container. Er ontbreekt zogezegd een eenduidig *container-server contract.* Vandaag de dag leveren de meeste aanbieders zowel een container als een server en bepalen zo zelf de onderlinge communicatie. Als er nu een specificatie komt van de interface, kunnen de server en de container van verschillende partijen komen. Een mogelijk nadeel hiervan is dat een standaard interface ten koste gaat van de performance. Een vuistregel is: hoe nauwer de integratie, hoe beter de performance; hoe hoger de abstractie, hoe groter de flexibiliteit [A3].
Het is belangrijk te vermelden dat Enteprise JavaBeans een *standaard* server-side componentenmodel is, zoals beschreven in paragraaf 2.3.6. De EJB-specificatie kwam in 1997 op de markt. Er was behoefte aan een standaard server-side componentenmodel, aangezien tot dan toe de CTM-aanbieders allemaal hun eigen componentenmodel gebruikten. Componenten voor zo'n CTM werkten niet op de CTM van een andere aanbieder. CTM's die de EJB-specificatie implementeren, zijn *EJB-compliant*. Dit wil zeggen dat businessobjecten zonder code-aanpassingen gebruikt kunnen worden op CTM's van andere aanbieders die het EJB-model implementeren.

3.2.4 Standaard Componentdiensten

Het maken van een gedistribueerde multitier-applicatie is complex. Daar valt niets op af te dingen. Nu kan een applicatieontwikkelaar zelf alle systeemzaken implementeren, zoals transactiemanagement, multi-threading, resourcemanagement, en dergelijke. Maar het J2EE-platform biedt een hoop van deze diensten standaard aan. Zo kan een ontwikkelaar zich richten op de businesslogica en gebruikt hij de meegeleverde diensten als een *black box* [12].
Dit heeft ook gevolgen voor het realisatietraject van een applicatie, vandaar dat we vast een lijst presenteren met de primaire diensten die een "EJB compliant CTM" moet aanbieden. De inhoudelijke functie van een dienst staat los van het realisatietraject en wordt daarom besproken in het bijpassende hoofdstuk.
Deze diensten zijn op zich geen nieuwe concepten. De Object Management Group (OMG) definieerde al eens interfaces voor deze diensten voor CORBA. Daarbij kwam de coördinatie en implementatie van deze diensten echter op de schouders van de ontwikkelaar terecht. EJB-servers beheren automatisch alle primaire diensten en de ontwikkelaars stellen het gedrag van deze diensten met behulp van parameters in. Zie ook de paragraaf 'Applicatie Packaging en Deployment'.

Service	Omschrijving	Behandeld in
Object distributie	Synchrone communicatie tussen (remote) componenten.	Hoofdstuk 4
Asynchroon Messaging	Asynchrone communicatie tussen (remote) componenten	Hoofdstuk 4
Naming	Lokaliseren van objecten op basis van een logische naam.	Hoofdstuk 4
Resourcemanagement	Het beheer van resources die gebruikt worden door gedistribueerde objecten.	Hoofdstuk 5
Concurrency	Meerdere clients toegang geven tot een object.	Hoofdstuk 5, 6
Transactiemanagement	Het beheren van transacties op businessobjecten in een gedistribueerde omgeving.	Hoofdstuk 6
Persistentie	Synchronisatie van de status van een object met de data in een database	Hoofdstuk 6
Beveiliging	Een uniform beveiligingsmodel voor het toepassen van beveiligingsmaatregelen	Hoofdstuk 7

Tabel 1: Standaard Component Diensten

Het totaal aantal diensten dat een "EJB compliant CTM" levert staat overigens niet vast. EJB definieert alleen de minimale set van diensten waaraan een component transaction monitor moet voldoen. Een CTM-aanbieder kan extra functionaliteit toevoegen, om zo concurrentievoordeel te behalen.

[12] zie ook Figuur 2-4

3.2.5 JavaServer Pages en Servlets

JavaServer Pages (JSP) is een technologie voor het ontwikkelen van webpagina's met een dynamische inhoud [A9]. Dit in tegenstelling tot een HTML-pagina, die altijd een statische inhoud kent. Een JSP-pagina heeft zowel standaard opmaak elementen, zoals HTML-tags, als speciale JSP-elementen die het mogelijk maken voor de server om dynamische inhoud toe te voegen aan de pagina.
Het genereren van dynamische webinhoud is geen nieuw concept. In het begin van het webtijdperk kon dynamische webinhoud al gegenereerd worden met behulp van het *Common Gateway Interface (CGI)* model. Maar CGI is geen efficiënte oplossing, aangezien voor elke aanroep een nieuw proces gestart moet worden om het script uit te voeren. Verschillende alternatieven kwamen op de markt, zoals FastCGI, NSAPI, ISAPI en Java Servlets. Deze alternatieven verbeterden de performance en schaalbaarheid maar hadden allemaal dezelfde tekortkoming: het genereren van dynamische webpagina's door de HTML-code direct in de programmacode op te nemen. De scheiding van presentatie en businesslogica ontbreekt, waardoor het maken van dynamische webpagina's de verantwoordelijkheid is van de programmeur. Een nieuwe look&feel van de pagina's resulteert in een hoop integratiewerk voor de ontwikkelaars [C2].
JavaServer Pages pakt het probleem andersom aan en laat de programmeurs hun Java-code toevoegen aan de HTML-pagina's. Ook staat JSP het gebruik van scriptelementen toe in een HTML-pagina, maar dit is haast niet meer nodig[13] met de komst van de JSP Standaard Tag Library (JSTL). JSTL is een uitbreiding op de JSP-specificatie die het mogelijk maakt om specifieke taken uit te voeren, zoals het benaderen van een database of het sturen van een e-mail. Door deze scheiding van opmaak en logica kan er makkelijker specialisatie plaatsvinden, waarbij programmeurs de applicatielogica inbouwen en grafische ontwerpers verantwoordelijk zijn voor de opmaak van pagina's. Doordat Servlets en JavaServer Pages gebruik maken van de Java-programmeertaal, krijgen ze automatisch een aantal Java-realisatievoordelen cadeau. Ze zijn strongly typed en hebben een robuuste error-handling.

3.2.6 Applicatie Packaging en Deployment

De *deployment* van een applicatie is het installeren en configureren van een applicatie in een operationele omgeving. Het is voor het realisatiesucces belangrijk dat applicaties gemakkelijk gedeployed kunnen worden. Ook de mogelijkheid om dezelfde applicatie in een andere server-omgeving operationeel te maken, zonder al te veel programmeerwerk, is een belangrijke eis. Voor het onderhoud is herdeployment een belangrijke factor. Tijdens het ontwikkelen is er behoefte aan het snel deployen en undeployen van applicatieonderdelen.
Het J2EE-model maakt het mogelijk voor ontwikkelaars om componenten samen te voegen tot een applicatie. Het proces van het assembleren van

[13] Het gebruik van scripting wordt afgeraden, vanwege de slechte onderhoudbaarheid en de beperkte debugmogelijkheden.

componenten in modules en modules in enterprise-applicaties, wordt *packaging* genoemd. Deze woorden verdienen wat extra uitleg. Een *J2EE-component* is een onafhankelijke functionele software-eenheid, die voldoet aan de door de specificatie gedefinieerde interfaces. Het kan een klasse zijn, maar vaker is het een verzameling van klassen, interfaces en resources. Er bestaan vijf soorten J2EE-componenten: enterprise beans, Servlets en JSP-pagina's, applets, applicatie clients en connectors. Een of meer J2EE-componenten kunnen worden samengevoegd tot een *module.* Een module is de kleinst mogelijke deployment eenheid voor een component. Het kan direct gedeployed worden in een J2EE-container, of meerdere modules kunnen worden gecombineerd tot een J2EE-applicatie. Tabel 2 geeft een overzicht van de J2EE-modules [A7].

Type	Extensie	Inhoud
EJB module	JAR (*Java Archive*)	Enterprise beans, verwante klassen
Web module	WAR (*Web Archive*)	Web-tier componenten, resources
Applicatie client module	JAR (*Java Archive*)	Applicatie client klassen
Resource adapter module	RAR (*Resource Archive*)	Connectors, resource adapters, support libraries, resources

Tabel 2: J2EE-platform modules

Modules en applicaties worden verpakt en gedeployed als *deployment units.* Dit zijn gecomprimeerde archiefbestanden, compatible met het ZIP-formaat, met een gespecificeerde interne structuur. De bestandsextensie hangt af van het type module. Naast de componenten en de resources bevat elke deployment unit een deployment descriptor; een XML-bestand dat de expliciete afhankelijkheden tussen elk component en haar omgeving specificeert. Dit kan het beste toegelicht worden aan de hand van een voorbeeld. De volgende paragraaf beschrijft het gebruik van deployment descriptors voor enterprise beans.
Veel informatie over hoe enterprise beans bestuurd moeten worden tijdens het draaien van de applicatie, staat niet in de code van de beans zelf. Het gaat hierbij voornamelijk om de primaire diensten, besproken in de paragraaf 3.2.4. De EJB-container voert deze diensten uit, maar heeft informatie nodig over hoe deze diensten toegepast moeten worden op een bean. Deze informatie staat in een Deployment Descriptor. Een *Deployment Descriptor (DD)* is een XML-bestand dat te vergelijken is met een "property page". Er staan twee soorten informatie in een Deployment Descriptor: structurele informatie en assembly informatie [A3, A7, B5]:

- ***Structurele informatie***
  Structurele informatie beschrijft de metadata van de componenten in een deployment unit, hun onderlinge relaties en hun externe afhankelijkheden. Zo staan hierin voor enterprise beans de home en remote interfaces

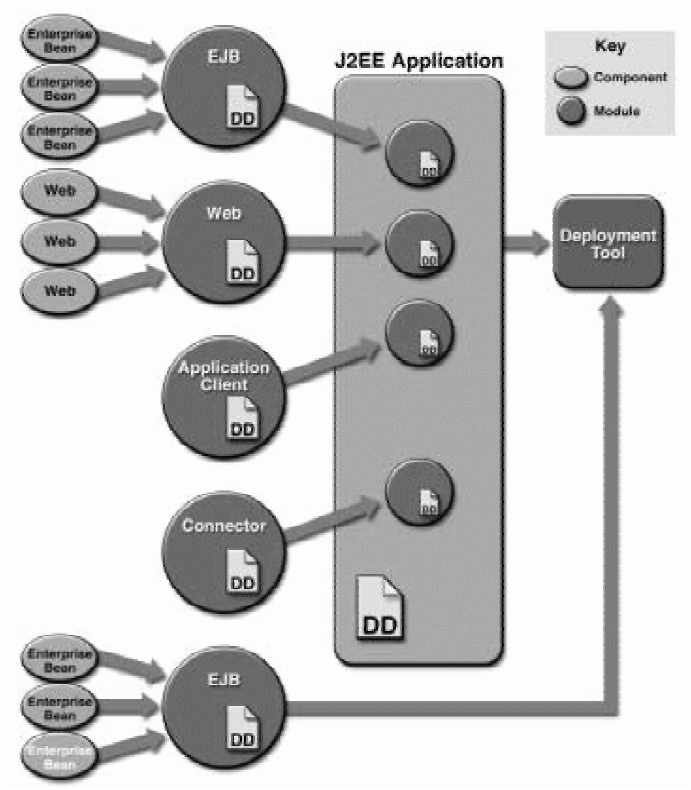
beschreven, de naam en het type bean, de vereiste resources, informatie over persistentie- en statusmanagement en query-informatie.
Structurele informatie is vereist voor elk component en bevat hard gecodeerde eigenschappen, die niet wijzigbaar zijn tijdens de deploymentfase. Dit zou de functie van een enterprise bean kunnen verstoren. De component-ontwikkelaar is verantwoordelijk voor de structurele informatie in de deployment descriptors.

- ***Assembly-informatie***
  De assembly-informatie beschrijft de inhoud van een deployment unit samengevoegd moeten worden met andere deployment units, om zo een nieuwe unit te vormen. Hierin staan beveiligingsaspecten, zoals de aanbevolen beveiligingsrollen en de methode permissies die de link leggen tussen een security rol en de methoden van een bean. Ook kunnen hier transactieattributen worden ingesteld.
  Assembly-informatie is optioneel en wijzigingen verstoren de functie van een component niet, maar het kan wel gebruikt worden om het applicatiegedrag te beïnvloeden. Component-ontwikkelaars kunnen initiële waarden toewijzen aan bepaalde onderdelen. Assemblers en deployers veranderen of definiëren de uiteindelijke assembly-informatie, om een component te configureren voor zijn rol in een applicatie. Tevens verzorgen zij de assembly-informatie voor de applicatie als geheel.
  Door het samenvoegen kan het voorkomen dat er *name clashes* ontstaan in de structurele informatie. In dat geval is het de taak van de assembler om dit op te lossen.

Veel programmeeromgevingen bieden de mogelijkheid om deployment descriptors grafisch in te stellen. Ze maken dan automatisch het XML bestand aan op basis van de Document Type Definition[14] van Enterprise JavaBeans zodat een ontwikkelaar hier geen omkijken naar heeft.
De uiteindelijke J2EE-applicatie wordt verpakt als een portable deployment unit, een *Enterprise Archive (EAR)* bestand. Een EAR-bestand is net zoals andere deployment units een ZIP-compatible bestand met in dit geval een ".ear" extensie. Een enterprise archive bestand bevat een of meer J2EE-modules en een J2EE-applicatie deployment descriptor. Figuur 3-2 laat de verschillende typen J2EE-modules zien en hoe ze worden gedeployed. Elke J2EE-module kan onafhankelijk gedeployed worden. Onderaan in de figuur is te zien hoe een EJB module onafhankelijk wordt gedeployed.
De deploymentfase bestaat uit twee hoofdstaken: installatie en configuratie. *Installatie* betekent in deze context het verplaatsen van de media op de server, het genereren van de additionele containerspecifieke klassen en interfaces die het nodig heeft om de componenten tijdens run-time te managen, en het installeren van de componenten zelf op de J2EE-server. De *configuratie* omvat het oplossen van de externe afhankelijkheden en het toepassen van de assembly-instructies. Bijvoorbeeld het configureren van de database die de applicatie gebruikt voor persistentie en het vertalen van de abstracte beveiligingsrollen naar concrete rollen in de operationele omgeving.

[14] Een DTD is een document dat de structuur beschrijft van een XML-document

Deployers kunnen tools gebruiken van de J2EE-productleverancier om J2EE-modules en applicaties te installeren en te configureren. Tot nu toe stelde de J2EE-specificatie wel een aantal eisen aan deployment tools, maar definieerde niet de interface tussen de deployment tools en de containers. Met andere woorden: deployment tools zijn aanbieder specifiek. Dit heeft als gevolg dat het deploymentproces niet gestandaardiseerd is en ook niet portable tussen producten. De J2EE 1.4 specificatie zal deze interface wel definiëren en daarmee het deploymentproces standaardiseren.



Figuur 3-2: J2EE deployment units

3.2.7 Rolverdeling

Sun Microsystems definieert zes verschillende rollen binnen het J2EE-platform. Deze rollen zijn in het algemeen bedoeld om te helpen bij het identificeren van de taken en zo bij te dragen tot specialisatie. Elke J2EE-rol kan vervuld worden door een aparte persoon of groep, maar een persoon kan ook meerdere rollen vervullen. Op deze manier kunnen verschillende personen of teams onafhankelijk van elkaar verschillende taken uitvoeren van het ontwikkelproces. Oftewel, er kan parallelle taakverwerking plaatsvinden, een belangrijk aspect voor de realisatietijd [A7, B3, B5].

- ***J2EE-productleverancier***
  De leverancier van een J2EE-product is vaak een aanbieder van besturingssystemen, database systemen, applicatieservers of webservers. Deze implementeert een J2EE-product door het leveren van de component containers, de J2EE-platform API's en andere onderdelen van de J2EE-

specificatie. Een J2EE-product is vrij om interfaces die niet door de J2EE-specificatie zijn gespecificeerd, op een specifieke manier te implementeren. Deze rol van J2EE-product leverancier kan verder onderverdeeld worden in subrollen. De EJB-specificatie maakt bijvoorbeeld onderscheid tussen de EJB-container leverancier en de EJB-server leverancier.

- ***Tool-leverancier***
  Een tool-leverancier levert de tools die gebruikt kunnen worden voor het ontwikkelen, verpakken en deployen van applicatiecomponenten.

- ***Applicatiecomponent-ontwikkelaar***
  Component-ontwikkelaars zijn programmeurs die de bouwstenen van een J2EE-applicatie maken. Ze hebben over het algemeen kennis over het ontwikkelen van herbruikbare componenten en de benodigde functionele domeinkennis. Ontwikkelaars hoeven niets te weten van de operationele omgeving waar hun componenten zullen draaien.
  Ook de rol van componentontwikkelaar kan verder onderverdeeld worden in bijvoorbeeld HTML-pagina ontwerpers, document programmeurs en enterprise bean ontwikkelaars.

- ***Assembler***
  De assemblers voegen alle componenten van de ontwikkelaars samen in een gedistribueerde applicatie. Ze hebben expertise in het oplossen van problemen voor een specifiek domein, bijvoorbeeld in de financiële wereld. Ze hoeven niet bekend te zijn met de source-code van de componenten. Vaak kunnen ze dit als een black box beschouwen en gebruiken ze de declaratieve descriptors van de componenten om erachter te komen hoe ze deze componenten kunnen gebruiken in een applicatie. Assemblers zijn naast het samenvoegen van componenten ook verantwoordelijk voor het definiëren van assembleerinstructies. Deze instructies beschrijven de externe afhankelijkheden van een applicatie, die de deployer moet oplossen.

- ***Deployer***
  Een deployer is een expert voor een specifieke operationele omgeving en installeert J2EE-componenten en applicaties op een J2EE-server. Zie paragraaf 3.2.6 voor meer informatie over de deployment van een J2EE-applicatie. Hij gebruikt de eigenschappen van de applicatie om zo het beveiliging- en transactiebeleid in te stellen. Ook zorgt hij voor de integratie met eventuele bestaande Enterprise Information Systemen (EIS).

- ***Systeembeheerder***
  Een systeembeheerder is verantwoordelijk voor de installatie en configuratie van de netwerkinfrastructuur, en de J2EE-server. Tevens houdt hij toezicht op de draaiende applicatie en onderneemt actie wanneer de applicatie niet naar behoren functioneert.

4 Openheid

4.1 Inleiding

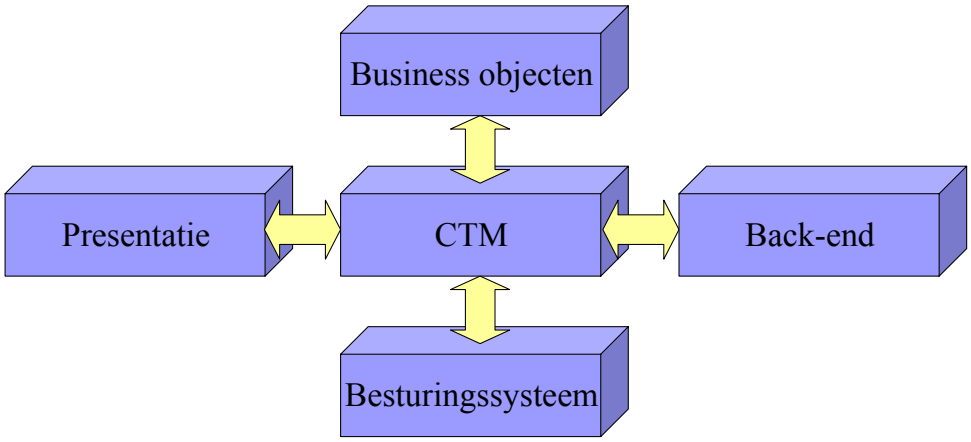
In theorie is *openheid* een eigenschap die aangeeft in welke mate een systeem uitgebreid kan worden op met betrekking tot hardware en software. Deze openheid komt tot stand door het gebruik van gestandaardiseerde, gepubliceerde externe interfaces. Open gedistribueerde systemen maken gebruik van een uniforme communicatie tussen processen en kunnen zo opgebouwd worden uit heterogene hard- en software componenten van verschillende aanbieders. Dit aspect heet *interoperability*. Het gebruik van standaard protocollen en dataformaten kan een open systeem onafhankelijk maken van een specifieke aanbieder. De openheid van zo'n systeem wordt verder bepaald door het *extensibility* aspect. Het zegt iets over de mate waarin nieuwe diensten of componenten kunnen worden toegevoegd aan het systeem, waarbij ze samenwerken met de bestaande diensten en deze niet kopiëren [A2, B7].
In de praktijk kan dit als volgt vertaald worden. De uitbreidbaarheid, of extensibility kan een hardware- of functionaliteituitbreiding zijn. Zoals het ondersteunen van een nieuw type client. De integratie met verschillende platformen of bestaande systemen zijn voorbeelden van interoperability mogelijkheden. Veel organisaties hebben bijvoorbeeld de afgelopen jaren hun data verzameld in een zogenaamd enterprise informatiesysteem. Er zit veel investering in zo'n applicatie. Enterprise-applicaties moeten deze systemen zoveel mogelijk hergebruiken in plaats van ze te herschrijven [A7, B7].

4.1.1 Criteria

We streven naar een *open gedistribueerd systeem*, waarbinnen componenten van verschillende aanbieders samen kunnen werken door het gebruik van standaard interfaces. Componenten kunnen in deze context kant-en-klare objecten zijn, maar ook bestaande (sub)systemen of gehele applicaties. Het systeem moet niet afhankelijk zijn van een specifieke aanbieder of van een specifiek platform.
Het moet mogelijk zijn nieuwe componenten aan het systeem toe te voegen en deze te laten samenwerken met bestaande componenten. Aan de andere kan moet er integratie plaats kunnen vinden met bestaande componenten, zodat deze niet opnieuw ontwikkeld hoeven worden.

4.1.2 Openheid binnen J2EE

Wanneer we kijken naar de openheid binnen het J2EE-model, nemen we als uitgangspunt Figuur 4-1. In hoeverre zijn de interfaces, aangegeven met de pijlen, open standaarden? J2EE biedt een hoop standaard interfaces aan voor communicatie tussen componenten. Zo zou elk van de van de componenten in Figuur 4-1 vervangen moeten kunnen worden door een soortgelijke component van een andere aanbieder.



Figuur 4-1: Schematisch overzicht J2EE

Allereerst de presentatielaag. Wanneer er standaard protocollen gebruikt worden voor deze interface, kunnen vele verschillende typen clients verbinding maken met de applicatieserver. J2EE ondersteunt verschillende soorten clients, variërend van web-browser clients, tot palmtops. Het is ook mogelijk om meerdere soorten clients te bedienen vanuit een enkele applicatie. Met behulp van design patterns en zogenaamde applicatie-frameworks, blijft de applicatie in zo'n situatie goed onderhoudbaar.
Een belangrijk aspect van de EJB-container (de CTM) is dat deze zelf vervangbaar[15] is en dankzij de Java-programmeertaal niet gebonden is aan een specifiek platform. De businesslogica-tier heeft een belangrijke rol wanneer het gaat om integratie met bestaande systemen. Er is ondersteuning voor synchrone en asynchrone communicatie met andere applicaties. Synchrone communicatie gaat met behulp van het RMI netwerkprotocol. Asynchrone communicatie kan door gebruik te maken van de Java Message Service. Dit is een standaard API waarmee asynchrone messages verstuurd en ontvangen kunnen worden, door middel van een messaging-systeem. Tenslotte tippen we nog even aan het onderwerp "web services". Als we de hype moeten geloven, is dit de manier van communiceren tussen heterogene applicaties in de toekomst. Met name door de werkelijke platformonafhankelijkheid en de adoptie als open standaard hebben web services een enorme populariteit gekregen. Zo is ondersteuning van web services de voornaamste reden voor de ontwikkeling van de J2EE 1.4 specificatie.
Dan de integratie met de Backend-tier. De Backend-tier bestaat uit een of meer *Enterprise Information Systems (EIS)* en hier is met name de integratie met bestaande systemen relevant. Voorbeelden van Enterprise Information Systems zijn relationele databases, enterprise resource planning (ERP) systemen, mainframe transaction processing systemen en legacy database systemen. Bij een open systeem moet het mogelijk zijn om op een standaard manier met bestaande heterogene informatiesystemen te integreren [A7]. J2EE biedt een standaard API voor het benaderen van databases en een standaard API voor de integratie met legacy applicaties.

15

Appendix C: J2EE Licentiehouders biedt een overzicht van de gecertificeerde J2EE-aanbieders en hun producten.

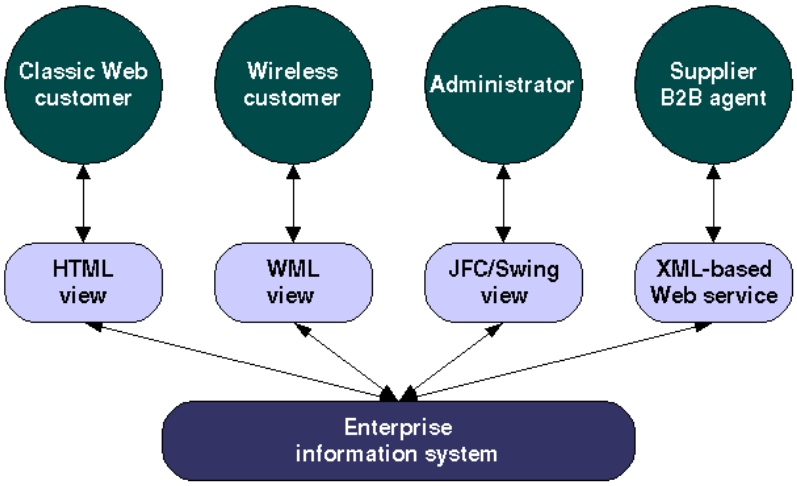
4.2 Presentatie-tier

4.2.1 JavaServer Pages en Servlets

Servlets en JavaServer Pages zijn, net zoals vele andere onderdelen binnen het J2EE-model, een specificatie. Zo kunnen aanbieders concurreren met verschillende implementaties. Dit komt de performance en kwaliteit van Java ServerPages ten goede. Ook zal dankzij de investeringen van deze bedrijven de ondersteuning voor JSP niet zomaar verdwijnen. Andere hedendaagse oplossingen zijn ASP, PHP, ColdFusion en Java Servlet template engines [A9]. Ze hebben elk hun voor- en nadelen, maar zijn in tegenstelling tot JSP een product en geen specificatie. Alle grote webservers ondersteunen de Servlet-specificatie, wat de technologie niet verbindt met een specifieke server-aanbieder. Bovendien draaien Servlets binnen een Java virtuele machine[16], wat ze ook platformonafhankelijk maakt. Zie paragraaf 4.3.1 voor meer informatie over de platformonafhankelijkheid van Java.

4.2.2 Typen clients

Een J2EE-applicatie kan vele typen clients ondersteunen. Clients kunnen draaien op laptops, desktops, palmtops of mobiele telefoons en maken verbinding met de server via het intranet, het Internet, een (draadloos) netwerk of een combinatie hiervan. De clients verschillen van *thin clients* in een web browser die grotendeels server afhankelijk zijn, tot *thick clients* die bijna volledig autonoom draaien [A7].



Figuur 4-2: Verschillende views van een applicatie

Met verschillende clients komen verschillende protocollen en presentatievormen. Zoals Figuur 4-2 laat zien kan een enterprise-applicatie een HTML-presentatie hebben voor webgebruikers, en WML-presentatie voor draadloze netwerkgebruikers, een “Java Swing”-presentatie voor bijvoorbeeld

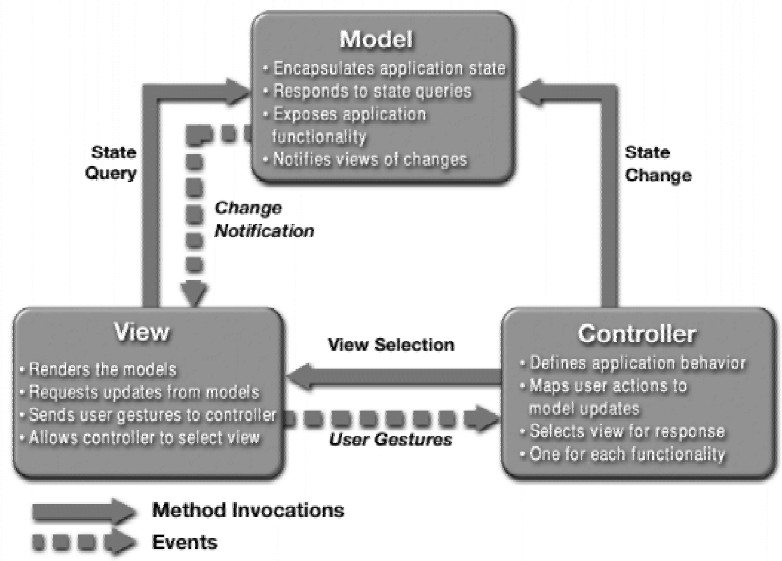
[16] Zie 4.3.1 voor meer uitleg over de Java virtuele machine.

systeembeheerders en een XML-interface voor business-to-business doeleinden. De protocollen kunnen ook nog afwijken. Vaak is de enige vorm van communicatie over het Internet met behulp van het stateless HTTP(S) protocol. Dit komt omdat veel servers afgeschermd worden door firewalls die alleen het HTTP-protocol doorlaten. Voor andere netwerken kan bijvoorbeeld een RMI-protocol gebruikt worden. Dit bespreken we verder in paragraaf 4.3.3. J2EE ondersteunt deze protocollen en biedt hiermee een flexibele presentatievorm voor haar applicaties.

4.2.3 Model View Controller design pattern

De technieken, interfaces en protocollen mogen nog zo veelzijdig zijn, maar zonder een goed ontwerp kan veel van de toegevoegde waarde tenietgedaan worden. We bespraken in paragraaf 3.2.5 reeds de ontwikkelingen van de technieken voor het maken van webpagina's met een dynamische inhoud. Dit begon met het CGI-model tot en met de JavaServer Pages. Het gebruik van JavaServer Pages biedt echter geen garantie voor een goed onderhoudbare applicatie. Omvangrijkere applicaties resulteren vaak in cryptische JSP-pagina's vol met een mix van Java-code en HTML-elementen. Bovendien staan er kopieën van de Java-code op verschillende pagina's [C2].
Een ander probleem komt voort uit de ondersteuning van meerdere clients vanuit een *enkelvoudige* applicatie. Dit komt omdat een enterprise-applicatie vandaag de dag steeds vaker verschillende soorten gebruikers moet kunnen ondersteunen met afwijkende interfaces [B13]. Er ontstaan dan vaak kopieën van applicatiecode voor verschillende clients, hetgeen het testen en het onderhoud moeilijker maakt. En uiteindelijk zullen ze uitgroeien tot aparte applicaties. Hoewel J2EE de benodigde protocollen ondersteunt, zoals uitgelegd in de vorige paragraaf, is daarmee het probleem nog niet verholpen.
De kern van het probleem is dat in beide gevallen de applicatiecode niet goed gescheiden is van de presentatie. Dit betekent dat wanneer de presentatie verandert, de applicatiecode aangepast moet worden en vice versa. Samengevat kunnen we zeggen dat er een aantal uitdagingen en eisen zijn met betrekking tot de presentatielaag van een applicatie: [B13, A9]

- Verschillende typen views moeten dezelfde applicatiedata kunnen weergeven.
- Interacties met verschillende typen views moeten dezelfde applicatiedata kunnen aanpassen.
- Het ondersteunen van verschillende typen views mag geen effect hebben op de businesslogica componenten van de applicatie.
- Een verandering van de paginavolgorde in een view mag geen effect hebben op de businesslogica-componenten van de applicatie, of op de pagina-inhoud zelf.



Figuur 4-3: Model View Controller design pattern

Het model view controller concept biedt een oplossing voor deze eisen. Het scheidt de businesslogica van de presentatie en voegt een extra 'sturingslaag' toe Allereerst een uitleg van de drie benamingen [A7]:

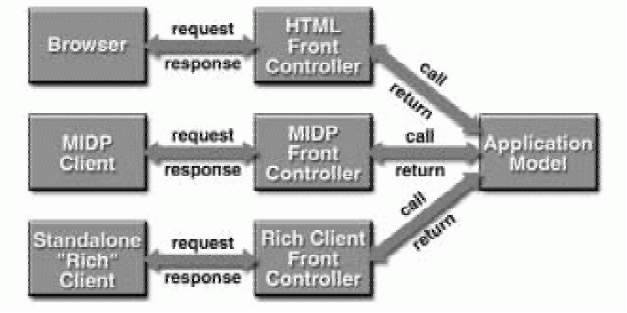
- ***Model***
  Het *model* representeert de enterprise data en de businesslogica die de toegang tot deze data beheerst. Het model is vaak een softwarematige benadering van een bestaand proces uit de realiteit.

- ***View***
  De *view* geeft de inhoud van een model weer en bepaalt hoe deze data gepresenteerd moeten worden. Om de data te benaderen gebruikt de view de business regels van het model. Het is de verantwoordelijkheid van de view om te zorgen dat de presentatie consistent blijft met de toestand van het model.

- ***Controller***
  Een *controller* bepaalt het applicatiegedrag. Het vertaalt de interacties van een gebruiker met de view in taken die door het model uitgevoerd moeten worden. Een voorbeeld van zo'n interactie is het indrukken van een knop of het versturen van een HTTP-request, waarna een taak de toestand van het model moet veranderen. Verder bepaalt de controller ook welke view er getoond wordt naar aanleiding van een interactie of een uitgevoerde taak. Dit is de zogenaamde *control flow*.

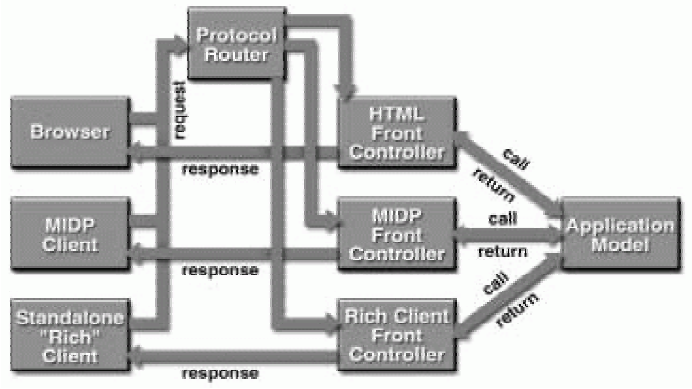
Door de duidelijke scheiding tussen presentatie en applicatiecode zijn de 'duplicatieproblemen' opgelost. De toevoeging van de controller maakt het makkelijker om verschillende views te hebben op hetzelfde model. Bovendien zorgt de contoller ervoor dat pagina's niet direct naar elkaar hoeven te verwijzen. De control flow is ook in handen van de controller.

De Servlet leent zich prima voor het implementeren van een controller, maar het maken van een controller Servlet voor elke pagina is niet ideaal. Een *front controller* is een centrale Servlet die als controller fungeert. Alle requests en responses gaan via deze Servlet, waardoor globale faciliteiten zoals logging en beveiliging hier plaats kunnen vinden. Verder kan door het centrale karakter van dit ontwerp de consistentie met betrekking tot navigatie, templates en inhoudsverversing makkelijker gewaarborgd worden. Ook maakt dit het onderhoud eenvoudiger. Globale veranderingen hoeven namelijk maar op een plaats doorgevoerd te worden.



Figuur 4-4: Meerdere typen clients met meerdere controllers

Het MVC-patroon biedt ook toegevoegde waarde bij het ondersteunen van meerdere typen clients, zoals beschreven in paragraaf 4.2.2. Elk type client heeft dan zijn eigen controller nodig, die gespecialiseerd is in het protocol en de presentatievorm voor dat clienttype (Figuur 4-4). De eerder genoemde applicatiebrede functionaliteiten, zoals logging en beveiliging wil je echter nog steeds graag centraal houden over de verschillende typen clients.

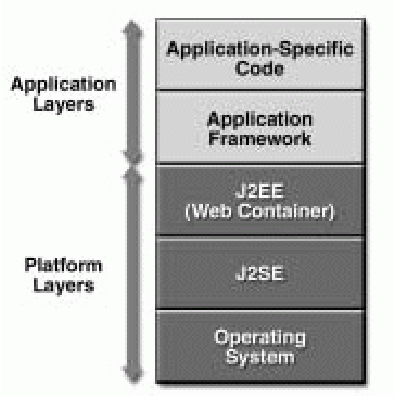


Figuur 4-5: Gebruik van een protocol router

Een oplossing is het introduceren van een *protocol router*, die kan dienen als een centraal stuurmiddel voor alle web clients (Figuur 4-5). Het is een Servlet die alle requests ontvangt, het type client vaststelt en de request doorstuurt naar de bijbehorende controller.

4.2.4 Web application frameworks

In de praktijk wil je je als applicatieontwikkelaar graag bezighouden met de applicatielogica, en niet met de implementatiedetails van complexe ontwerpstructuren. De besproken architectuur in paragraaf Figuur 4-3 is zo'n voorbeeld. Het maken van een 'monsterservet', die alle requests vertaalt naar methode aanroepen, wordt dan al snel een grote "if..else" bak. Na verloop van tijd zal dit onderhoudsproblemen geven [C2].



Figuur 4-6: platform- en applicatielagen voor de webtier

Er is behoefte aan een extra softwarelaag boven op het J2EE-platform, die het detailwerk van de implementatie van deze patroon uit handen neemt. Deze laag heet een *web application framework* en biedt onder andere functionaliteit voor het doorsturen van requests, het aanroepen van methoden in het model en het selecteren en opmaken van de views.
Door de populariteit van de MVC-architectuur zijn er nogal wat web application frameworks op de markt verschenen[17]. Sommige zijn aanbiederspecifiek en geïntegreerd met specifieke servers en tools. Andere zijn vrij beschikbare, open-source projecten. Naast de genoemde basisfunctionaliteit, bieden veel frameworks extra diensten aan en moedigen ze aan tot het gebruik van een MVC-structuur [A7].
Apache Struts is zo'n framework. Het is een open-source project en is op dit moment een van de meest gebruikte frameworks voor het ontwikkelen van web applicaties. Een van de voordelen van Struts is de vertaling, of *mapping*, van een request naar een methode aanroep. De Struts-controller kan requests afvangen door middel van de extensie[18] en gebruikt het *command* design patroon voor het aanroepen van de juiste methode. Dit command pattern maakt gebruik van inheritance in plaats van if..else statements. Als applicatieontwikkelaar creëer je een afgeleide klasse van de Struts Action-klasse en implementeer je de "perform" methode. De controller maakt een instantie aan van de correcte Action-klasse op basis van de ontvangen requestnaam. Deze mapping gebeurt met behulp van de deployment descriptor. Een wijziging in de mapping kan zo eenvoudig plaatsvinden, zonder aanpassingen in de programmacode.

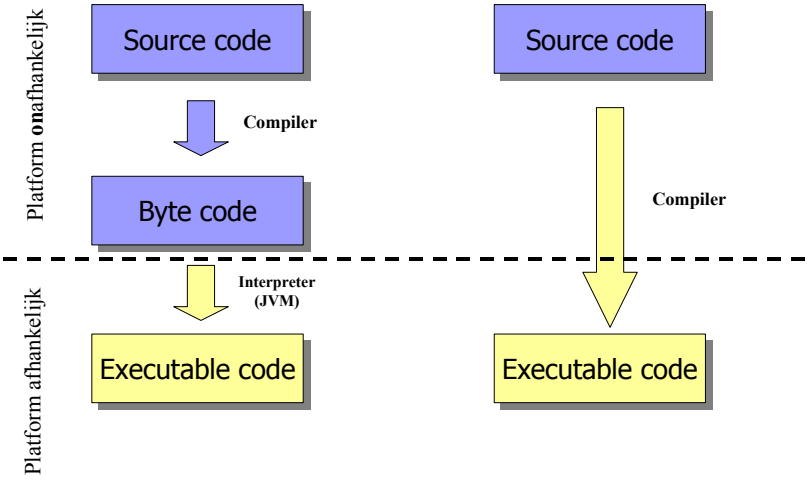
[17] JavaServer Faces, Apache Struts, J2EE blueprints WAF, Apache Turbine
[18] Een onofficiële Struts standaard is de ".do" extensie

Een ander belangrijk voordeel van Struts zijn de custom tags. Deze tags halen de Java-code weg uit de views. Struts tags zijn hetzelfde concept als JSTL tags[19]. Een aantal van de Struts tag-libraries zijn zo overbodig geworden door de introductie van JTSL [A9].

4.3 Businesslogica-tier

4.3.1 Platformonafhankelijkheid

SUN Microsystems bracht in 1995 Java uit met de 'write once, run anywhere' (WORA) belofte. [A5] Dit betekent dat applicaties geschreven in de Java-programmeertaal op elk platform zouden moeten draaien. Java moest een platform neutrale taal zijn. Dit is in theorie realiseerbaar door de broncode niet direct te compileren naar uitvoerbare code, maar naar een tussenvorm: de bytecode. Met behulp van een *interpreter* wordt deze bytecode uitgevoerd. Zo'n interpreter wordt ook wel een *virtual machine* genoemd, omdat de gecompileerde code als het ware op deze 'machine' wordt uitgevoerd.



Figuur 4-7: De platformonafhankelijkheid van Java

De interpreter zet de bytecode naar uitvoerbare code, en is daarom wel platform afhankelijk. Om een Java applicatie te kunnen draaien op een platform moet er dus een *Java Virtuele Machine (JVM)* bestaan voor dit platform. Java heeft veel ondersteuning gekregen van grote software bedrijven. Zo bestaan er Java virtuele machines voor veel platformen, waaronder Solaris, Linux, Macintosh, OS/2 en alle versies van MS Windows. Hiermee hebben ze in grote mate voldaan aan de "write once, run anywhere" belofte.
Omdat elke Java-applicatie op een willekeurige virtuele machine moet kunnen draaien, is het een vereiste dat de Byte code die door een Java-compiler gegenereerd wordt in zijn geheel aan de Java-standaard voldoet. Zo bracht Microsoft een tijd geleden J++ uit, dat grotendeels dezelfde bytecode genereerde als een Java compiler, maar waarbij bijvoorbeeld voor grafische doeleinden gebruik werd gemaakt van de Windows API. Hierdoor konden

[19] zie paragraaf 3.2.5

applicaties geschreven in J++ niet langer draaien op andere platformen, hetgeen net de essentie was van de bytecode.
De CTM van een J2EE-model draait binnen een Java virtuele Machine en is dus platformonafhankelijk. J2EE draait overigens niet op alle Windows-platformen.

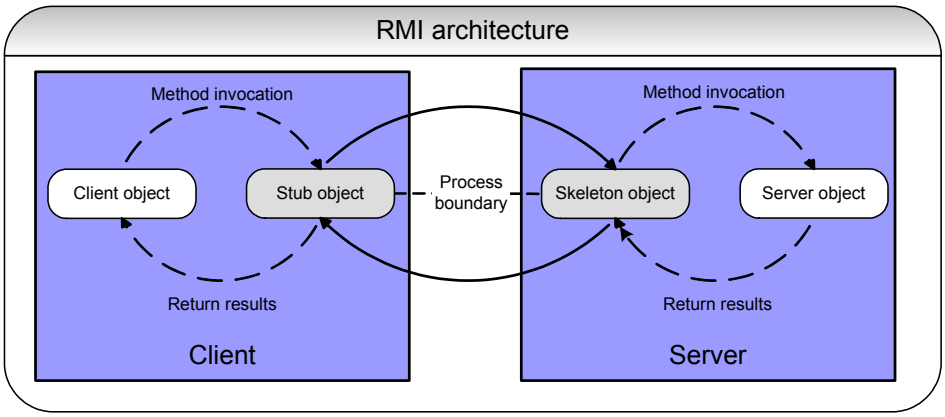
4.3.2 Implementatie onafhankelijkheid

"Java 2, Enterprise Edition" en "Enterprise JavaBeans" en zijn specificaties. Ze definiëren een standaard voor respectievelijk applicatieservers en voor server-side componenten [A3]. SUN definieert deze standaard, maar laat de commerciële implementaties over aan andere marktpartijen. Zelf leveren ze alleen een zogenaamde 'referentie implementatie', die dient als een operationele definitie van het J2EE-platform. Het is een vrij verkrijgbaar product, dat niet voor commerciële doeleinden gebruikt mag worden, maar wel voor demonstraties, onderzoek en het maken van prototypen. Tevens gebruiken ontwikkelaars deze implementatie om de portabiliteit van een applicatie te testen [B1].
De aanbieders van een J2EE-applicatieserver moeten een *compatibility test* doen, om een licentie te krijgen van SUN. Ze draaien hiervoor de J2EE *compatibility test suite,* die bestaat uit meer dan 6000 tests. Deze tests geven aan of alle J2EE-diensten (API's) zijn geïmplementeerd, of de component technologieën aanwezig zijn en samenwerken en tenslotte of applicaties consistent op meerdere platformen gedeployed kunnen worden.
Het maakt niet uit hoe aanbieders de diensten implementeren, zolang ze maar aan de specificaties voldoen. Hierdoor kunnen applicatieontwikkelaars ervan op aan dat hun J2EE-applicatie draait op een J2EE-applicatieserver van een willekeurige erkende aanbieder. Een J2EE-applicatie is dan zogezegd *implementatie onafhankelijk*.
Deze implementatie onafhankelijkheid heeft naast portabiliteitsmogelijkheden ook nog een ander voordeel: het vormt een groeimodel voor 3rd party producten [A3]. Het is voor aanbieders van deze producten aantrekkelijk dat meerdere grote partijen de standaard ondersteunen. Hun product werkt immers op alle implementaties.

4.3.3 Gedistribueerde objecttechnologie

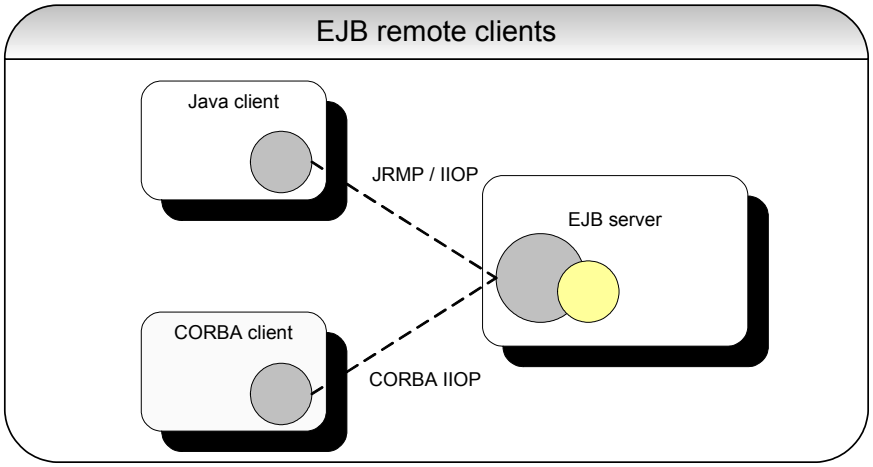
Businessobjecten zijn autonome objecten die een specifieke taak vervullen binnen een applicatie. Er is sprake van gedistribueerde objecten wanneer er communicatie plaatsvindt tussen deze objecten, terwijl ze niet op dezelfde lokatie staan. Dit kan zijn op een andere machine, of op dezelfde machine maar niet dan niet binnen hetzelfde proces. In termen van Java betekent dit het communiceren met objecten die binnen verschillende Java virtuele machines draaien. Deze objecten communiceren met elkaar via een *Remote Method Invocation (RMI)* netwerkprotocol [A13].
Het principe werkt als volgt. Een ontwikkelaar specificeert de *remote interface* voor een businessobject. Dit zijn de methoden die een client kan aanroepen. Vervolgens implementeert hij deze methode in een klasse, het *Server-object*. De RMI-compiler genereert op basis van de remote interface en de implementatieklasse een client *stub* en een server *skeleton*. Deze objecten

zorgen voor de communicatie tussen het client-object en het server-object over een netwerk. De client roept een methode aan van het stub-object, die dit verpakt (*marshalling)* en transparant doorstuurt naar het skeleton-object. Het skeleton-object pakt de aanroep uit (*unmarshalling),* roept de server-object methode aan en stuurt het resultaat terug naar het stub-object. Deze stuurt het resultaat door naar het client-object. Voor het client-object lijkt het alsof de werkelijke uitvoering van de methode in hetzelfde proces gebeurd is [A13]. Figuur 4-8 geeft een schematisch overzicht. De grijze objecten worden automatisch gegenereerd door de RMI-compiler.



Figuur 4-8: RMI-architectuur

Elk protocol heeft als hoofdtaak het bereiken van *lokatietransparantie.* Objecten moeten met elkaar kunnen communiceren, zonder van elkaar te weten waar ze zich bevinden.



Figuur 4-9: EJB RMI clients

Er zijn verschillende protocollen in omloop met elk hun voor- en nadelen. Zo maakt Microsoft in het .NET platform gebruik van het Distributed Component Object Model (DCOM), maar dat wordt alleen ondersteund op een Windows platform. CORBA IIOP is een volwassen protocol, dat zowel taal als platformonafhankelijk is. Het kan gebruikt worden om systemen te integreren

die in verschillende talen zijn geprogrammeerd. Java kent ook een RMI-protocol. Oorspronkelijk was dit het Java Remote Method Protocol (JRMP), dat alleen gebruikt kan worden tussen Java applicaties. Bovendien biedt het geen ondersteuning voor beveiliging- en transactiediensten. Wel was het mogelijk om sub- of supertypen van de remote interface te creëren, zodat polymorfische remote objecten kunnen bestaan. Met de komst van EJB 1.1 kwam de ondersteuning van Java-RMI over IIOP.
Naast businessobjecten kunnen ook clients de businesslogica van een J2EE-applicatie benaderen door middel van zo'n protocol. De J2EE-specificatie verplicht de ondersteuning van meerdere protocollen. Hierdoor kunnen verschillende type clients gebruik maken van enterprise beans, zoals te zien is in Figuur 4-9. Mogelijke CORBA clients zijn C++, Visual Basic, Ada, Smalltalk, COBOL en Delphi applicaties [A3].

4.3.4 Java Naming and Directory Interface

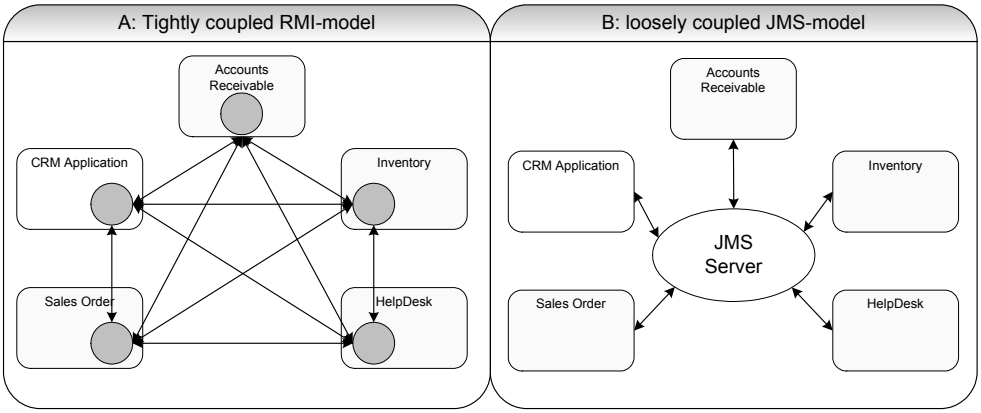
Naming services associëren logische namen met objecten en geven de mogelijkheid om deze objecten te benaderen op basis van hun naam. De associatie tussen een naam en een object heet een *binding* en een set van zulke bindings heet een *context* [A13]. Sommige naming systems gebruik je dagelijks. Bijvoorbeeld bij het gebruik van de verkenner of bij het invoeren van een URL in een web browser. Objecten in een naming system variëren van bestanden in een bestandsysteem en IP-adressen in een DNS-database, tot Enterprise JavaBeans in een applicatieserver en gebruikersprofielen in een informatiedirectory.
Directory services zijn een natuurlijke uitbreiding op naming services. Het belangrijkste verschil is dat een directory service de associatie van attributen met objecten toestaat, zoals een e-mail adres met een user object.
De *Java Naming and Directory Interface (JNDI)* maakt het mogelijk om naming en directory services the benaderen vanuit Java applicaties. Dit wordt ook wel een *lookup API* genoemd. De architectuur van JNDI lijkt een beetje op die van JDBC[20], omdat ze allebei een protocol-onafhankelijke interface aanbieden, boven op een specifieke driver van een protocol. Een service-provider levert de implementatie van de JNDI-interfaces voor een specifieke naming of directory service, net zoals een JDBC-driver de JDBC interface implementeert voor een specifieke database.
De EJB-specificatie verplicht het gebruik van JNDI als een lookup API om verbinding te maken met een EJB-server en om de EJB home interface te lokaliseren [A3]. Zo kunnen EJB-objecten gemakkelijk gevonden worden in een gedistribueerde omgeving.

4.3.5 Java Message Service

Zowel mensen als computers kunnen communiceren door het uitwisselen van berichten over elektronische netwerken. Het meest bekende voorbeeld van een messaging systeem tussen mensen is e-mail. In dit document zijn we echter vooral geïnteresseerd in messaging systemen die de communicatie tussen verschillende software applicaties mogelijk maken in zakelijke systemen. Dit

[20] Zie paragraaf 4.4.1

soort messaging systemen gaan onder de naam *enterprise messaging systems,* of *Message-Oriented Middleware (MOM)* [A11].
Een bericht of *message* is een data pakket met netwerk routing headers. MOM producten verzekeren dat berichten correct verspreid worden tussen applicaties. In feite worden deze berichten uitgewisseld over virtuele kanalen, zogenaamde *destinations.* Applicaties kunnen zich abonneren op een destination om zo de berichten te ontvangen. Naast het verzorgen van communicatie, bieden MOM producten normaliter ook ondersteuning voor fouttolerantie, load-balancing, schaalbaarheid en transactiemanagement, wanneer het gaat om grote hoeveelheden berichten. Een probleem is dat ze afwijkende berichtformaten en netwerk protocollen gebruiken. Daarom is de *Java Message Service (JMS)* ontwikkeld. Het is een Java API die gebruikt kan worden met verschillende MOM producten, net zoals JNDI dat kan bij verschillende naming systems. JMS is dus ondanks de naam zelf geen messaging systeem, maar een abstractie van de interfaces die nodig zijn de communicatie met messaging systems. Binnen het JMS-model hanteert men de volgende terminologie: applicaties die berichten versturen en ontvangen heten *JMS-clients,* message systems heten *JMS-providers.*
Nu kun je je afvragen wat het nut is van een messaging systeem naast het reeds besproken RPC- of RMI-model. Dat maakt communicatie tussen (heterogene) applicaties immers ook mogelijk. Het verschil zit in manier van communicatie. Het RMI-model gebruikt *synchrone communicatie*, wat inhoudt dat de client wacht na een aanroep, tot hij antwoord krijgt van de server. Dit gesynchroniseerde model imiteert zo het gedrag van een systeem dat in hetzelfde proces draait. De taken worden in de voorgedefinieerde volgorde uitgevoerd. Doordat de client blokkeert tot de server antwoord geeft, is hij erg afhankelijk van de server. Een storing op de server heeft onherroepelijk gevolgen voor de client. Deze afhankelijkheid wordt ook wel *tight coupling* genoemd[21].



Figuur 4-10 : applicatiescenario's met RMI en JMS

Het tightly-coupled RMI-model werkt het beste in een n-tier applicatie, waar de onderlinge componenten goed op elkaar afgestemd zijn. Denk bijvoorbeeld

[21] Er zijn opties om deze afhankelijkheid in het RMI-model te verkleinen, zoals het gebruik van multithreading en RPC mechanismen als CORBA one-way call. Echter, deze oplossingen vergen geavanceerde en complexe ontwerpen.

aan de communicatie tussen Servlets en Enterprise JavaBeans. Er zijn echter een aantal applicatiescenario's waar het RMI-model minder geschikt voor is. Een voorbeeld is het integreren van subsystemen binnen een organisatie die geografisch gescheiden zijn. Hier heb je vaak te maken met meerdere communicatielijnen en communicatie in meerdere richtingen. Een voorbeeld is te zien in Figuur 4-10(A). Hier Door de directe communicatie in een RMI-model betekent het toevoegen van een extra subsysteem het updaten van alle bestaande systemen. Bovendien, wanneer een onderdeel van het systeem crasht, hangt met een RMI-model het hele systeem.
Message systemen maken gebruik van *asynchrone communicatie.* Een client stuurt een bericht en wacht niet op een antwoord. Hij kan doorgaan met het uitvoeren van andere taken. De systeemonderdelen staan niet direct met elkaar in verbinding. Clients sturen een bericht naar een JMS-server, die dat doorstuurt naar de opgegeven destination. Deze architectuur vormt een *loosely coupled* systeem, zoals te zien in Figuur 4-10(B). Storingen kunnen natuurlijk nog altijd optreden. JMS garandeert in zo'n geval toch de aflevering van de berichten. Dit gebeurt middels een *store-and-forward* mechanisme. De message server slaat berichten op en stuurt deze opnieuw naar de ontvangende applicatie, die daarvoor "uit de lucht" was.
De J2EE 1.2 specificatie eiste alleen ondersteuning voor JMS-clients. Een J2EE 1.3 aanbieder moet ook een volledige JMS-provider leveren [A13]. De EJB 2.0 specificatie verplicht de ondersteuning van *Message Driven Beans (MDB)*. Het fundamentele verschil met entity en session beans[22] is dat entity en session beans een remote interface aanbieden aan clients, zodat deze middels RMI de methoden kunnen aanroepen. Echter, message-driven beans melden zich aan bij de server voor specifieke asynchrone messages, en starten hun taak bij de ontvangst van zo'n bericht. Qua functionaliteit komen ze overeen met stateless session beans: ze coordineren taken waarbij andere session en entity beans betrokken zijn, zonder een toestand bij te houden. Message-driven beans zijn JMS-destinations voor externe applicaties.

4.3.6 Web Services

Web services zijn de meest recente ontwikkeling op het gebied van distributed computing en misschien wel een net zo belangrijke innovatie als de komst van Java in 1995 en XML in 1998. Het is dan ook de hoofdreden voor de komst van J2EE 1.4. Er zijn vele algemene definities van web services in omloop, maar omdat web services niet gebonden zijn aan een technologie of platform zijn deze vaak erg abstract. De volgende definitie is van toepassing binnen de context van J2EE, EJB, .NET en de meeste andere Web service platformen:

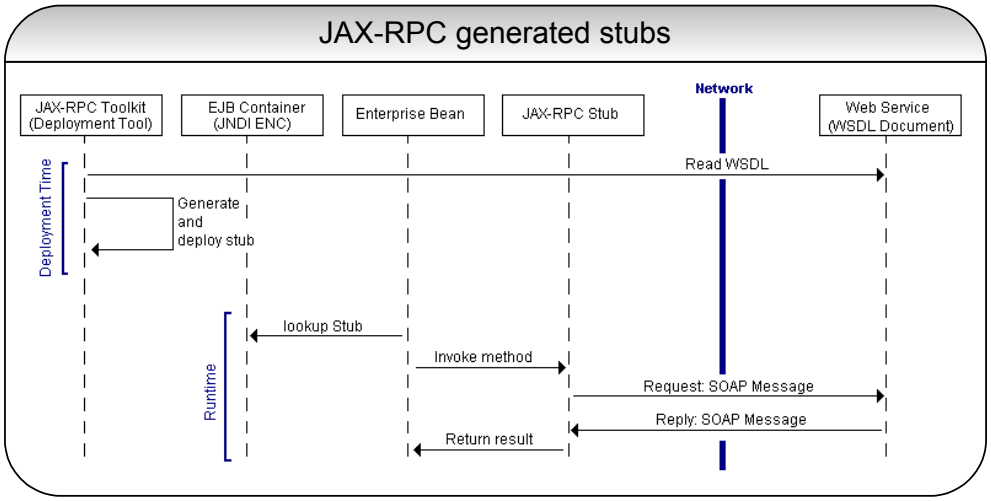
> *"Web services zijn netwerkapplicaties die SOAP en WSDL gebruiken om informatie in de vorm van XML documenten uit te wisselen."* [B15]

Een hele praktische omschrijving van web services luidt:

> *"Web services zijn herbruikbare softwarecomponenten. Ze zijn ontworpen om systeemintegratie te vergemakkelijken."* [C6]

[22] Entity beans en session beans worden beschreven in paragraaf 3.2.3 Component Model Architectuur

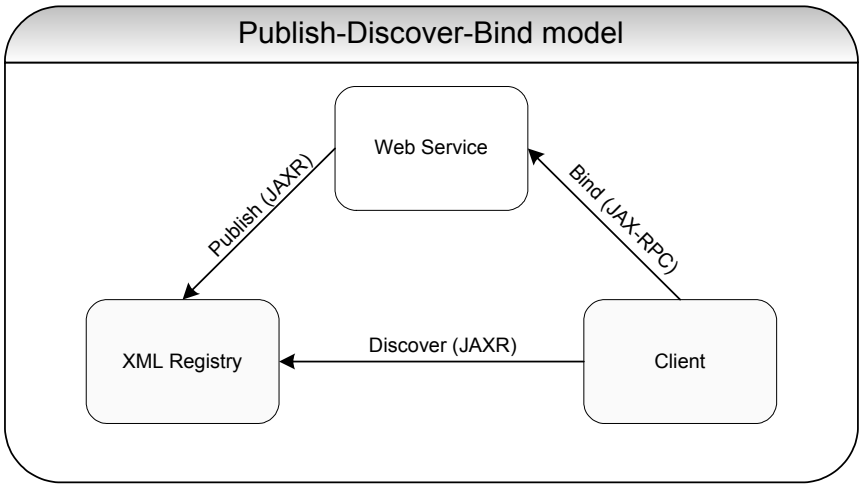
SOAP staat voor *Simple Object Access Protocol* en is een op XML gebaseerd lightweight protocol dat zowel gebruikt kan worden bij synchrone als bij asynchrone communicatie. Het definieert de structuur van berichten. In principe kan SOAP gebruikt worden met verschillende transportprotocollen, zoals TCP/IP en JMS, maar de meest voorkomende combinatie is met HTTP. SOAP is flexibel, uitbreidbaar en een open standaard. Door die standaardisatie, iets wat de voorlopers[23] misten, is SOAP geadopteerd door bijna elke grote aanbieder [B17].
Web Services Description Language (WDSL) is een XML-document dat web services beschrijft met behulp van een *Interface Definition Language (IDL)*. Deze beschrijving bevat informatie over het berichtformaat, de functionaliteit van de service, het gebruikte protocol en het Internet adres. [B16].
In essentie representeren web services een nieuw gedistribueerde objecttechnologie, net zoals CORBA IIOP en Java RMI maar met een paar cruciale verschillen. Het belangrijkste verschil is dat web services écht platformonafhankelijk zijn. Het is mogelijk om web services te maken op elk platform, in elke programmeertaal. CORBA en Java RMI claimen hetzelfde, maar hebben hun eigen platform nodig. Zo werkt Java RMI alleen met een Java virtuele machine en de Java-programmeertaal. Het IIOP protocol vereist een uitgebreide infrastructuur zoals een CORBA ORB, dat ontwikkelaars beperkt tot een beperkt aantal aanbieders die CORBA ondersteunen, of tot de J2EE-omgeving (zie paragraaf 4.3.3). Een ander belangrijk voordeel van web services ten opzichte van andere gedistribueerde object technologieën, is dat ze voortborduren op een bestaande technologische infrastructuur, wat het makkelijker maakt voor aanbieders ze te implementeren. SOAP en WDSL zijn gebaseerd op XML, waarvoor reeds uitgebreide ondersteuning bestaat, in de vorm van XML-parsers. Bovendien worden web services berichten normaliter verstuurd over het TCP/IP-protocol, dat ondersteund wordt door bijna elk modern software platform en de meeste programmeertalen. Web services leggen de nadruk op protocollen, terwijl CORBA, DCOM en Java-RMI een implementatie vereisen.



Figuur 4-11: JAX-RPC sequentiediagram

[23] Zoals CORBA IIOP, DCOM en Java RMI-JRMP

Een *SOAP toolkit* is een API voor het versturen en ontvangen van SOAP berichten. Deze toolkits zijn verder te categoriseren in stub-based toolkits en API-based toolkits. *Stub-based* toolkits maken gebruik van RPC stubs en maken de web service, vanuit het perspectief van de client, een object met methoden. De *JAX-RPC* API is het standaard stub-based programmeermodel voor J2EE en is in essentie Java RMI over SOAP. Een stub generatie tool leidt de remote interface, de methoden en de parameters direct af van het WDSL document. De stub zelf wordt ook afgeleid van het WDSL document en implementeert het protocol, de codeerstijl en het internet adres van de web service. Dit gebeurt tijdens de deployment fase, zoals het sequentiediagram in Figuur 4-11 laat zien. Tijdens run time kan de stub dan gebruikt worden om methoden aan te roepen van de web service. Het is met de JAX-RPC API ook mogelijk dat de stub dynamisch tijdens run time gegenereerd wordt. Dit is het *dynamic proxy* programmeermodel. *API-based* toolkits modelleren expliciet de structuur van een SOAP bericht. Java ontwikkelaars kunnen de SAAJ API gebruiken om SOAP messages te creëren, te lezen of aan te passen. SAAJ maakt het mogelijk om object georiënteerde representaties van SOAP messages te maken, inclusief eventuele attachments. Web services kunnen dankzij de twee typen toolkits een synchrone en een asynchrone architectuur hebben.



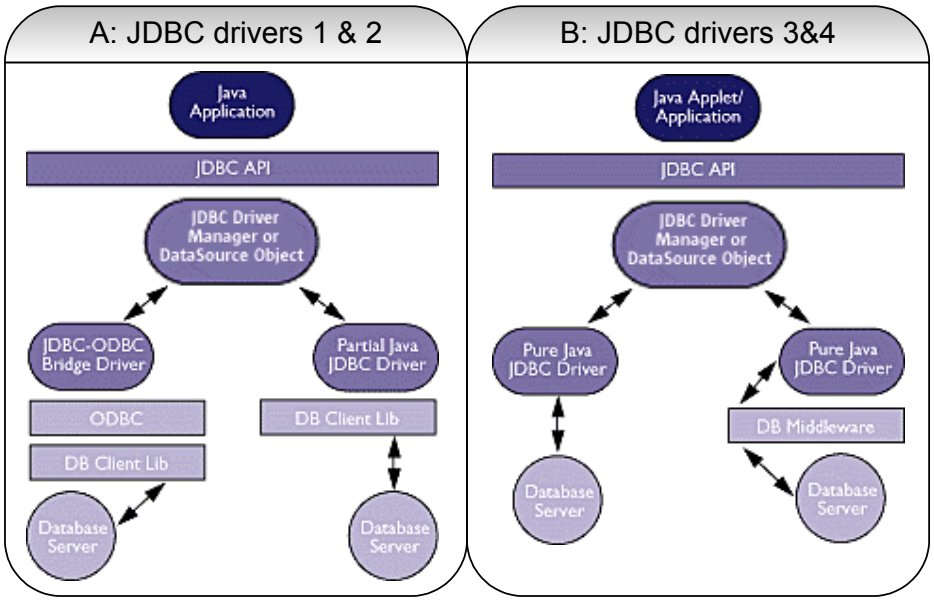
Figuur 4-12: J2EE publish-discover-bind model

Het publiceren van web services voor de buitenwereld gaat met behulp van XML-registries, te zien in Figuur 4-12. Een J2EE-applicatie publiceert zijn web services met behulp van de Java API for XML-Registries (JAXR). Een client kan vervolgens de registry doorzoeken om de gewenste web service te "ontdekken". Tenslotte bindt hij zich aan de web service en kan hij methoden aanroepen (of asynchrone berichten versturen) [C4].

4.4 EIS tier

4.4.1 Java Database Connectivity

De *Java Database Connectivity (JDBC)* API is een standaard service die de verbinding vormt tussen de Java-programmeertaal en een database [B8]. Deze service bestaat uit twee interfaces: de JDBC API voor applicatieontwikkelaars en de JDBC-driver API.
Met behulp van de JDBC API kan een applicatieontwikkelaar verbinding met een database maken, resultaten opvragen en data updaten met behulp van de *Structured Query Language (SQL)*. Een JDBC-driver zorgt voor de connectie met een database van een specifieke leverancier. Vaak ontwikkelt de leverancier zelf de JDBC-driver zodat deze geoptimaliseerd is voor zijn product. In Figuur 4-13 is te zien hoe de communicatie tussen een Java-applicatie en een database verloopt bij het gebruik van een JDBC-driver[24]. JDBC is een open, leverancier en platformonafhankelijke standaard voor de communicatie tussen Java en een Database Management Systeem [A6].

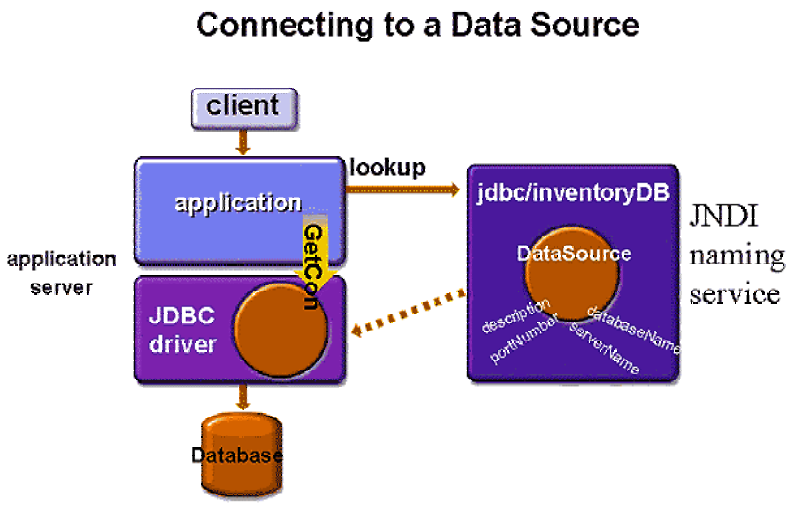


Figuur 4-13: JDBC Communicatiepaden

In 1996 bracht Sun de eerste versie van de Java Database Connectivity set uit [B9]. Met behulp van SQL konden programmeurs hiermee verbinding maken met een database, resultaten verkrijgen en updaten. Twee jaar later komt versie twee uit, die geavanceerde data typen ondersteunt, verbeterde navigatie biedt binnen resultaatsets en de mogelijkheid geeft tot batch updates. Deze versie kent naast de kern API ook een zogenaamde extensie API die zorg draagt voor gedistribueerd transactiemanagement en een uitgebreider connectiemanagement. Zo biedt de JDBC 2 extensie API ondersteuning voor *connection pooling*, waarbij database connecties niet daadwerkelijk gesloten worden, maar daarentegen bewaard en hergebruikt voor een volgende client.

[24] Er zijn vier verschillende JDBC-drivertypen. Deze worden besproken in paragraaf 5.4.2.

Daarnaast kan de *Java Naming and Directory Interface (JNDI)* gebruikt worden om een connectie te verkrijgen naar een database. De JNDI service lokaliseert een datasource, op basis van een logische naam die een database instantie beschrijft en maakt verbinding met behulp van de opgegeven JDBC-driver. Dit is te zien in Figuur 4-14 en heet *connection naming*. Het gebruik van JNDI maakt de applicatie code onafhankelijk van een specifieke JDBC-driver en van de database URL.
In de JDBC 3.0 API specificaties zijn de verbeteringen voornamelijk van technische aard [B10]. Hierbij valt te denken aan verfijnde configuratiemogelijkheden voor connectiemanagement, metadata en resultaatsets. Ook legt deze specificatie het verband met de Java Connector Architectuur (zie volgende paragraaf). In feite is de JDBC namelijk ook een *pluggable resource adaptor* [A7].



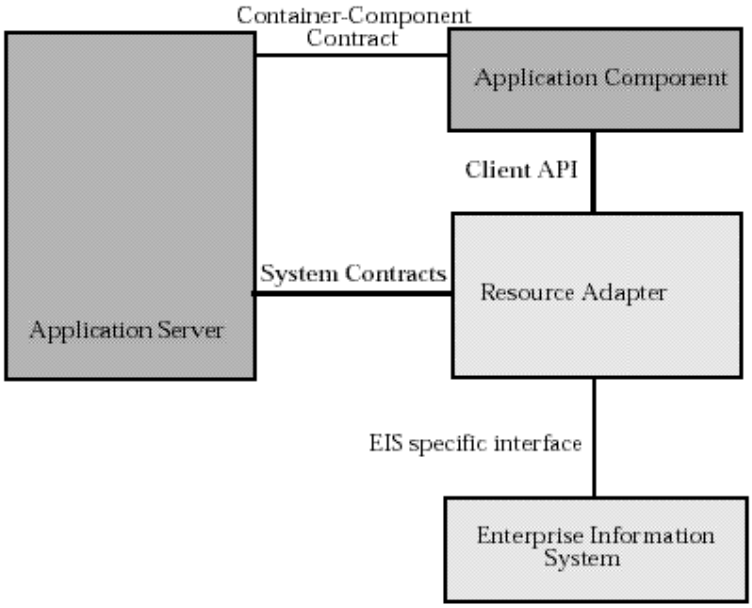
Figuur 4-14: JDBC en JNDI

Het is mogelijk om database statements te sturen die alleen geldig zijn voor een database van een specifieke aanbieder. Dit doet echter af aan de openheid van de JDBC standaard en beperkt daarmee de portabiliteit van de applicatie. Het is dan ook een best practice om alleen *ANSI SQL* te gebruiken in de JDBC aanroepen, tenzij de functionaliteit van de applicatie-aanbieder specifieke SQL vereist [A8].
Alle applicatieservers die voldoen aan de J2EE 1.3 standaard ondersteunen de JDBC 2.0 kern API plus een aantal diensten van de JDBC 2.0 Extensie API, waaronder gedistribueerd transactiemanagement, connection pooling en connection naming. De JDBC 3.0 API zal onderdeel uitmaken van de J2EE 1.4 specificatie.

4.4.2 J2EE Connector Architectuur

Vaak bestaat er binnen een bedrijf al een informatiesysteem, dat de vereiste informatie infrastructuur bevat voor de bedrijfsprocessen. Zo'n *Enterprise Information System (EIS)* kan bijvoorbeeld een Enterprise Resource Planning systeem zijn, een TP-systeem of een bestaande applicatie die niet in Java is

geschreven [A7]. De integratie met een EIS is een belangrijke kwestie voor een succesvolle e-commerce applicatie. Dit is veel goedkoper en sneller realiseerbaar dan het bouwen van een compleet nieuw systeem.
Het integreren met bestaande applicaties gaat onder de naam *Enterprise Application Integration (EAI)*. Dit gebeurt met behulp van een *Resource Adapter*, een driver van een specifiek informatiesysteem die een interface aanbiedt aan de client. Tot voor kort was er geen standaard specificatie voor de integratie van verschillende informatiesystemen met het Java platform. Elke EAI-tool leverancier bouwde zijn eigen resource adapter interface, waardoor er voor elke EAI-tool / EIS combinatie een resource adapter nodig was [B12].



Figuur 4-15: Overzicht van de Connector Architectuur

De *J2EE Connector Architecture (JCA)* definieert standaard contracten voor de communicatie tussen een J2EE-applicatieserver en een informatiesysteem [B11]. De EIS leverancier of een EAI onderneming maakt een resource adapter voor het informatiesysteem, terwijl de aanbieder van de applicatieserver een uitbreiding maakt voor de J2EE connector architectuur. Het concept is gelijk aan JDBC, waarbij de JDBC een resource adapter is voor relationele databases. Er is nu maar 1 resource adapter nodig per EIS, die werkt op alle J2EE-applicatieservers met JCA ondersteuning. Merk op dat de interface tussen de resource adapter en het informatiesysteem buiten de scope van J2EE valt. Vaak is dit een *proprietary* protocol.
Als we wat verder inzoomen op de integratie met behulp van de connector architectuur, dan zijn er twee aspecten: systeemcontracten en applicatie interfaces.
Systeemcontracten zorgen ervoor dat de applicatieserver en het informatiesysteem samen de onderliggende mechanismen van een gedistribueerd systeem kunnen beheren, zonder dat de applicatieontwikkelaar hiermee belast wordt. Bij de eerste release van de JCA waren er drie systeemcontracten. Het connection management contract zorgt voor de

connection pooling naar het EIS. Vervolgens stelt het transactie management contract de applicatieserver in staat om zowel transacties te managen over meerdere resource managers als het overdragen van een transactie aan een resource manager. Tenslotte is er het security contract dat authenticatie toepast bij de benadering van het informatiesysteem. Met de huidige 1.5 specificatie van de JCA is asynchrone communicatie mogelijk, hetgeen inhoudt dat het informatiesysteem ook informatie kan sturen naar de applicatieserver en ook bijvoorbeeld transacties kan starten en beheren. Tevens is er een workflow-contract, dat ervoor zorgt dat de applicatieserver op threadniveau het systeem kan managen.
De *applicatie interface,* of *client interface,* is de functionaliteit die aangeboden wordt aan de applicatieontwikkelaar. J2EE schrijft een gemeenschappelijke interface voor, de *Common Client Interface (CCI)*, maar deze is niet verplicht. De makers van een resource adapter kunnen ook hun eigen client API aanbieden. In de praktijk is deze interface met name voor aanbieders van EAI tools. Ze kunnen zo met een algemene interface en zo min mogelijk EIS specifieke code een integratietool bouwen. J2EE-ontwikkelaars gebruiken vaak deze tools en benaderen niet direct de CCI.

5 Performance & Schaalbaarheid

5.1 Inleiding

De performance van een systeem is misschien wel na de kernfunctionaliteit het belangrijkste criterium van elke willekeurige applicatie. Het enige soort applicaties waarbij performance geen rol hoeft te spelen zijn educatieve opdrachten. Performance kent meerdere aspecten, waaronder de responstijd naar gebruikers, de doorlooptijd van batchprocessen en het verwerkingsniveau, zoals bijvoorbeeld het aantal transacties per minuut [A8].
Gedistribueerde systemen hebben ook nog eens te maken met *schaalvergroting*. Dit kan een (verwachte) toename van het aantal gebruikers, data of objecten binnen het systeem betekenen. De schaalbaarheid van een applicatie zegt iets over het omgaan met dergelijke schaalvergrotingen, zonder dat dit ten koste gaat van de gestelde performance eisen. Dit is de directe link tussen performance en schaalbaarheid bij enterprise-applicaties.
Een goed schaalbare applicatie vangt de schaalvergroting op door het uitbreiden van de hulpbronnen, zoals het aantal servers. De software van een goed schaalbare applicatie hoeft niet te wijzigen als gevolg van zo'n toename.

5.1.1 Criteria

Een gedistribueerd systeem moet aan vooraf gestelde performance-eisen kunnen voldoen. In de meeste gevallen betekent dit dat de responstijd naar gebruikers toe binnen gestelde grenzen moet liggen [A8].
Gedistribueerde systemen hebben niet allemaal hetzelfde schalingsniveau. Een gedistribueerde architectuur moet het daarom mogelijk maken om enterprise-applicaties te realiseren met een afwijkend schalingskarakter. Systeem- en applicatiesoftware hoeven niet te wijzigen als gevolg van een verwachte schaalvergroting [A2].

5.1.2 Performance & schaalbaarheid binnen J2EE

Een belangrijke eigenschap met betrekking tot performance en schaalbaarheid van een J2EE-applicatie is haar server-side architectuur. Het goed inrichten van deze architectuur kan voor een hoop performancewinst zorgen. De harde schijf heeft bijvoorbeeld veel impact op de performance van de database, terwijl voor de performance van de applicatieserver, de processorsnelheid een belangrijke factor is. Deze performance-afwegingen zijn echter gerelateerd aan de architectuur en niet specifiek voor J2EE. Dit hoofdstuk zal daarom hier niet verder op ingaan.
Aan de presentatie kant zitten een aantal sturingsmogelijkheden binnen het J2EE-model. Het gebruik van JavaServer Pages en Servlets biedt een performancevoordeel ten opzichte van het oude Common Gateway Interface model. Ook lenen deze technieken zich voor het gebruik van caching, wat de responstijd versnelt en de server ontlast. Met behulp van load-balancing mogelijkheden van een webserver kunnen webapplicaties tevens schaalbaar gemaakt worden.

Op de applicatielogica-tier is het gebruik van Enterprise JavaBeans een belangrijke afweging tussen performance en schaalbaarheid. Een fundamenteel voordeel van EJB-servers is dat ze zware, intensieve taken kunnen uitvoeren terwijl ze een goede performance behouden [A3]. EJB-servers verbeteren de performance door hulpbronnen te delen en objectinteracties te synchroniseren. Naarmate het aantal gebruikers toeneemt, zal het aantal objectinstanties in het systeem toenemen. Enterprise JavaBeans ondersteunt twee resource management technieken om grote aantallen beans makkelijker te managen: *instance pooling* en *activation*. Een van de primaire diensten van een EJB-server is het ondersteunen van concurrency, waarbij meerdere clients tegelijkertijd een bean kunnen benaderen via RMI of JMS. Tenslotte bespreken we nog wat EJB design patterns, die de performance verbeteren door het netwerk verkeer te beperken.
Voor een snelle communicatie met een database biedt de JDBC API de mogelijkheid om gebruik te maken van connection pooling. Ook zijn er nog een aantal wijze lessen vanuit de praktijk; de zogenaamde "JDBC best practices".

5.2 Presentatie tier

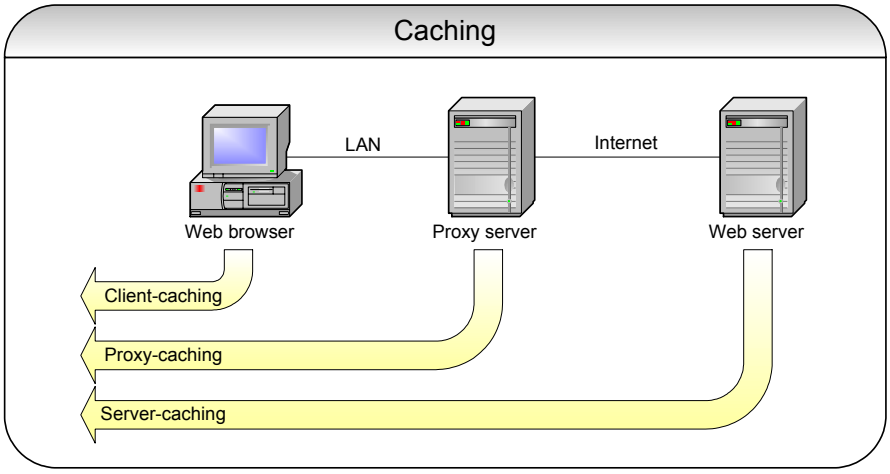
5.2.1 JavaServer Pages en Servlets

Het afhandelen van gebruikersrequests bepaalt met name de performance in de webtier. Het CGI-model is wat dat betreft geen efficiënte oplossing. Voor elk request start de server een nieuw proces, laadt de interpreter plus het script en sluit deze na afloop weer af [A9].
Servlets draaien als een aparte thread binnen een proces, wat de overhead per request aanzienlijk vermindert. Ook hoeft de interpreter niet opgestart te worden. Een persistente Java virtuele machine op de webserver voorkomt dit. Een laatste tijdswinst is de mogelijkheid om gebruik te maken van resources die ook tussen de requests door in het geheugen blijven, zoals een database connectie of een persistente toestand van een Servlet.
JavaServer Pages worden, zoals eerder gezegd, gecompileerd naar een Servlet. Dit gebeurt altijd voordat de server een request afhandelt. Hierdoor krijgen JSP's dezelfde performance voordelen als Servlets.

5.2.2 Caching

Webobjecten zoals JavaServer Pages en Servlets kunnen gebruik maken van caching. *Caching* is een techniek waarbij bezochte data opgeslagen wordt op schijf, zodat deze opnieuw kan worden verstuurd. Hierdoor kan een herhaalde vraag naar opgeslagen data afgehandeld worden op de cache-lokatie, waardoor er geen verkeer van de cache-lokatie naar de oorspronkelijk bron nodig is. Ook hoeft de oorspronkelijke bron de inhoud niet opnieuw te genereren. Cachen kan op drie plaatsen: op de client in de browser cache, op de proxy server, of op de webserver [A8]. Figuur 5-2 geeft een schematisch overzicht.
Web-browsers hebben allemaal een cache. Webobjecten kunnen gebruik maken van deze client cache door rekening te houden met de "last modified" HTTP-header. De browser geeft de datum van de opgeslagen pagina door aan

de server, waarop deze alleen een volledige pagina stuurt wanneer de inhoud van deze pagina daarna veranderd is.
Het webverkeer van grote bedrijven en service-providers gaat vaak via een proxy-server. Deze server heeft een gemeenschappelijke cache waar bezochte webpagina's worden bewaard. Wanneer een gebruiker een reeds opgeslagen pagina wil benaderen, genereert de proxy-server een respons, zonder daadwerkelijk de oorspronkelijke webserver te benaderen. Web objecten kunnen helpen met deze manier van cachen door de "Expires" HTTP-header mee te geven. Wanneer de gebruiker een opgeslagen pagina voor haar expiratiedatum opvraagt, raadpleegt de proxy-server de webserver niet.



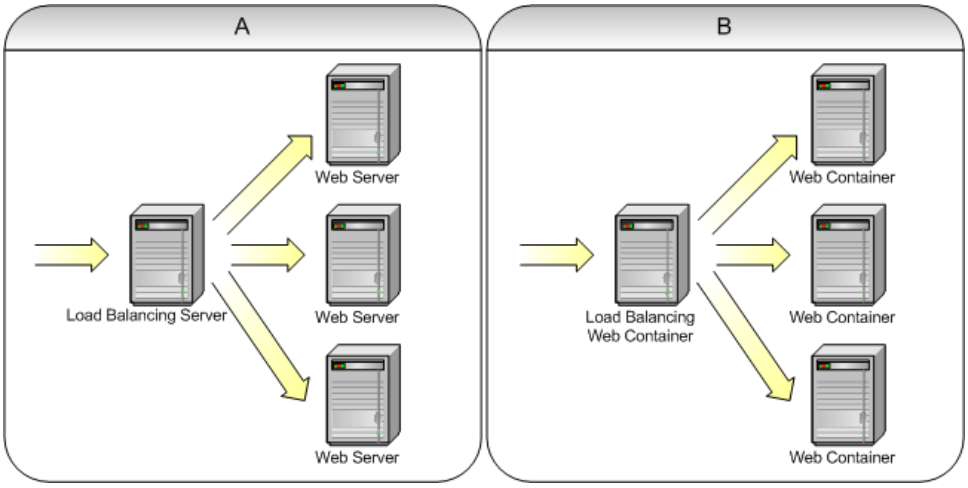
Figuur 5-1: Caching mogelijkheden

Tenslotte is er ook nog caching op de webserver mogelijk. Hier gebeurt het cachen op basis van een sessie of applicatie breed, waarbij het niet beperkt wordt tot een gebruiker of een organisatie. Deze techniek kan ook delen van pagina's opslaan en werkt goed samen in combinatie met statische, voorgegeneerde inhoud, zoals plaatjes.

5.2.3 Load-balancing

Een schaalbare applicatie kan een toenemend aantal gebruikers aan door de hardware configuratie uit te breiden, in plaats van de applicatiesoftware zelf. Een veel gebruikte techniek is load-balancing. *Load-balancing* is het verdelen van reken- en communicatieactiviteiten over meerdere servers binnen een netwerk, zodat een enkelvoudige server niet overbelast raakt.
In deze paragraaf komen twee scenario's voor load-balancing aan de orde: het partitioneren van webservers en webcontainers [A9]. Er is met betrekking tot load-balancing een essentieel verschil tussen webservers en webcontainers. Webservers zijn verantwoordelijk voor de statische inhoud, zoals plaatjes en HTML-pagina's, terwijl een webcontainer de Servlets en JSP-pagina's host, die de dynamische inhoud genereren.
Voor webapplicaties met veel verkeer kunnen meerdere webservers ingezet worden, zoals te zien in Figuur 5-2(A). Een load-balancing server vangt de requests af via internet en stuurt deze door naar een webserver. De beste performance bij de opstelling wordt verkregen door een webserver te gebruiken

die de webcontainer in hetzelfde proces draait, om de onderlinge communicatie te minimaliseren. De meeste grote webservers[25] ondersteunen in-proces communicatie tussen de webserver en de webcontainer.



Figuur 5-2: Load-balancing

Wanneer de webserver de vraag naar statische inhoud aankan, maar niet de vraag naar JSP's en Servlets, dan kan voor een architectuur gekozen worden waarbij de webcontainers over meerdere servers verdeeld worden, zoals te zien in Figuur 5-2(B). Het daadwerkelijke verdelen wordt gedaan door een load-balancing webcontainer.
Binnen beide configuraties is de afhandeling van sessiedata lastig. De meeste containers houden deze data alleen bij in het geheugen. In zo'n geval moeten alle requests binnen een sessie naar dezelfde server gestuurd worden. Geavanceerdere containers kunnen sessiedata ook op disk of in een database opslaan. Dan kan elk request naar een willekeurige server gedistribueerd worden. Dit valt buiten de J2EE-specificatie en is dus een item in de concurrentiestrijd tussen de leveranciers.

5.3 Businesslogica-tier

5.3.1 Enterprise JavaBeans

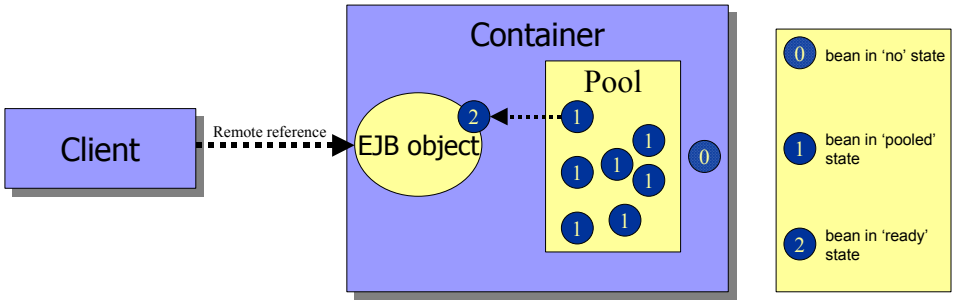
Een belangrijke ontwerpbeslissing die genomen moet worden is het wel of niet gebruiken van Enterprise JavaBeans in een J2EE-applicatie. Hoewel EJB vandaag de dag gezien wordt als het walhalla op het gebied van gedistribueerde applicatieontwikkeling, is het in veel gevallen een voorbeeld van "met een kanon op een mug schieten". Ze zijn niet altijd de juiste oplossing voor een probleem [A8, A9].
Enerzijds heeft deze keuze te maken met een functionaliteitaspect. Gedistribueerd transactiemanagement zal bijvoorbeeld moeilijk realiseerbaar zijn zonder het gebruik van Enterprise JavaBeans. Anderzijds is de afweging tussen schaalbaarheid en performance belangrijk. Het gebruik van Enterprise

[25] Zoals Apache's Tomcat, BEA's WebLogic, Caucho Technology's Resin en New Atlanta's ServletExec

JavaBeans vergroot de schaalbaarheid, maar stelt hoge eisen aan de hardware, wat ten koste van de performance kan gaan bij kleinschalige applicaties. Voor dit soort applicaties kan het beter zijn om voor een architectuur te kiezen met alleen JavaBeans[26], JSP's en Servlets. De JavaBeans nemen dan de taak van het model in het MVC-patroon over. Met behulp van de eerder besproken load-balancing techniek kunnen deze applicaties schaalbaar gemaakt worden.
Enterprise JavaBeans zijn bedoeld voor de wat complexere applicaties waar schaalbaarheid een belangrijk criterium is. Zo zijn de meeste diensten die een EJB-server levert, heel bruikbaar voor het schrijven van schaalbare applicaties, maar leggen ze een grote claim op de aanwezige resources. Als de applicatie niet gericht is op deze schaalbaarheid kan het gebruik van EJB's de performance verslechteren, door het onnodig cachen en poolen van objecten, controleren van deployment descriptors en het veroorzaken van extra communicatieoverhead tussen de webtier en de EJB-tier. De volgende paragrafen hebben dan ook voornamelijk betrekking op de schaalbaarheidsmogelijkheden van EJB.

5.3.2 Instance pooling

Met *instance pooling* wordt bedoeld dat de EJB-container een "bak"[27] met enterprise beans bewaart, om het aantal component-instanties te beheren [A3]. Wanneer een client een methode van zo'n component wil aanroepen, gaat de component uit de bak en wordt hij aan de client gekoppeld. Het is minder belastend voor het systeem om instanties op deze manier te hergebruiken, dan om ze telkens te moeten aanmaken en weggooien.

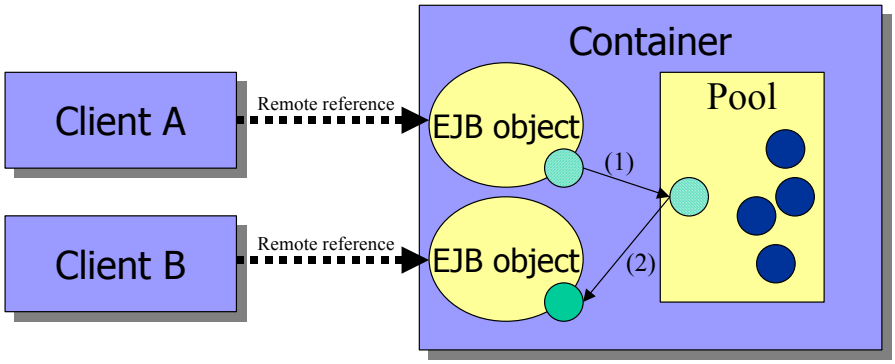


Figuur 5-3: Instance pooling toestanden

Omdat een client nooit direct gebruik maakt van een enterprise bean, maar verbonden is met een EJB-object, hoeft de server niet elke client een aparte instantie te geven. Hij kan uit de voeten met een veel kleinere set beans, door deze achter de schermen over meer clients te verdelen. De container beheert de capaciteit van de bak tijdens het draaien van de applicatie. Tijdens 'drukke tijden' verhoogt hij het aantal instanties en vice versa.
Figuur 5-3 laat de toestanden van de life-cycle van een entity bean zien met betrekking tot instance pooling. Een bean in 'no'-state is nog niet geïnstantieerd. Wanneer de container een bean instantieert maar nog niet aan

[26] JavaBeans is een componentenmodel, bedoeld voor componenten binnen één proces. EJB is een server-side componentenmodel, bedoeld voor gedistribueerde componenten [A3].
[27] een "pool"

een EJB-object koppelt, gaat deze van de ‘no’-state naar de ‘pooled’-state. Als tenslotte de bean wordt geassocieerd met een EJB-object, komt hij in de ‘ready’-state, waar hij klaar is om methode aanroepen te ontvangen.
Ook message-driven beans lenen zich voor instance pooling, omdat ze geen toestand bijhouden. Het type van een message bean wordt bepaald door de berichtbestemming waar ze zich voor ingeschreven hebben. Alle beans met dezelfde bestemming zijn van hetzelfde type. Nu houdt een EJB-container vaak evenveel bakken aan voor het instance poolen als er typen beans zijn. Stuurt een client een bericht naar bestemming A, dan zal de container een message bean uit de bak met type A beans halen. Deze bean ontvangt en verwerkt het bericht.



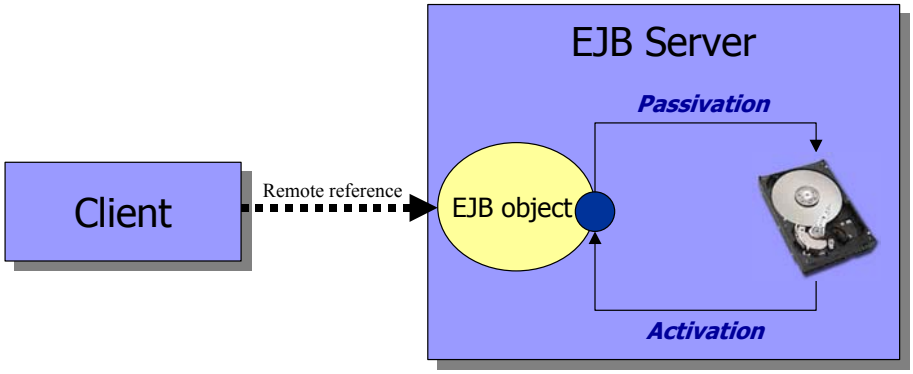
Figuur 5-4: Instance swapping

Stateless session beans geven instance pooling nog een extra dimensie. Omdat stateless session beans geen ‘toestand’ bewaren en elke methode aanroep uitvoeren zonder gebruik te maken van klasse variabelen, kunnen ze diensten verlenen aan een willekeurig EJB-object van hetzelfde type. Dit betekent dat wanneer een client een tijd lang geen methode aanroept van een object, de container de bean die aan dit object toegekend is, weer in de bak kan stoppen. Dit is in Figuur 5-4 aangegeven met (1). Vervolgens kan deze bean worden ingezet voor een aanroep van een andere client, aangegeven in de figuur met (2). Deze techniek heet *instance swapping* en is alleen van toepassing op stateless session beans. Beans die wel een toestand bewaren, zoals entity beans en stateful session beans, kunnen niet aan een willekeurig EJB-object gekoppeld worden van hetzelfde type, omdat deze informatie verloren zou gaan. Omdat de pauzes tussen methode aanroepen van clients vaak langer zijn dan de tijd die nodig is om een aanroep uit te voeren kunnen een paar bean instanties een groot aantal clients bedienen. Instance swapping minimaliseert zo de inactieve periode waarbij een stateless session bean niet productief is, maar wel aan een EJB-object vastzit.

5.3.3 Activation en Passivation

Stateful session beans houden een toestand van de sessie bij, een zogenaamde *conversational state*. Deze toestand moet worden blijven behouden zolang de bean een client bedient. Stateful session beans nemen geen deel aan instance

pooling. In plaats daarvan maken ze gebruik van activation en passivation [A3].
Wanneer een EJB-server geheugen moet vrijmaken, kan hij stateful session beans uit het geheugen verwijderen. De bean wordt losgekoppeld van het EJB-object en de toestand van de bean wordt op disk opgeslagen. Dit proces heet *passivation*. De client merkt hier niets van, omdat de verbinding met het EJB-object gehandhaafd blijft. Het *activation*-mechanisme treedt in werking wanneer een client een methode aanroept van een gepassiveerde bean. De container maakt een nieuwe instantie van een stateful session bean en haalt de toestand van schijf op.



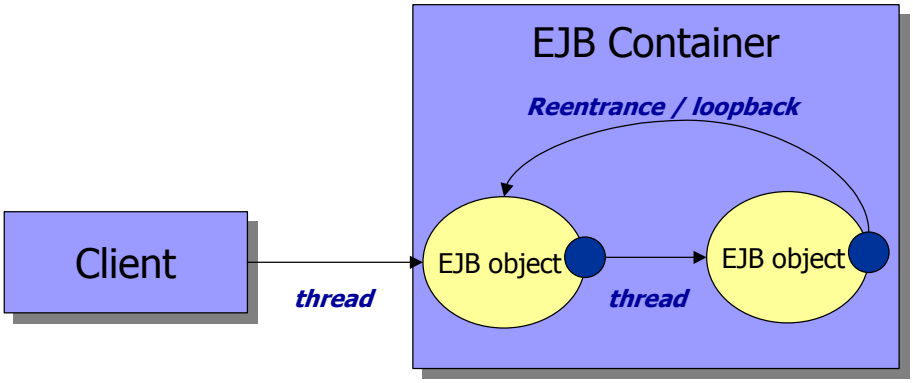
Figuur 5-5: Activation en Passivation van stateful session beans

Omdat de toestand van entity beans direct opgeslagen wordt in de database, hebben ze geen conversational-state die opgeslagen moet worden op schijf. Entity beans kennen wel dezelfde activation-methoden, maar deze hebben te maken met het weg zetten van de entity bean in de instance pool en deze later weer op te halen uit de pool.

5.3.4 Concurrency

Wanneer meerdere client-processen tegelijkertijd gebruik maken van een serverproces of wanneer er meerdere serverprocessen parallel draaien, is er sprake van *concurrency* [A2]. De parallelle uitvoering van processen levert een performancewinst op. Concurrency is een van de primaire diensten van een Component Transaction Monitor. Het heeft binnen EJB, met uitzondering van message-driven beans, te maken met het omgaan met de situatie waarbij meerde clients tegelijkertijd dezelfde bean benaderen [A3].
Allereerst is het goed om te weten dat session beans niet geschikt zijn voor concurrency. Stateful session beans hebben namelijk een toestand die specifiek is voor de client die hem gebruikt en kunnen dus niet gedeeld worden met andere clients. Bij stateless session beans is er geen behoefte aan concurrency, omdat ze geen toestand of data bijhouden die gedeeld moet worden. Entity beans daarentegen, representeren data uit de database die gedeeld wordt en ook tegelijkertijd toegankelijk moet zijn voor meerdere clients.
Data mogen echter niet corrupt raken door het simultaan gebruik ervan. Zo kunnen problemen ontstaan wanneer je entity beans deelt onder clients. De data die door de beans wordt gerepresenteerd, kan corrupt raken doordat verschillende operaties op de data door elkaar heen lopen. EJB heeft een vrij

simpele oplossing voor dit probleem: het is standaard niet toegstaan dat entity bean instanties tegelijkertijd gebruikt worden. Er kunnen meerdere clients verbonden zijn met een EJB-object, maar slechts een *client-thread* kan de bean instantie benaderen. Wanneer een de aanroep van een methode onderdeel is van een langere transactie[28], dan blijft de aanroeper de enige client, tot de transactie voltooid is. De EJB-container zorgt voor het thread-management en verbetert zo de performance van bean instanties. Daarom mogen programmeurs zelf geen thread-safe beans maken. Ook mogen beans zelf geen nieuwe threads creëren. Als dat zou gebeuren, verliest de container de controle over de bean om de primaire diensten uit te kunnen voeren.



Figuur 5-6: reentrance door middel van een loopback

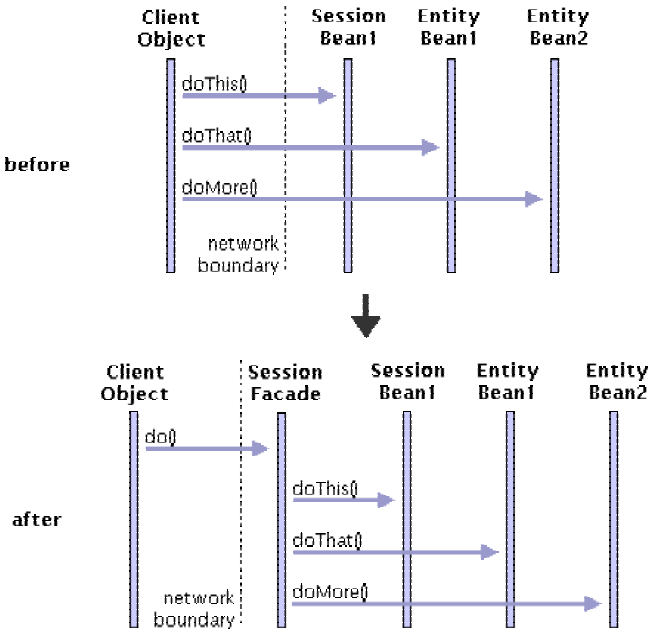
Een ander concept binnen concurrency is *reentrance.* Dit treedt op als een thread opnieuw probeert een bean binnen te komen. Een EJB-object geen onderscheid kan maken tussen reentrant code en multi-threaded gebruik binnen een transactie. Alle clients zijn gelijk voor een bean, of het nu gaat om een remote client of een andere bean. Als je reentrance toestaat, sta je ook toe dat bean instanties tegelijkertijd worden gebruikt en dit is juist wat je wilt voorkomen. Overigens kunnen entity beans reentrant gemaakt worden, maar dit wordt afgeraden in de EJB-specificatie.
Concurrency bij message beans betekent het verwerken van meer dan één bericht op een tijdstip. Veel JMS-clients kunnen tegelijkertijd een bericht sturen naar dezelfde bestemming. De EJB-container zorgt er dan voor dat een gelijk aantal message bean instanties gebruikt wordt om de berichten te kunnen verwerken. Dit wordt *concurrent processing* genoemd.

5.3.5 EJB Design Patterns

Er zijn twee design patterns voor EJB, die de performance ten goede komen door het netwerkverkeer te beperken [A8].
Session Façade groepeert businesslogica aanroepen in een stateless session bean. Figuur 5-7 schets dit concept met behulp van twee sequentiediagrammen. Eerst een diagram zonder, en daarna een met het gebruik van session façade.
Deze implementatievorm heeft meerdere voordelen. Ten eerste verbetert dit de performance, doordat het netwerkverkeer tussen de client en de applicatie vermindert. Zeker in het geval van een webapplicatie is dit een aanzienlijke

[28] Zie paragraaf 6.3.2 voor meer informatie over transacties binnen het J2EE-platform

winst. Omdat alle businesslogica zo in dezelfde laag terechtkomt, is de programmatuur veel schoner en beter onderhoudbaar. Het feit dat alle acties binnen een enkele client aanroep vallen, maakt het makkelijk om het geheel binnen een transactie uit te voeren in de businesslogica-tier.



Figuur 5-7: Sequentie diagram zonder en met session façade

Het *transfer object* patroon definieert een manier om samenhangende data te versturen tussen verschillende tiers, met zo weinig mogelijk netwerkverkeer. In plaats van het aanroepen van losse 'get'-functies maak je een object aan dat alle benodigde data bevat en gebruik je dit object om de data te versturen. Een andere naam voor dit patroon is *value object.*

5.4 EIS tier

5.4.1 Connection pooling

Net zoals je instanties van businessobjecten bij instance pooling hergebruikt, zo kun je ook database-connecties hergebruiken. *Connection pooling* zorgt ervoor dat een applicatie gebruik kan maken van een bak met bestaande datatabase-connectieobjecten. Hierdoor vervallen de dynamische overheadkosten van het openen en sluiten van database connecties. Met de komst van JDBC 2.0 specificatie, zie ook paragraaf 4.4.1, is dit een kwestie van configureren geworden. Een ontwikkelaar hoeft alleen aan te geven dat het om een *pooled datasource* gaat.

5.4.2 JDBC Best Practices

Er zijn een aantal best practices met betrekking tot JDBC, wanneer het gaat om performance [A8]. We zijn vooral geïnteresseerd in de specifieke mogelijkheden binnen J2EE en laten daarom hier de algemene database best

practices, zoals het gebruik van prepared statements, statement pooling en optimististic concurrency, achterwege. We bekijken hier twee best practices.
De eerste gaat over de keuze van de JDBC-driver[29]. De JDBC-drivers zijn onder te verdelen in vier categorieën. Elke driver heeft zijn voor- en nadelen. Tabel 3 geeft een overzicht van deze typen drivers en hun werking [A6, A13].
De eerste best practice op het gebied van JDBC performance luidt: "Probeer type 1 and type 3 drivers te vermijden". Dit zijn zogenaamde *bridging drivers* die meerdere vertaalslagen nodig hebben om met de database te communiceren. Daaruit volgt rechtstreeks dat de performance van deze drivers slechter is dan die van de andere typen. Gebruik alleen zo'n driver wanneer de database geen ander type ondersteunt. De keuze tussen type 2 en 4 hangt af van de applicatieomstandigheden. Gebruik een type 4 driver als portabiliteit belangrijk is en een type 2 driver voor standalone of server applicaties.

Type driver	Werking	Voordelen	Nadelen
1:JDBC/ODBC bridge	Driver maakt gebruik van een ODBC driver voor de connectie met de database.	Enige mogelijkheid voor sommige databases, zoals MS Access.	Slechte performance Beperkte functionaliteit Vereist ODBC driver
2: Partial Java driver (native)	Driver communiceert met de database via een database native client API library.	Goede performance	Afhankelijk van de native library JDBC-driver nodig op elke client machine
3: Pure Java driver (middleware)	Java driver vertaalt JDBC aanroepen naar een generiek netwerkprotocol, waar een stuk middleware weer een andere driver gebruikt om de database te benaderen.	Kan afwijkende databases benaderen met 1 JDBC-driver. Kan dynamisch gedownload worden Kan door een firewall	Matige performance Database specifieke code nodig op het middleware platform
4: Pure Java driver (direct-to-database)	Java driver vertaalt JDBC aanroepen direct naar een database specifiek protocol.	Aardige performance Goede portabiliteit Geen andere driver software nodig Kan dynamisch gedownload worden	Niet geoptimaliseerd voor een server OS Elke afwijkende database vereist een andere driver

Tabel 3: Typen JDBC-drivers

Het tweede advies met betrekking tot JDBC raadt aan om de eigenschappen van de database-connectie af te stemmen op de applicatie. *Connection properties* zijn opties die niet aanbieder specifiek zijn, maar die wel gebruikt kunnen worden binnen de JDBC API om de database functionaliteit van een specifieke applicatie te optimaliseren. Ze bevatten onder andere informatie voor de driver over de grootte van de resultaatsets, de sorteermethoden en time-out instellingen.

[29] Paragraaf 4.4.1 bespreekt de Java Database Connectivity.

6 Betrouwbaarheid

6.1 Inleiding

Computersystemen kunnen storingen vertonen. Er kunnen fouten optreden in de hardware of software, waardoor applicaties vastlopen of verkeerde resultaten produceren. Een *betrouwbaar systeem* heeft slechts een kleine kans dat het afwijkt van het geplande gedrag waarvoor het ontworpen is. Dit houdt ook het stoppen van een systeem in, zonder dat het daadwerkelijk incorrecte acties uitgevoerd heeft [A2]. Een betrouwbaar systeem zal dus moeten kunnen omgaan met storingen en fouten.
In de 24-uurs economie moet het mogelijk zijn dat een systeem dag en nacht gebruikt kan worden, zeven dagen per week. Een klant kan bij een on-line boekenwinkel dan ook 's-nachts een boek bestellen. Wanneer een computersysteem een vitale rol vervult in een kritiek bedrijfsproces, kan het uitvallen van zo'n systeem ernstige gevolgen hebben. Hier zullen systemen een up-time moeten kunnen garanderen. De beschikbaarheid of *availability* van een systeem is een maatstaf die aangeeft welk deel van een periode het systeem beschikbaar is voor gebruik.

6.1.1 Criteria

We streven naar een *fouttolerant systeem*. Dit is een systeem dat fouten kan detecteren en "netjes" en voorspelbaar afbreekt, of dat fouten kan maskeren zodat gebruikers van het systeem niets merken van de fout [A2]. Het gaat hierbij niet om preventieve maatregelen ter voorkoming van fouten.
Gegeven dat applicaties essentieel kunnen zijn voor kritieke bedrijfsprocessen, moeten ze een hoog beschikbaarheidniveau kunnen garanderen. Dit zijn systemen met een *high availability*.

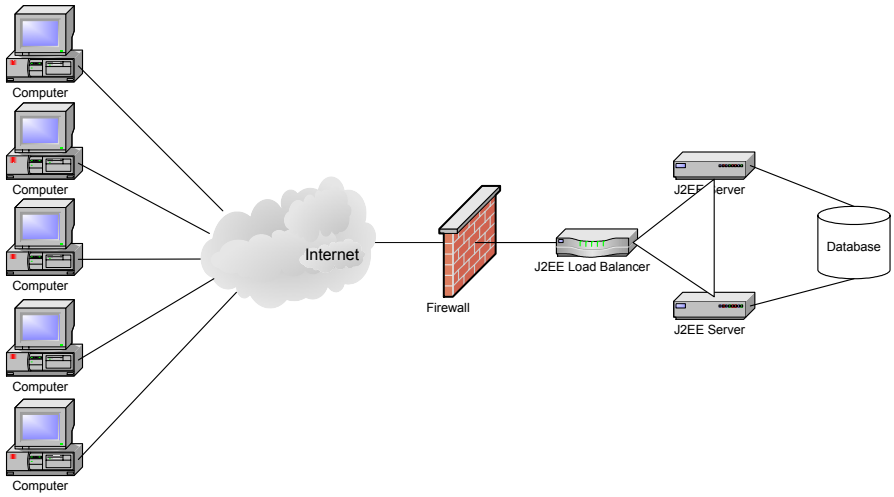
6.1.2 Betrouwbaarheid binnen J2EE

Dit hoofdstuk maakt onderscheid tussen twee aspecten van betrouwbaarheid. Enerzijds de eerder genoemde beschikbaarheid van het systeem, anderzijds de mogelijkheid om te herstellen van opgetreden fouten.
De beschikbaarheid van een systeem kan vergroot worden door hardware redundancy. *Hardware redundancy* is het dupliceren van hardware componenten in een systeem. Wanneer een hardwarecomponent uitvalt, neemt een andere component het werk over. Het J2EE-model ondersteunt high availability door het gebruik van clusters.
De status van permanente data in een systeem moet na het optreden van een fout hersteld of teruggezet kunnen worden naar een consistente toestand. Deze *software recovery* in het J2EE-model vindt plaats door middel van transacties en persistentiemechanismen.

6.2 Beschikbaarheid

6.2.1 Clustering

Gedistribueerde systemen kunnen door hun architectuur meer weerstand bieden tegen hardware fouten, dan niet-gedistribueerde systemen. Het uitvallen van een client hoeft geen gevolgen te hebben voor andere gebruikers. Ook het uitvallen van een server hoeft geen belangrijke impact te hebben op de diensten die het systeem aanbiedt aan gebruikers, als andere servers de taken kunnen overnemen [A2].
Een *cluster* is een groep webservers of applicatieservers waarop een J2EE-applicatie draait, en die voor de buitenwereld één entiteit lijken. Voor de schaalbaarheid[30] zijn extra machines nodig om de capaciteit te vergoten. Voor availability moet elke component van de cluster redundant zijn, zodat bij een serverfout een andere server de taak overneemt. *Shared nothing* clusters hebben onafhankelijke servers, die elk hun eigen opslagruimte hebben. Dit vereist veel onderhoud. Bij een *Shared disk* cluster maken de applicatieservers gebruik van dezelfde disk om applicaties te laden. Deze vorm vereist minder onderhoud, maar heeft extra technieken nodig om geen *single point of failure* te hebben [B19].



Figuur 6-1: Load-Balanced Cluster van J2EE-servers

De J2EE-specificatie verplicht het ondersteunen van clusters niet, maar de meeste J2EE-servers bieden clustering aan op twee niveaus. *HttpSession Clustering* biedt web clients de mogelijkheid om de sessiestatus van een secundaire J2EE-server te halen wanneer een fout optreedt. Dit kan door het opslaan van sessiedata in een database of op schijf. Het clusteren van EJB componenten, *EJB Clustering,* gebeurt met behulp van *replica-aware stubs.* Deze stubs weten van het bestaan van alle servers in de cluster af en kunnen een load-balancing algoritme gebruiken om te bepalen waar de objecten vandaan gehaald moeten worden [B20].

[30] Zie paragraaf 5.2.3 over load-balancing

6.3 Fouttolerantie

6.3.1 Persistentie

Entity beans representeren het gedrag en de data van een entiteit uit de echte wereld, zoals een persoon, een plaats of een ding. De status van een entity bean wordt permanent bijgehouden in een database. Dit maakt entiteiten duurzaam en betrouwbaar. De informatie zal niet verloren gaan als gevolg van een storing in het systeem. *Persistentie* is het coördineren van de data van een bean instantie met de database. Dit gebeurt door synchronisatie. [A3].
Persistentie kan automatisch en handmatig plaatsvinden. Bij *Container managed persistence (CMP)* is de EJB-container verantwoordelijk voor de persistentie. Hij zorgt automatisch voor het toevoegen, updaten en verwijderen van de data, die geassocieerd is met entiteiten in de database. De handmatige variant, *bean managed persistence (BMP)*, laat de verantwoordelijkheid van persistentie over aan de bean zelf. De container laat de bean weten wanneer het veilig is om database acties uit te voeren, maar biedt verder geen hulp. De ontwikkelaar moet zelf de benodigde JDBC en SQL code schrijven om de persistentie te waarborgen.
Aanbieders mogen zelf het mechanisme kiezen om container managed persistentie te implementeren, mits ze de specificatie volgen. Hier volgen de meest gebruikte persistentiemechanismen.

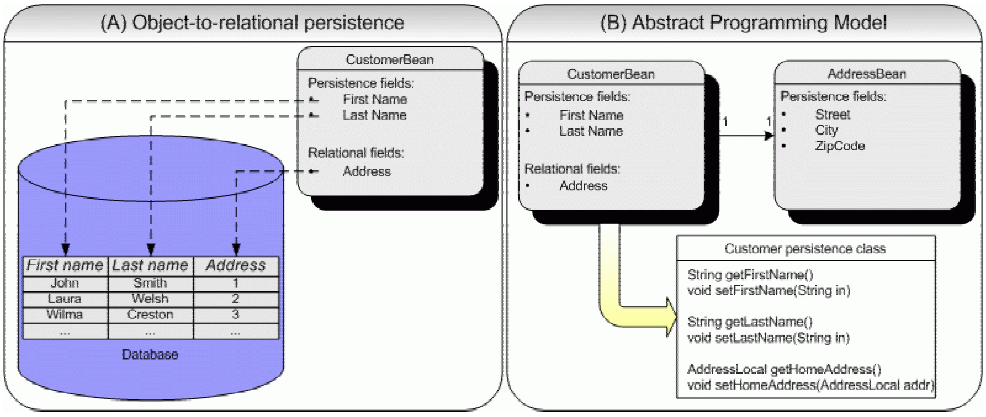
- ***Object-naar-relationeel persistentie***
  Object-naar-relationeel persistentie is waarschijnlijk het meest gebruikte persistentiemechanisme vandaag de dag. Deze vorm vertaalt de entity bean status naar relationele database tabellen en kolommen. Figuur 6-2(A) laat een voorbeeld zien van een customer bean met drie velden. Het komt regelmatig voor dat bepaalde bean typen vertaald worden naar meerdere tabellen. Goede EJB systemen bieden wizards of administratieve interfaces aan om deze vertaling vast te leggen tijdens de deployment van een applicatie.

- ***Object database persistentie***
  Object georiënteerde databases zijn een veel beter opslagmedium voor componenten uit een OO-taal zoals Java. Ze bieden een veel logischere mapping tussen entity beans en de database. Helaas zijn OO-databases nog altijd vrij nieuw voor businesssystemen en daarom niet zo algemeen geaccepteerd als relationele databases. Dit maakt dat ze ook niet zo gestandaardiseerd zijn als relationele databases, wat het moeilijker maakt om over te stappen van een database-systeem naar een ander.

- ***Legacy persistentie***
  Legacy systemen zijn systemen die gebaseerd zijn op mainframe applicaties of niet relationele databases. Het EJB-model wordt vaak gebruikt om een object georiënteerde laag om legacy systemen heen te bouwen. Container managed persistentie vereist hier een speciale EJB-container, die speciaal

ontworpen is voor databenadering van een legacy systeem[31]. Vaak bestaat zo'n container niet en is bean managed persistentie de enige mogelijkheid.

Container managed persistence werkt met een abstract programmeermodel en een abstract persistentieschema. Het *abstract programmeermodel* maakt een vertaling van een OO-omgeving naar een database mogelijk zonder dat dit expliciet geprogrammeerd hoeft te worden. De ontwikkelaar definieert een abstracte bean met *virtuele persitentievelden* en *relatievelden.* Ze zijn virtueel omdat het geen variabelen of velden zijn, maar accessor-methoden. De EJB-container tools zorgen voor de implementatie van een abstracte entity tijdens de deploymentfase. Dit is de persistentieklasse, te zien in Figuur 6-2(B).Dit bevat onder andere de benodigde logica om verbinding te maken met de database.



Figuur 6-2: Persistentie van entity beans

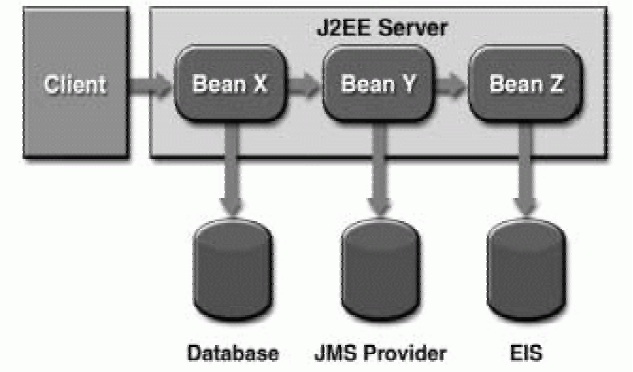
Binnen de EJB 2.0 specificatie kunnen relaties tussen beans bestaan, net zoals tussen tabellen in een database. Naast de relatievelden van het abstract programmeermodel, moet een ontwikkelaar de cardinaliteit en de richting van de relaties definiëren. Deze informatie staat in de XML deployment descriptor en vormt het *abstract persistentieschema.* In het voorbeeld in Figuur 6-2(B) gaat het om een one-to-one unidirectionele relatie.
Het gebruik van entity beans in plaats van directe database benadering heeft een aantal voordelen. Het belangrijkste is dat EJB's de data in een object gieten, wat voor programmeurs een simpeler mechanisme vormt om data te benaderen en te wijzigen. Ook bevordert het hergebruik van software, doordat de definitie van een entity beans consistent gebruikt kan worden binnen een systeem.

6.3.2 Transactiemanagement

Een *transactie* is een serie taken[32] die gezamenlijk uitgevoerd moet worden. In de zakelijke wereld houdt een transactie vaak een uitwisseling tussen twee

[31] Aanbieders kunnen mapping tools leveren die het mogelijk maken om beans te mappen naar IMS, CICS, b-trieve of een andere legacy applicatie.

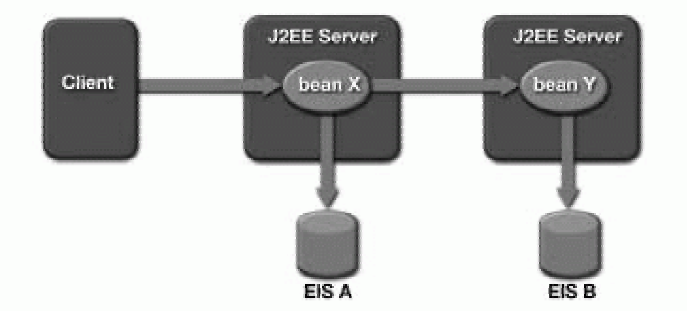
partijen in. Transacties zijn atomair, wat betekent dat alle taken binnen een transactie uitgevoerd moeten worden, of dat geen van de taken uitgevoerd wordt. Zo wordt de integriteit van de onderliggende data gewaarborgd. In softwaresystemen hebben de taken betrekking op het gebruiken van gedeelde hulpbronnen, meestal databases [A3].
Gedistribueerde transacties zijn transacties waarbij meerdere, onafhankelijke, transactiesystemen, of *resource managers,* samenwerken. De associatie tussen een transactie en een applicatiecomponent of een resource manager wordt *transactiecontext genoemd.* Gedistribueerde transacties zijn complexer dan niet-gedistribueerde transacties. Dit wordt veroorzaakt door latentietijden[33], server- en netwerkstoringen, en interoperability problemen. Doordat resource managers zelfstandig werken en niet van elkaars bestaan afweten, vereisen gedistribueerde transacties een externe coördinatie. Deze coördinatie kan plaatsvinden in de applicatie zelf. Dit heeft een aantal nadelen, want het handmatig coördineren is complex, niet herbruikbaar en foutgevoelig. Een programmeur wordt dan belast met zaken als het beheer van data toegang, gesynchroniseerde updates, foutherstel en multi-user problemen [A7].
Een makkelijkere oplossing voor het coördineren van gedistribueerde transacties, is het introduceren van een externe verantwoordelijke, de *transactiemanager*. Deze treedt op als mediator tussen applicaties en de hulpbronnen. De EJB-architectuur biedt *transactiemanagement* als een standaard dienst aan. Bij *container managed transactiedemarcatie* is de EJB-container verantwoordelijk voor het definiëren van de transactiegrenzen binnen een EJB-object (*transactie demarcatie*), en voor het informeren van EJB-objecten over transactiegrenzen van transacties die door de client aangevraagd zijn. Ook regelt de container de commit en roll-back beslissingen. De assembler kan met behulp van *transactieattributen* de transactiesemantiek instellen. Moet een specifieke bean bijvoorbeeld altijd een eigen transactie starten, of mag hij nooit binnen een transactie aangeroepen worden. EJB ondersteunt een *flat transaction model*, wat betekent dat transacties niet "genest" kunnen worden. Sinds versie 1.3 is een J2EE-product verplicht om binnen een transactie de toegang te ondersteunen naar meerdere resource managers. Zo moet binnen één transactie in ieder geval de toegang tot één JDBC-database, één JMS-provider en meerdere enterprise informatiesystemen (door middel van connectors) ondersteund kunnen worden (Figuur 6-3).



Figuur 6-3: meerdere resource managers binnen een transactie

[32] Ook wel *unit of work* genoemd.
[33] Tijdsvetragingen/overhead door netwerkcommunicatie

Net zoals bij persistentie is er naast de hierboven beschreven container managed transactiedemarcatie, een bean managed variant. Bij bean managed transacties onderbreekt de container altijd een eventuele client transactie voor het aanroepen van een EJB-methode. Na het voltooien van de bean transactie, vervolgt de container de client transactie weer. De enterprise bean moet zelf zijn transactiegrenzen definiëren. Deze handmatige variant is bijna nooit nodig, tenzij je meerdere transactie wilt definiëren binnen een methode.
Een ontwikkelaar maakt direct gebruik van de Java Transactie API bij bean managed transactiedemarcatie. De *Java Transactie API (JTA)* is een enterprise API voor het beheren van gedistribueerde transacties. De EJB-server maakt in de container managed instelling ook gebruik van de JTA. Overigens mogen entity beans geen gebruik maken van bean managed transacties, omdat dan de persistentiemechanismen niet goed meer kunnen functioneren [A13].
De Java Transactie API maakt gebruik van het *two-phase commit protocol (2PC)* voor gedistribueerde transacties. In de eerste fase vraagt de transactiemanager alle resource managers om te committen. Wanneer alle resource managers akkoord gaan, geeft de transactiemanager in fase twee de resource managers de opdracht om daadwerkelijk een *commit* uit te voeren. Als een of meer resource managers niet akkoord gaat in fase een, geeft de transactiemanager het bevel tot een *roll-back.*



Figuur 6-4: Een transactie over meerdere applicatieservers

J2EE-producten kunnen ook transacties distribueren over meerdere applicatieservers. Een voorbeeld is te zien in Figuur 6-4. Hierbij werken de twee J2EE-servers samen om de transactiecontext te propageren van x naar y. Dit is transparant voor de applicatie [A7].

7 Beveiliging

7.1 Inleiding

Het beveiligen van gegevens binnen een organisatie is niets nieuws. Informatiebeveiliging moet er onder andere voor zorgen dat 'gevoelige' gegevens niet bij onbevoegden terecht kunnen komen. Voordat bedrijven de computer grootschalig gingen gebruiken voor dataverwerking, bestond deze beveiliging voornamelijk uit fysieke en administratieve middelen [A10]. Met de komst van de computer kwam ook de behoefte aan softwarematige tools om databestanden te beschermen. Deze oudere centrale systemen bieden door hun isolatie nog een redelijke bescherming tegen aanvallen van buiten af [A2] De introductie van gedistribueerde systemen neemt deze isolatie weg en zorgt daarmee voor nieuwe bedreigingen. Computers verbonden met bijvoorbeeld het Internet, zijn kwetsbaar voor aanvallen van buitenaf.

7.1.1 Criteria

Computersystemen dienen beveiligd te worden tegen onrechtmatig gebruik. Om dit goed te kunnen doen, moeten we weten met welke bedreigingen (*security threats)* we te maken hebben en op welke manier de concrete aanvallen plaats kunnen vinden. De bedreigingen zijn onder te verdelen in vier hoofdcategorieën [A2]:

- ***Lekkage***
  Het verkrijgen van informatie door onbevoegde gebruikers. Aanvallen hebben een passieve vorm, zoals het afluisteren van communicatielijnen, waarbij informatie wordt opgedaan over de inhoud van de boodschap of over het patroon van de communicatie.

- ***Vervalsing***
  Het onrechtmatig veranderen van informatie Mogelijke aanvallen zijn *maskerades, replays* en *message modifications.* Bij een maskerade doet een entiteit zich voor als een andere entiteit. Een entiteit kan ook een bericht afvangen en opnieuw versturen om een ongeautoriseerd effect te krijgen. Het vertragen van een bericht, het aanpassen van de inhoud ervan of het veranderen van de volgorde van berichten valt onder message modification.

- ***Onrechtmatig gebruik van resources***
  Het gebruik van faciliteiten zonder autorisatie. Dit gebeurt door middel van infiltratie. Personen maar ook programma's kunnen een systeem infiltreren. Voorbeelden van dergelijke programma's zijn virussen, wormen en Trojaanse paarden.

- ***Vandalisme***
  Ongepast gebruik van het systeem, zonder dat het voordelen biedt aan de gebruiker. Dit heet ook wel *denial of service*, waarbij een entiteit (een deel

van het) systeem buiten werking stelt of vertraagt. Concrete voorbeelden zijn het afvangen van berichten of het overbelasten van een netwerk.

Om een systeem te beschermen tegen deze bedreigingen moeten gedragslijnen, *security policies,* worden opgesteld. Deze gedragslijnen bepalen de benodigde niveaus van beveiliging voor de activiteiten binnen een systeem. Het toepassen beveiligingsmaatregelen (*security services)* maakt het naleven van de opgestelde gedragslijnen mogelijk. De gebruikte technieken voor het realiseren van de beveiligingsmaatregelen heten beveiligingsmechanismen. De samenhang kan als volgt gedefinieerd worden: beveiligingsmaatregelen implementeren gedragslijnen en worden geïmplementeerd door beveiligingsmechanismen. [A10]

7.1.2 Beveiliging binnen J2EE

Binnen het J2EE-applicatiemodel kunnen een aantal beveiligingsmechanismen toegepast worden [A7, B7]. Het is vooral belangrijk om te weten wat géén doelstellingen zijn van de J2EE-specificatie met betrekking tot beveiliging. Hier volgen een paar hoofdpunten:

- Beveiligingsmechanismen stellen geen gedragslijnen vast. Ze maken alleen het naleven ervan mogelijk. De gedragsregels worden tijdens de deployment fase gespecificeerd met behulp van Deployment Descriptors, bij het bepalen van de security-rollen.
- De specificatie schrijft geen verplichte beveiligingstechnologie voor, zoals bijvoorbeeld een encryptie-algoritme. Dit wordt overgelaten aan de leverancier.
- Het is niet de bedoeling om de bestaande beveiligingsinfrastructuur van een organisatie te vervangen, maar om deze uit te breiden.

We zullen de beveiligingsmechanismen die het J2EE-platform aanbiedt wat meer in detail doornemen. Dit doen we aan de hand van de beveiligingsmaatregelen authenticatie, autorisatie, data confidentiality, data integrity en accountability[34] [A10, C8]. Per maatregel bespreken we de bijbehorende mechanismen.

7.2 J2EE Beveiligingsmaatregelen

7.2.1 Authenticatie

Alleen systeemgerechtigde personen mogen van het systeem gebruik maken. Het systeem moet weten met wie het te maken heeft en moet daarom de identiteit van een gebruiker kunnen controleren [A7]. Dit proces heet *authenticatie.* De gebruiker bewijst aan het systeem dat hij handelt namens een systeemgerechtigde identiteit. Het meest gebruikte mechanisme is een login-scherm waar een gebruikersnaam en een wachtwoord ingevuld moeten worden. Bij wederzijdse authenticatie moet ook het systeem haar ‘echtheid’ bewijzen,

[34] Availability wordt ook vaak genoemd als een beveiligingsmaatregel. Echter, in dit document beschrijven we het onder ‘betrouwbaarheid’ (paragraaf 6.2).

waarbij gebruik gemaakt wordt van digitale handtekeningen en certificaten [C4].

De J2EE-specificatie maakt onderscheid tussen authenticatie voor web clients en applicatie clients. Bij applicatie clients is de container verantwoordelijk voor de authenticatie, terwijl bij web clients drie voorgedefinieerde authenticatiemechanismen ondersteund moeten worden:

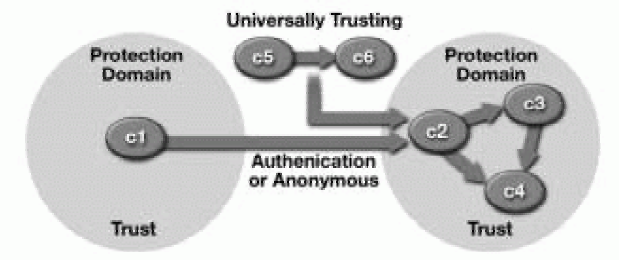
- ***HTTP Basic Authenticatie***
  Dit mechanisme wordt ondersteund door het HTTP-protocol, waarbij de web-client de gebruikersnaam en het wachtwoord van de gebruiker verkrijgt via een standaard dialoog van de webbrowser en dit doorstuurt naar de webserver. Echter, wachtwoorden worden ongecodeerd verstuurd en er vindt geen authenticatie van de server plaats. Deze problemen kunnen verholpen worden door extra beveiligingsmaatregelen toe te passen. Dit kan in de transportlaag (bijv. HTTPS), of in de netwerklaag (bijv. VPN). Er is dan sprake van een hybride authenticatie.

- ***Form-based login***
  Wanneer ontwikkelaars hun eigen user interface willen gebruiken binnen een webbrowser voor de authenticatie dan kunnen ze gebruik maken van het form-based mechanisme. Het lijdt aan dezelfde tekortkomingen als het HTTP-basic mechanisme, maar ook hier kan hybride authenticatie plaatsvinden.

- ***HTTPS mutual authentication***
  Hier stellen zowel de gebruiker en de server elkaars identiteit vast aan de hand van *public key certificates (PKC)*. De benodigde communicatie voor de wederzijdse authenticatie is beveiligd door SSL 3.0.

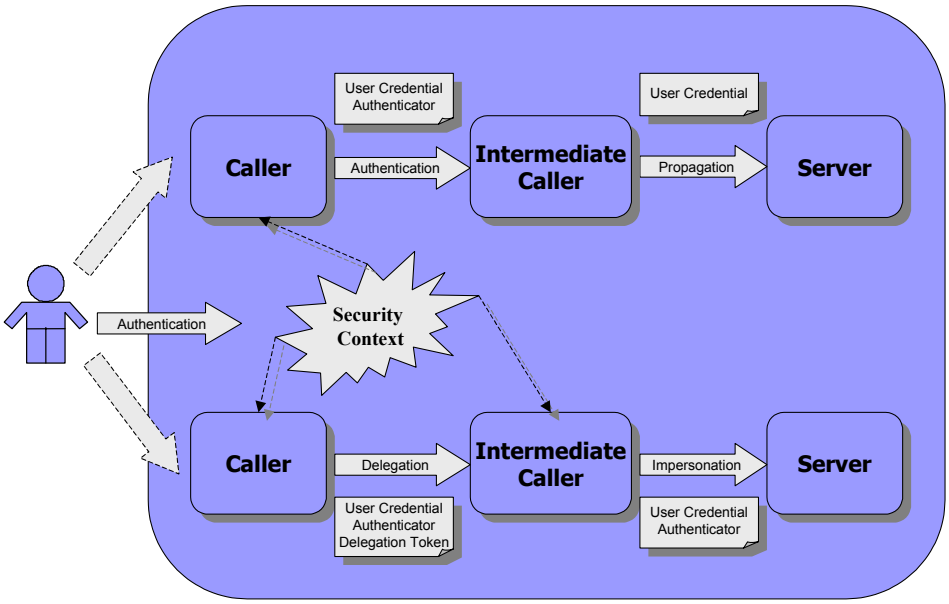
Naast de genoemde verplichte authenticatiemechanismen is er ook nog HTTP-Digest authenticatie. Hierbij maakt de web-client een *digest* aan, middels een hash-algoritme op basis van een HTTP request en het wachtwoord, en stuurt dit naar de webserver. Het digest is veel kleiner dan het HTTP request en bovendien wordt het wachtwoord niet verstuurd. Deze techniek wordt echter niet op grote schaal ondersteund door web-browsers en is daarom niet verplicht in de J2EE-specificatie.
Nu kun je je afvragen wanneer iemand zich moet identificeren. Voor onderdelen van een applicatie die toegankelijk zijn voor iedereen is authenticatie een onnodige overhead. Denk bijvoorbeeld aan publiekelijke HTML-documenten of plaatjes. Het is wenselijk om pas authenticatie plaats te laten vinden wanneer iemand een beschermd onderdeel probeert te benaderen. Dit heet *lazy authentication.* Tot die tijd blijft de gebruiker anoniem voor het systeem. J2EE web clients zijn verplicht lazy authentication te ondersteunen.
Het zou onhandig zijn als een gebruiker zich bij elke actie opnieuw moet identificeren. Er is behoefte aan het bijhouden van de identiteit van de gebruiker tijdens diens sessie, ook wanneer er een stateless protocol gebruikt wordt, zoals HTTP. Een J2EE-server maakt gebruik van *credentials*. Na het controleren van de identiteit maakt het systeem een credential aan waarin de beveiligingsinstellingen (*security attributes)* van de gebruiker staan. Naast het

behouden van identiteitsinformatie moeten ook grenzen gedefinieerd worden waarbinnen entiteiten elkaar vertrouwen. Met vertrouwen wordt bedoeld dat deze entiteiten niet gauthenticeerd hoeven worden naar elkaar toe. Zo'n gebied heet een *protection domain*, of *realm*. Authenticatie is dan alleen nodig bij interacties die de grens van een protection domain overschrijden. In de J2EE-architectuur vormen containers de authenticatiegrens tussen de componenten binnen de container en externe clients.



Figuur 7-1: protection domains

Figuur 7-2 laat een mogelijk authenticatie scenario zien. De gebruiker roept een component aan binnen het systeem. Aangezien een gebruiker een externe client is voor een container, zal deze zich moeten identificeren. Nu zijn er ook indirecte vormen van authenticatie mogelijk. Een *intermediate caller* is een tussenobject wat aangeroepen wordt en zelf weer een ander component aanroept. Zo'n intermediate caller heeft qua identiteit verschillende mogelijkheden bij het aanroepen van een ander component.



Figuur 7-2: Authenticatie scenario

- ***Propagation***
  Het tussencomponent propageert de identiteit van de oorspronkelijke aanroeper middels een credential. Dit is niet controleerbaar voor het server-

object en deze zal dan ook alleen de aanroep honoreren als het tussencomponent zich in hetzelfde protection domain bevindt.

- ***Delegation***
  Bij delegatie geeft de intermediate caller het aangeroepen server-component toegang tot zijn security context, zodat deze zelf de identiteit van de aanroeper kan controleren en zich kan verpersoonlijken met de oorspronkelijke aanroeper. Zie ook impersonation.

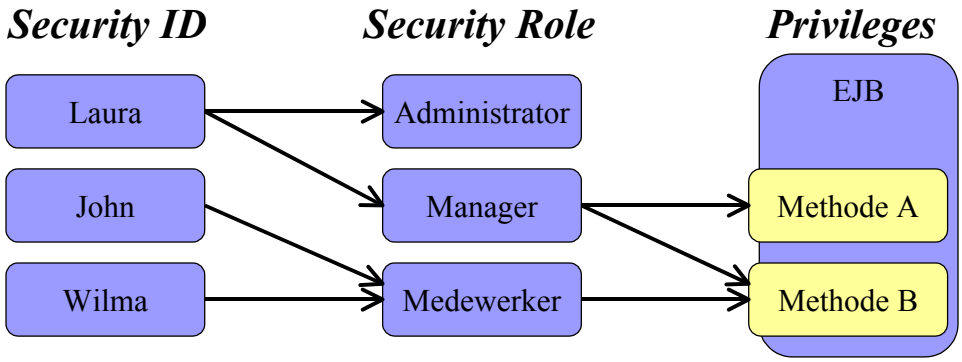
- ***Impersonation***
  Impersonation wil zeggen dat een tussenobject zich mag voordoen als de oorspronkelijke aanroeper. Hiervoor moet de aanroeper het tussenobject vertrouwen.

Het is denkbaar dat een organisatie meerdere applicaties heeft draaien binnen de J2EE-architectuur. Door het configureren van de protection domains over de containers heen en door het gebruik van de intermediate callers kan het model een *single sign-on* implementeren. Een gebruiker hoeft dan slechts een keer in te loggen en kan zo gebruik maken van alle applicaties waartoe hij gerechtigd is.
Aan de backend-kant, oftewel de EIS-tier zijn er twee opties mogelijk met betrekking tot beveiliging. De *container managed sign-on* variant belast de J2EE-container met de authenticatie voor EIS-resources. Bij de *component managed sign-on* aanpak geeft het applicatiecomponent de gebruikersgegevens door en zorgt het EIS zelf voor de authenticatie.

7.2.2 Autorisatie

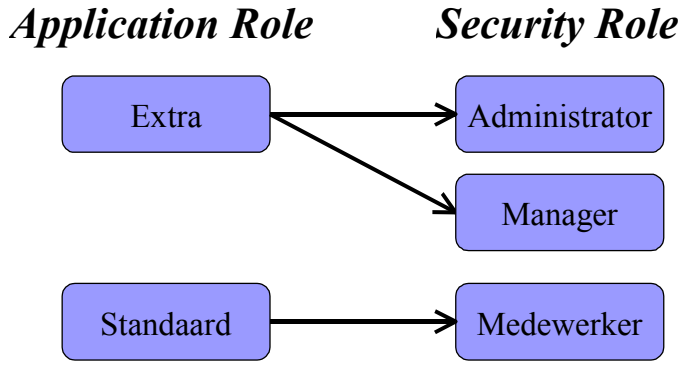
Binnen de geldige gebruikersgroep kunnen verschillende bevoegdheden bestaan. Met andere woorden: wie mag wat wel en wat niet in het systeem doen. Authenticatie bepaalt alleen of iemand gebruik mag maken van het systeem. *Autorisatie* beheert de toegang tot beschermde resources en verfijnt het toegangsmodel van authenticatie.
Binnen een J2EE-applicatie gaat dit door middel van rollen. Een assembler definieert de *security roles* [B3]. Dit zijn abstracte gebruikerscategorieën, zoals “klant”, “manager” of “medewerker”. De toegang tot web resources en Enterprise JavaBeans methoden wordt door de container bepaald op basis van deze beveiligingsrollen. Anders gezegd: een rol draagt bepaalde privileges met zich mee. Containers in een J2EE omgeving ondersteunen twee manieren van beveiliging binnen het autorisatiemodel: declaratieve beveiliging en geprogrammeerde beveiliging [A7, B7, B3].
Declaratieve beveiliging houdt in dat de beveiligingsstructuur van een applicatie buiten de applicatie zelf wordt gedefinieerd. Dit omvat het definiëren beveiligingsrollen, het autorisatiemodel en de authenticatie vereisten. J2EE hanteert hiervoor de deployment descriptor, zoals beschreven in hoofdstuk 3. Hier dient de deployment descriptor als een contract tussen een assembler en een deployer. Zoals gezegd legt de assembler de abstracte beveiligingsrollen vast in de deployment descriptor. Dit gebeurt vanuit het oogpunt van de applicatie. Voor web resources geeft de assembler in de deployment descriptor aan welke rollen toegang hebben tot een bepaalde URL. Voor publiekelijke

web resources worden geen restricties opgenomen in de deployment descriptor. Bij Enterprise JavaBeans moet de toegang op methode niveau gespecificeerd worden. Hier geldt dat anonieme gebruikers een standaard identiteit moeten krijgen om publieke methoden aan te roepen.
Een deployer vertaalt deze abstracte rollen naar *security identities*. Dit zijn concrete gebruikers (*principals)* of gebruikersgroepen binnen een specifieke operationele omgeving. Deze zogenaamde *role mapping* zorgt ervoor dat de vertaling naar werkelijke gebruikers pas plaatsvindt in de deploymentfase en niet in de ontwikkelfase. Dit vergroot de portabiliteit van een J2EE-applicatie. Merk op dat deze mapping niet perse 1-op-1 hoeft te zijn. Een identiteit kan bijvoorbeeld gekoppeld worden aan een of meer beveiligingsrollen en vice versa.



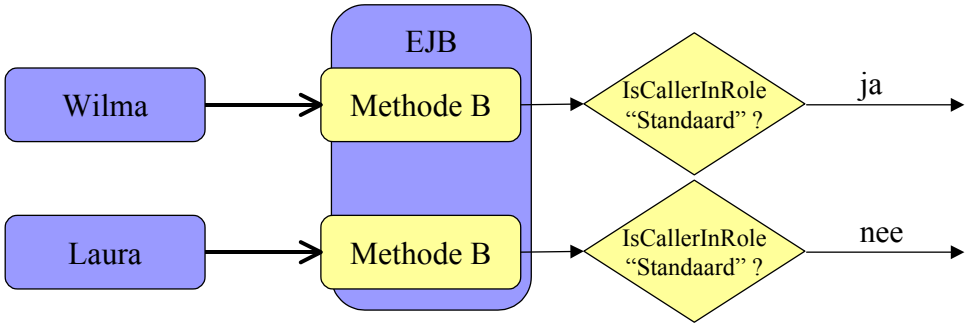
Figuur 7-3: Role mapping voorbeeld

In Figuur 7-3 is een voorbeeld te zien van zo'n role mapping binnen de applicatie. Een gebruiker maakt zijn identiteit bekend aan het systeem door middel van authenticatie. De container maakt vervolgens de vertaling naar een of meerdere beveiligingsrollen. Zo krijgt Laura de rol "Manager" en "Administrator", maar John en Wilma de rol "Medewerker". De figuur laat zien hoe de rol samenhangt met de privileges van EJB methoden. Een "Manager" mag methode A en B aanroepen, terwijl een "Medeweker" alleen toegang heeft tot methode B.



Figuur 7-4: mapping applicatierol naar beveiligingsrol

Naast declaratieve beveiliging ondersteunen J2EE-containers ook geprogrammeerde beveiliging. *Geprogrammeerde beveiliging* zit in de applicatie "gebakken" en zorgt ervoor dat applicaties zelf beveiligingsbeslissingen maken. Deze vorm van beveiliging is nuttig wanneer alleen declaratieve beveiliging niet voldoende is. Voorbeelden zijn het maken van autorisatiebeslissingen op basis van het tijdstip, de aanroepparameters, of de status van een component. Het is een verfijning op het declaratieve autorisatiemodel [A7].
De naam geeft al aan dat deze manier van beveiliging tijdens het ontwikkelproces ingebouwd moet worden. Dit in tegenstelling tot het declaratieve autorisatiemodel, dat pas ingevuld wordt tijdens de deploymentfase van de applicatie. Ontwikkelaars kunnen in hun Enterprise JavaBeans of Webcomponenten ook rollen definiëren. Dit zijn zogenaamde *application roles* of *privilege names*. Ze worden net zoals security identiteiten gekoppeld aan beveiligingsrollen. Een voorbeeld is te zien in Figuur 7-4.



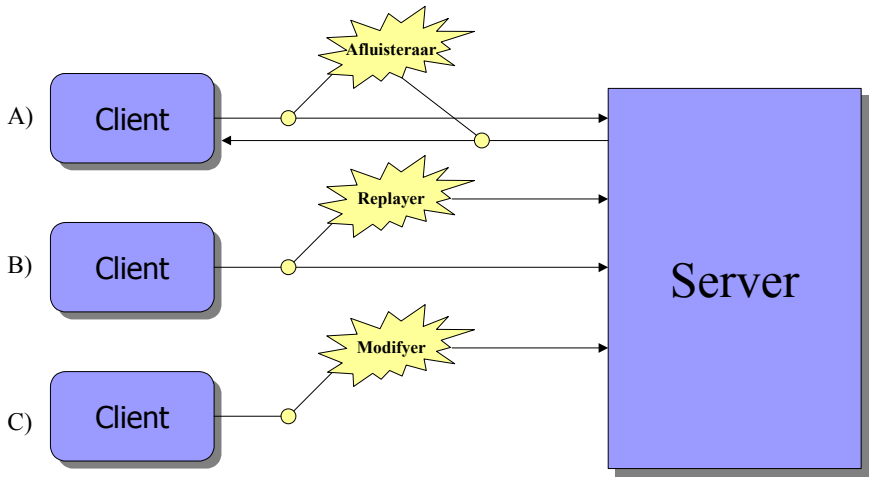
Figuur 7-5: Voorbeeld geprogrammeerde beveiliging

Enterprise JavaBeans en Servlets zijn verplicht om een tweetal methoden aan te bieden voor deze vorm van beveiliging. Een daarvan is de methode "IsCallerInRole" voor Enterprise JavaBeans. Deze methode krijgt een applicatierol mee als parameter en controleert of de aanroeper inderdaad binnen die rol valt. Voortbordurend op het voorbeeld in Figuur 7-3, laat Figuur 7-5 zien wat er gebeurt als Wilma en Laura methode B aanroepen. Methode B controleert of de aanroeper in de applicatierol "Standaard" valt. Wilma vervult een beveiligingsrol, die van "Medewerker". Zij valt daarmee ook in de "Standaard" rol, aangezien deze wordt vertaald naar "Medewerker". Echter, Laura vervult twee beveiligingsrollen, die van "Administrator" en "Manager", maar geen van beide is gekoppeld aan de applicatierol "Standaard".
Met de aanroeper wordt de directe aanroeper bedoeld. Het gebruik van intermediate callers, zoals beschreven in de paragraaf "authenticatie", heeft invloed op zowel het declaratieve als het geprogrammeerde autorisatiemodel. Wanneer een Enterprise JavaBean wordt aangeroepen door een intermediate caller die niet de identiteit van de gebruiker aanneemt, maar draait onder een zelfgekozen beveiligingsrol, gelden de privileges van de caller in plaats van de gebruiker.
Het onderscheid tussen declaratieve en geprogrammeerde beveiliging heeft een aantal voordelen. Tijdens de ontwikkeling kunnen ontwikkelaars meer flexibele functionaliteit inbouwen aan de hand van applicatierollen, terwijl het externe beleid pas wordt vastgesteld nadat de applicatie geschreven is. Zo hoeft

een deployer ook niet alle details van het interne beveiligingsbeleid te weten. Overigens kan een deployer hier wel invloed op uitoefenen, door de vertaling van applicatierol naar beveiligingsrol te veranderen. Ontwikkelaars hoeven van tevoren geen rekening te houden met een specifieke operationele omgeving van de applicatie.

7.2.3 Data Confidentiality

Communicatielijnen tussen client en server zijn vaak het doelwit van aanvallen. Onbevoegden kunnen deze lijnen 'aftappen' om zo berichten over deze lijnen te ontvangen, en verkeerde of gewijzigde berichten door te sturen.
Het fysiek isoleren van de communicatielijnen kan een oplossing zijn tegen vervalsing en lekkage. Dit is echter niet bij alle gedistribueerde systemen mogelijk. Denk bijvoorbeeld aan systemen die het Internet gebruiken voor communicatie, zoals een webapplicatie of applicaties met een business-to-business karakter.
Een ander beveiligingsmechanisme voor de communicatie is encryptie. Daarbij worden de communicatieberichten gecodeerd, zodat ze niet gelezen of aangepast kunnen worden door onbevoegden. We maken onderscheid tussen integrity en confidentiality mechanismen.



Figuur 7-6: Communicatie aanvallen

Confidentiality mechanismen verzekeren private communicatie tussen entiteiten, door de inhoud van het bericht te encrypten. Dit voorkomt het afluisteren, zoals te zien is in situatie A van Figuur 7-6. Omdat symmetrische encryptie minder rekenintensief is dan asymmetrische encryptie, komt het vaak voor dat het berichtenverkeer via een symmetrische methode gecodeerd wordt en de sleutel hiervan met behulp van een asymmetrische methode.
Ook deze mechanismen worden binnen de J2EE-architectuur door de containers verzorgd, op basis van SSL/TLS[35] zodat de beveiliging toegepast wordt als bijeffect bij het maken van de connectie. De deployer, eventueel met hulp van de assembler, stelt de containers zo in dat ze confidentiality mechanismen toepassen bij communicatie over onveilige netwerken.

[35] Secure Socket Layer / Transaction Layer Security

Bovendien dient hij ook aan te geven dat containers onbeveiligde berichten weigeren wanneer deze gecodeerd zouden moeten zijn.

7.2.4 Data integriteit

Integrity mechanismen voorkomen dat een onbevoegde partij de communicatie tussen entiteiten vervalst. Vervalsing betekent hier het meerdere malen versturen van een onderschept bericht of het aanpassen en doorsturen van een onderschept bericht. Zie situatie B en C in Figuur 7-6. De integriteit van het bericht wordt gewaarborgd door een *signature* aan een bericht te koppelen. Deze signature wordt met een hash-algoritme berekend en converteert het bericht naar een veel kleiner *message digest.* Dit digest wordt tenslotte gecodeerd, zodat een verandering van het bericht door iemand anders dan de oorspronkelijke verstuurder gedetecteerd kan worden door de ontvanger.
Binnen de J2EE-architectuur implementeert de container de integriteitmechanismen voor de communicatie tussen de aanroepers en de componenten die de container host. Dit moet gebeuren in de transportlaag. De deployer dient de containers zo te configureren dat ze integriteitmechanismen toepassen wanneer de interactie over onbeveiligde communicatielijnen plaatsvindt, of wanneer de communicerende componenten niet in hetzelfde protectiedomein zitten. De overhead prijs die betaald moet worden voor deze beveiliging kan onder andere beperkt worden door te selecteren van welke berichten de integriteit beveiligd moet worden. Dit vereist applicatiespecifieke kennis en wordt in de deployment descriptor van de applicatie bewaard.

7.2.5 Accountability

De besproken mechanismen hebben allemaal een preventief karakter. Ze proberen te voorkomen dat er een beveiligingsbreuk plaatsvindt. Wanneer dit echter gebeurt, is het veel belangrijker om na te kunnen gaan wie toegang heeft gekregen tot het systeem en welke acties hij heeft uitgevoerd, dan om te weten wie de toegang geweigerd is. *Accountability mechanismen* verzekeren dat een actie zonder twijfel gekoppeld kan worden aan zijn initiator [C8].
Auditing is een accountability mechanisme. *Auditing* is het onomkeerbaar vastleggen van alle gebeurtenissen die te maken hebben met beveiliging, om zo gebruikers (zowel mensen als systemen) aansprakelijk te stellen voor hun acties. [A7]. Bovendien kan zo de effectiviteit van de gedragslijnen en beveiligingsmechanismen geëvalueerd worden. Hiervoor moeten ontwikkelaars wel een goed beeld hebben wat ze moeten auditen en wat niet. De opgeslagen gegevens moeten ook beveiligd worden, zodat overtreders hun eigen sporen niet kunnen wissen.
De huidige J2EE-specificatie verplicht ondersteuning van auditing-functionaliteit niet, maar naar verwachting zal dit in de toekomst wel het geval zijn.

8 Index Tracken

<Verwijderd op verzoek van ORTEC bv>

9 J2EE in de praktijk: testopzet

9.1 Inleiding

De theorie achter het Java 2, Enterprise Edition platform is veelbelovend. Maar de theorie is slechts een kant van het verhaal. De toepasbaarheid, het draagvlak en het nut van de concepten en stuurmiddelen in de praktijk is veelbepalend voor de totale waarde van een gedistribueerde architectuur zoals J2EE.
Hoofdstuk 9 en 10 gaan in op een empirische toetsing van het J2EE-platform. Dit hoofdstuk bespreekt de opzet van de toetsing, in het kader van het besproken "Index Tracking"-model. Het beschrijft wat er getoetst wordt en op welke manier. Het volgende hoofdstuk beschrijft de bevindingen.

9.1.1 Test opzet

Het in detail opzetten en testen van een volledige gedistribueerde webapplicatie als Index Tracking valt buiten de strekking van dit rapport. We zullen daarom de test zo inrichten dat we de basisfunctionaliteit van J2EE kunnen toetsen en een globale indruk krijgen van de toegevoegde waarde van het J2EE-model bij een implementatietraject. De techniek is daarom leidend, en het Index Tracking model dient als een voorbeeld. De "design goals" van enterprise-applicaties dienen hierbij als uitgangspunt.
We gebruiken een beperkt functioneel model van Index Tracking, dat zal dienen als implementatievoorbeeld bij de individuele testen. Bij elke test breiden we het model uit, of passen we het aan, om zo het J2EE-stuurelement mee te kunnen nemen in de toetsing.
Sommige elementen uit de theorie zijn moeilijk te testen. Bijvoorbeeld omdat ze erg veel tijd in beslag nemen, of omdat ze veel kosten met zich meedragen. Zo zijn schaalbaarheid en performance in het kader van dit rapport moeilijk te toetsen. Hiervoor zouden we verschillende configuraties moeten vergelijken en een hoop gebruikers simuleren. Voor een indruk van deze elementen, vallen we terug op de literatuur en de bevindingen van anderen. Deze ervaringen komen alleen terug in het volgende hoofdstuk, evenals de onderdelen die geen specifieke testopzet kennen, zoals het gebruik van de Java-programmeertaal en deployment descriptors.

9.1.2 Specificaties

De gebruikte hardware en software spelen een grote rol in het uitvoeren van een dergelijke test. Ze hebben invloed op bijvoorbeeld performance, functionaliteit en het realisatietraject. Zo bepaalt een harde schijf voornamelijk de performance van de database server, een J2EE-server de aangeboden diensten en een ontwikkelomgeving het scala aan tools voor ontwikkeling, integratie en deployment.
Voor de volledigheid melden we hier de gebruikte hardware- en softwarespecificaties, die als uitgangspunt hebben gediend voor de testprojecten.

Server	Processor	Geheugen	Harde schijf	Besturingssysteem
All-in-one	PIII-850 MHz	512 MB SDRAM	20 GB IDE 5400 RPM	Windows 2000 Pro
Applicatieserver	PIII-500 MHz	256 MB SDRAM	10 GB IDE 5400 RPM	Windows 2000 Pro

Tabel 4: Hardware configuratie

De gebruikte hardware staat vermeld in Tabel 4. De meeste testen vinden plaats op een enkele machine, vanwege het kleinschalige karakter. Voor het specifiek testen van gedistribueerde techniekonderdelen is een tweede applicatieserver gebruikt. Tabel 5 laat de relevante software zien voor deze test. De paragraaf "softwarevrijheid" licht de softwarekeuzes toe.

Software	Versie	Opmerkingen
JBoss	3.0.4	J2EE 1.3 applicatieserver
Borland Applicatieserver (BAS)	5.0	J2EE 1.3 applicatieserver
Borland JBuilder Enterprise	7.0	Ontwikkelomgeving
Apache Struts	1.0.2	Web application framework
Tomcat	4.1.12	J2EE Webserver
MySQL	4.0.12	Database Management Systeem
MS Access	2000	Database Management Systeem

Tabel 5: Gebruikte softwareproducten

9.2 Realisatietraject

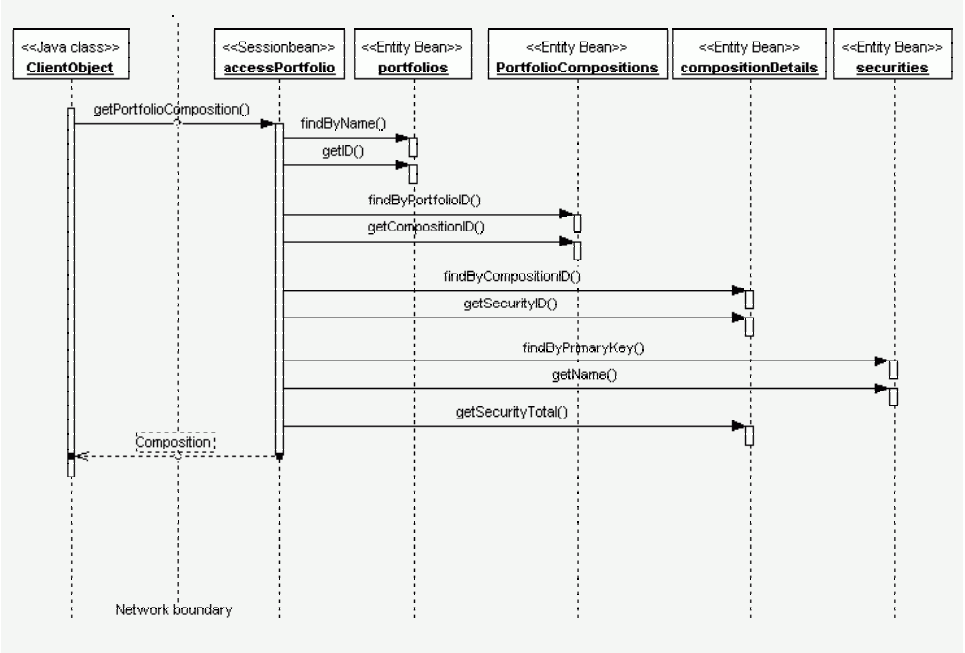
9.2.1 Softwarevrijheid

De softwarekeuzes voor deze testcase zijn gemaakt op basis van drie factoren: populariteit, kosten en gebruikersgemak. JBuilder is op dit moment de populairste ontwikkelomgeving voor Java en heeft goede grafische ondersteuning voor het ontwikkelen van J2EE-componenten. JBoss is een populaire open-source J2EE-implementatie, die goed concurreert met commerciële implementaties. Tomcat is ook een voorbeeld van een open-source server. JBoss geniet veel populariteit door zijn volledigheid, terwijl Tomcat juist meer geschikt is voor de wat eenvoudigere webapplicaties. Samen eisen ze een steeds groter deel van de applicatieserver-markt op [B21]. De Borland applicatieserver wordt meegeleverd met JBuilder en is een mooie test voor de geclaimde implementatie onafhankelijkheid. Jakarta STRUTS is het populairste web application framework op dit moment en tevens gratis te

verkrijgen. MySQL is op haar beurt een populaire open-source database service en MS Access is gekozen omdat het geen JDBC compatible database-systeem is. Deze keuzes zorgen voor de softwarelijst in Tabel 5.

9.2.2 Component Model Architectuur

In de eerste opzet willen we het gebruik van enterprise beans bekijken. Zij vormen één van de J2EE-componenttypen. We representeren de tabellen van het datamodel met behulp van entity beans. Een session bean vraagt de samenstelling van een portefeuille op door de lokale interfaces van de benodigde entity beans te benaderen en print deze op het scherm. De test maakt gebruik van de sessionfaçade-techniek, beschreven in hoofdstuk 5.



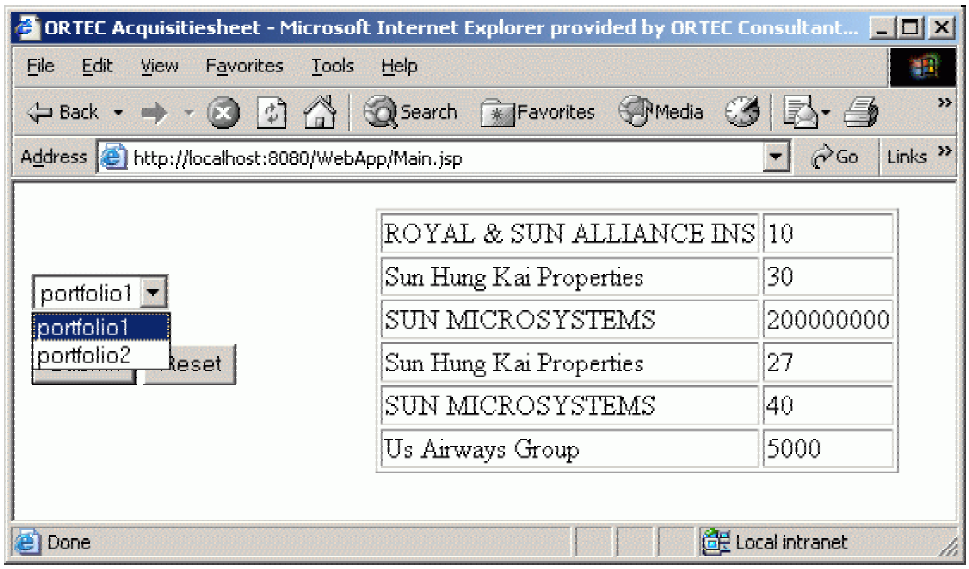
Figuur 9-1: Sequentie diagram 'ophalen portefeuillesamenstelling'

Figuur 9-1 laat een sequentiediagram zien van deze operatie. De eerste aanroep naar een entity bean heeft dezelfde functie als de "where" clause van een database query. Deze zoekt de betreffende entity bean instantie op basis van een sleutelveld.

9.2.3 JavaServer Pages

Een klant die gebruik maakt van de managementservice, moet met behulp van een web browser een portefeuillesamenstelling kunnen opvragen uit de database. Vervolgens drukt hij op een knop om de samenstelling van deze portefeuille te kunnen bekijken. Figuur 9-2 laat een screenshot zien van de gebruikersinterface.
Voor het model betekent dit een aanpassing van het clientobject. Dit wordt een JSP-pagina, die de portefeuilleselectie van de gebruiker doorgeeft aan de

'accessPortfolio' session bean. Een tweede JSP-pagina ontvangt vervolgens de portefeuillesamenstelling en drukt deze af in een tabel.



Figuur 9-2: JSP-view van Index Tracking

9.3 Openheid

9.3.1 MVC-patroon en Struts

De implementatie van het JSP-project heeft wat nadelen. Het toevoegen van een nieuw type client of het veranderen van de paginavolgorde brengt veel werk met zich mee. We willen het MVC-patroon toepassen met behulp van het Jakarta Struts-framework om zo de presentatie beter te scheiden van de businesslogica.

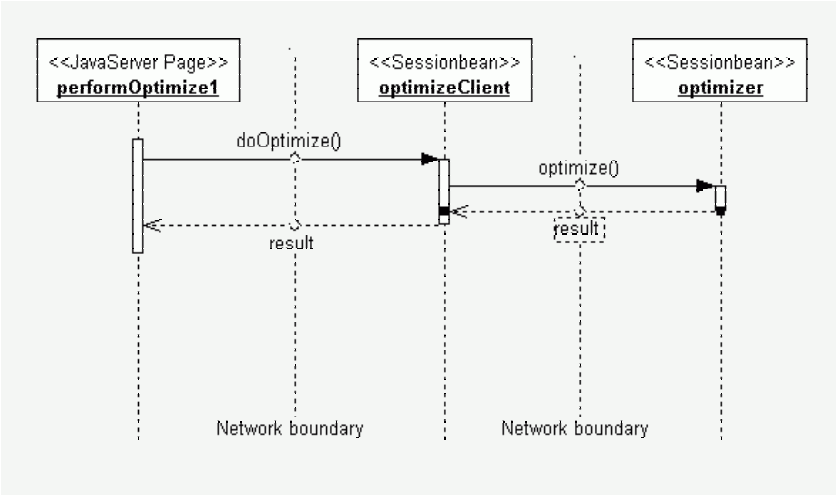
9.3.2 Implementatie onafhankelijkheid

Doordat J2EE zelf een specificatie is en geen implementatie, is het in theorie mogelijk om een J2EE-applicatieserver van een aanbieder te vervangen door een J2EE-applicatieserver van een andere aanbieder. Daarvoor hoeft de applicatiecode niet gewijzigd te worden, mits er geen gebruik gemaakt is van de additionele functionaliteit van een applicatieserver, die buiten de J2EE-specificatie valt. Met wat configuratieaanpassingen zou de applicatie moeten kunnen draaien op de nieuwe applicatieserver.
Voor deze test nemen we het Struts testproject van de vorige paragraaf. We vervangen de JBoss-applicatieserver door de applicatieserver van Borland om te zien of deze overstap eenvoudig te maken is.

9.3.3 Gedistribueerde object technologie & Naming

Naar verwachting zullen er meer managementservice gebruikers zijn, en minder indexeerservice gebruikers. We willen de intensieve optimalisatieberekeningen op een andere server laten plaatsvinden, zodat deze

geen of weinig invloed hebben op de performance van de managementservicediensten.



Figuur 9-3: Sequentie diagram 'optimalisatie'

We plaatsen een stateless session bean op een tweede applicatieserver, die alleen een 'optimize'-methode kent. Een session bean op de andere server roept deze functie aan en maakt daarvoor gebruik van RMI en JNDI. Met JNDI kan de client-component de optimizer-bean lokaliseren en met RMI moet de client op een transparante manier de optimalisatiemethode kunnen aanroepen.

9.3.4 Database onafhankelijkheid

Met behulp van de JDBC API kan een Java-applicatie communiceren met een willekeurige relationele database, waar een JDBC-driver voor bestaat. Wanneer er geen database specifieke SQL-code gebruikt is, zou het overstappen naar een ander database systeem moeten kunnen door het simpelweg vervangen van de JDBC-driver.
We vervangen de MySQL database service door een Microsoft Access database. Hiervoor is een JDBC-ODBC driver nodig. De nieuwe database heeft een afwijkend datamodel, waarbij sommige tabelnamen veranderd zijn. Wat voor gevolgen heeft dit voor de gebruikte entity beans in het model?

9.4 Beveiliging

9.4.1 Authenticatie

Binnen de Index Tracking applicatie kunnen twee soorten gebruikersgroepen onderscheiden worden. De indexeerservice-klanten en de managementservice-klanten. Gebruikers moeten zich met behulp van een inlogscherm kunnen identificeren. Het beveiligingsmodel specificeert in welke gebruikersgroep een klant zit. De applicatie bewaart de authenticatiegegevens.

9.4.2 Autorisatie

De mogelijkheid om te rebalancen is alleen voor gebruikers van de indexeerservice. We passen daarom autorisatie toe op het testproject van paragraaf 9.3.3. Daarbij maken gebruik van de authententicatiegegevens van het vorige project. Managementservice-klanten die gebruik proberen te maken van deze dienst krijgen een foutmelding.

10 J2EE in de praktijk: ervaringen

10.1 Inleiding

De evaluatie van een ontwikkelplatform kan het best op basis van ervaringen uit de praktijk. In het vorige hoofdstuk definieerden we reeds de opzet voor de J2EE-testprojecten. Dit hoofdstuk bespreekt de resultaten van de testprojecten en de algemene indruk die J2EE achtergelaten heeft bij de implementatie van deze projecten. De indeling van dit hoofdstuk is op basis van de opgestelde criteria voor enterprise-applicaties.
Naast eigen bevindingen staan in dit hoofdstuk de ervaringen van anderen en de stand van zaken met betrekking tot bepaalde ontwikkelingen. Deze informatie komt uit de literatuur.

10.2 Realisatie

10.2.1 Softwarevrijheid

De J2EE-specificatie heeft de aandacht van een groot aantal bedrijven gekregen, door de ondersteuning van vele standaarden en de betrokkenheid van verschillende marktpartijen bij het definiëren van deze specificatie. Medio 2002 waren er een dertigtal J2EE 1.3 implementaties van verschillende aanbieders. Deze implementaties zijn onder te verdelen in commerciële producten (Appendix C: J2EE Licentiehouders) en open-source projecten.
Het scala aan J2EE-servers neemt inderdaad de afhankelijkheid van een specifieke aanbieder weg, maar vereist wel een extra vooronderzoek. Het in kaart brengen van functionaliteit, kosten, performance en integratiemogelijkheden van de verschillende implementaties is een kostbare investering.
De “luxe” commerciële implementaties kunnen tienduizenden of honderdduizenden dollars kosten [B21]. De opmars van de open-source implementaties is dan ook begrijpelijk. Mede door de complexiteit van J2EE is het moeilijk voor commerciële leveranciers om de toegevoegde waarde van hun applicatieserver te laten zien. BEA en IBM hebben al minder uitgebreide applicatieservers op de markt gebracht, die een paar honderd of een paar duizend dollar kosten. Er zullen naar verwachting ook geen nieuwe J2EE-leveranciers meer bijkomen. De harde concurrentie heeft er al voor gezorgd dat sommige leveranciers, waaronder HP, hebben moeten stoppen met hun J2EE-implementatie.
Er is een grote markt van 3rd-party tools voor J2EE. Hiervoor geldt eigenlijk hetzelfde verhaal als bij de J2EE-implementaties. Je zit niet vast aan een tool, maar een goede afweging kost een hoop tijd.
.

10.2.2 Opstartfase

Een punt wat niet in de literatuur genoemd wordt, maar waar elke ontwikkelaar tegenaan loopt bij een nieuwe technologie is de installatie, de configuratie en het gebruik ervan.

De praktijk blijkt lastiger dan de theorie belooft. Het ontwikkelen van een enterprise-applicatie blijft verre van triviaal, ondanks de vele diensten en tools die een ontwikkelteam aangereikt worden. Voor de opstartfase is concrete documentatie cruciaal, net zoals hulpmiddelen in de vorm van tools en wizards. We bespreken de ervaringen met documentatie en hulpmiddelen apart.
De documentatie over J2EE is talrijk, dankzij de grote community die erachter staat. Op het Internet word je overspoeld met white papers, handleidingen, tutorials, specificaties, columns, newsgroups en andere artikelen. Ook zijn er vele boeken geschreven over J2EE en haar bijbehorende technologieën. Toch zitten er wat nadelen aan al deze documentatie. Door de omvang van de informatie en de verschillende J2EE-implementaties, ontbreekt het aan een eenduidige richtlijn. Boeken zijn over het algemeen goed voor theorie, maar minder nuttig in de praktijk. Er is behoefte aan documentatie voor elke implementatie, omdat deze allemaal hun eigen problemen en oplossingen hebben. Wat voor een beginnende J2EE-ontwikkelaar cruciaal is, zijn concrete artikelen, die stap voor stap beschrijven hoe je van A naar B komt, gegeven dat je *die* applicatieserver gebruikt, *die* webserver, *die* ontwikkelomgeving, etc. En die documentatie is soms moeilijk te vinden.
Door de populariteit van J2EE, komen ook legio tools op de markt. Deze tools variëren van open-source hobby-tooltjes, tot dure grootschalige oplossingen. De integratie van verschillende tools gaat niet altijd even gemakkelijk en soms helemaal niet. In het laatste geval moet de ontwikkelaar voor een handmatige oplossing kiezen. Voor de handmatige oplossing is veel kennis vereist van de onderliggende techniek, en dat is niet gewenst in een opstartfase. Kortom: standaardisatie van de tools is een belangrijk punt voor J2EE. Nu kan tool X het beste zijn voor configuratie A, terwijl tool Y het populairst is voor configuratie B. In de praktijk komt het voor dat tool X niet werkt met configuratie B.

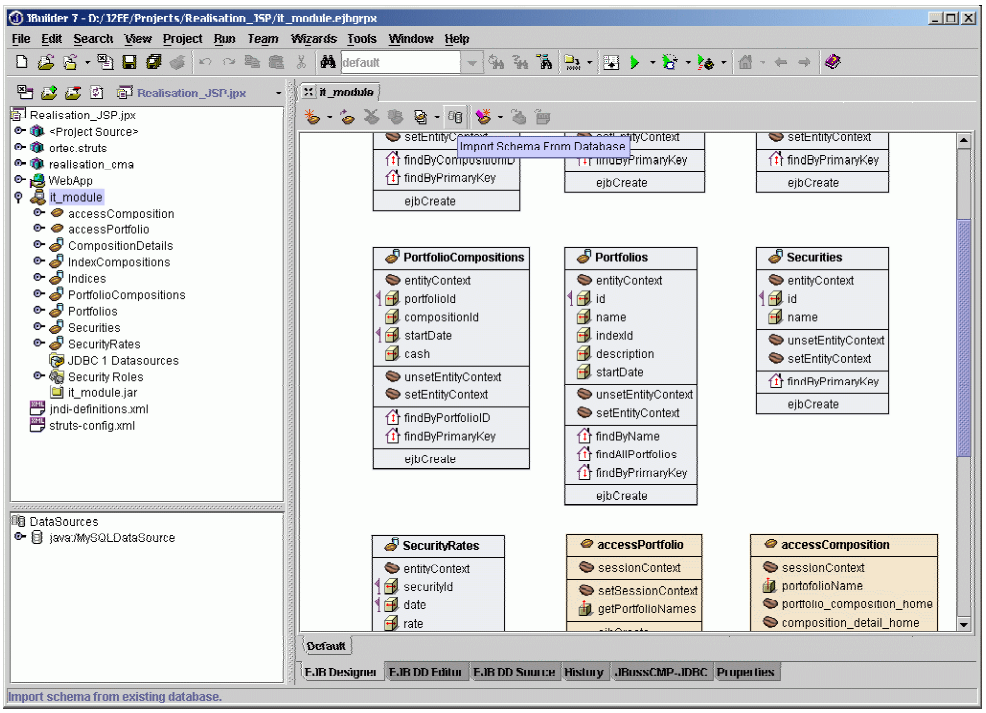
10.2.3 Java-programmeertaal

Met C++ als achtergrondkennis is de overstap naar Java goed te doen. Java is een goed doordachte taal met veel bibliotheken en documentatie. Het biedt de flexibiliteit van C++, maar op een hoger abstractieniveau. De verplichting om binnen het J2EE-platform in Java te programmeren is dan ook geen groot bezwaar.
Een minpuntje is de ondoorzichtige garbage collectie. Waar volgens de literatuur het geheugenbeheer niet langer de zorg is van de ontwikkelaar wijst de praktijk anders uit.

10.2.4 Component Model Architectuur

Het implementeren van entity beans objecten is verbazingwekkend eenvoudig, met behulp van de grafische tools in JBuilder. Figuur 10-1 laat met behulp van een screenshot zien hoe enterprise beans grafisch worden weergegeven in de ontwikkelomgeving. Er is directe interactie mogelijk op de objecten in de view. De mapping van de Java-datatypen naar de database typen staat beschreven in een deployment descriptor van de leverancier. Wanneer een mapping niet beschreven is voor een specifieke database, dan moet de ontwikkelaar dit aanpassen in de descriptor.

Het gebruik van het OO-model binnen enterprise beans is consistent. De enige complicatie die we in dit testproject tegenkwamen was de overstap van remote interfaces naar local interfaces. Local interfaces verdienen de voorkeur vanuit een performance oogpunt. In deze test draaien alle componenten binnen dezelfde virtuele machine, dus de overhead voor remote communicatie is niet nodig. Nu blijkt het dat de naming context van JNDI voor local interfaces implementatieafhankelijk is.



Figuur 10-1: EJB-view in JBuilder

10.2.5 JavaServer Pages

Het gebruik van JavaServer Pages is een prettige manier om webpagina's te maken met een dynamische inhoud. Een ontwerper kan de HTML-pagina maken, waarna een ontwikkelaar de benodigde code toevoegt. Het vereist wel inspanning van de ontwikkelaar om de JSP-pagina overzichtelijk te houden. Voor het aanroepen van de enterprise beans is bijvoorbeeld een JNDI-context nodig. Als dat op meerdere plaatsen staat gedefinieerd, wordt de pagina al snel onoverzichtelijk.

10.2.6 Packaging & Deployment

Het snel en eenvoudig deployen en undeployen van applicaties en applicatieonderdelen (modules) is essentieel tijdens de ontwikkelfase. De testprojecten geven een goed beeld van het deploymentproces, omdat ze vaak gedeployed moeten worden en daarbij ook nog eens op verschillende machines.
J2EE scoort goed op dit punt. Het is een verbazingwekkend rechttoe rechtaan proces, waarbij het wel helpt als de ontwikkelaar weet wat er achter de

schermen gebeurt (zie paragraaf 3.2.6). Er moet bij gezegd worden dat een centrale rol is weggelegd voor de ontwikkelomgeving. Deze bepaalt in grote mate het gemak van het packaging- en deploymentproces.
Het verpakken van componenten in modules gebeurt bijna volledig zonder interventie van de ontwikkelaar. Deze geeft alleen aan welke componenten bij een module behoren. De ontwikkelomgeving zorgt automatisch voor de benodigde deployment descriptors en bijbehorende editors. Met een druk op de knop worden de componenten gecompileerd en samengevoegd in een deployment unit.
Het deployen is al even gemakkelijk. Wederom met een druk op de knop kan iemand een module of applicatie deployen, redeployen of undeployen. Hierbij ondersteunen sommige applicatieservers de *hot deployment* eigenschap, wat inhoudt dat de server niet gestopt hoeft te worden tijdens de deploymentfase. Op dit moment zijn deployment-tools nog niet gestandaardiseerd. De literatuur belooft dat dit wel het geval zal zijn in de volgende versie van J2EE. Dan zal het deployen naar andere applicatieservers met dezelfde deployment tools kunnen gebeuren.

10.2.7 Rolverdeling

De onderverdeling tussen ontwikkelaar, assembler en deployer lijkt in eerste instantie misschien niet logisch. Maar door het consistente componentenmodel en het “declaratieve” karakter van J2EE, heeft deze expliciete onderverdeling wel degelijk nut voor de specialisatie in een ontwikkeltraject. Modules zijn autonome objecten die gebruikt kunnen worden als bouwstenen voor een applicatie. Dit kan door iemand anders gedaan worden dan de ontwikkelaar zelf, zoals de assembler. Bovendien kunnen de assembler en de deployer veel sturen in het applicatiegedrag door alleen het wijzigen van deployment descriptors.
Nu gooit de J2EE-specificatie in haar rolverdeling verschillende soorten “ontwikkelaars” op één hoop. De eerder genoemde specialisatiemogelijkheid met de JavaServer Pages techniek mag niet ongenoemd blijven. De paginaontwerper hoeft niets van de programmeercode af te weten, terwijl de programmeur met weinig moeite zijn code kan toevoegen. Dit laatste geldt des te meer wanneer er gebruik gemaakt wordt van het MVC-patroon en een webapplicatie-framework zoals Struts (zie 10.3.1).
De twee rollen “J2EE-productleverancier” en “Tool-leverancier” zijn vanuit het oogpunt van de organisatie die J2EE gebruikt om enterprise-applicaties te creëren eigenlijk overbodig. Het is logisch dat de J2EE-server implementatie en verwante tools door externe partijen worden geleverd. Dit is niet anders bij andere ontwikkelplatformen.

10.3 Openheid

10.3.1 MVC-patroon en Struts

Struts blijkt ook in de praktijk een heel nuttig framework. Hoewel de kracht van Struts pas echt naar voren komt bij grote projecten, heeft het gebruik van Struts al voordelen bij een kleine applicatie (zoals het uitgevoerde testproject).

Ten eerste houdt het gebruik van Struts de JSP-pagina's schoon. Het koppelen van een *action* aan een pagina in het Struts-framework, is voldoende om de gewenste methode aan te roepen. Dit maakt bijvoorbeeld het specificeren van een JNDI-context in de JSP-pagina overbodig Een voorbeeld is te zien in Figuur 10-2, waar de "showComposition"-action wordt geïnstantieerd.

showComposition

```
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>

<html:html>
<head>
<title>
showComposition
</title>
</head>
<body>

<html:form action='/showComposition'>
<br><br>
portfolioName:<html:text property="portfolioName" />
<html:submit property="submitButton" />
</html:form>
</body>
</html:html>
```

Figuur 10-2: JSP-pagina met gebruik van Struts

Een belangrijk voordeel van Struts is de mogelijkheid om gebruik te maken van tag-libraries. Deze dragen ook een steentje bij in het schoonhouden van JSP-pagina's. In dit voorbeeld is gebruikt gemaakt van de 'HTML- tag-library'. Struts maakt zo automatisch een *ActionForm* aan, met een stel propertjes. In ons voorbeeld de portefeuillenaam. Op de server kan dan de waarde van een attribuut met behulp van methoden verkregen worden. Dit is vele malen handiger dan het uitpluizen van een request-object. Bovendien blijven de datatypen bewaard. In een request-object staat alles opgeslagen in stringparen. Een nadeel van de tag-libraries is dat ze geen debugmogelijkheden bieden. De enige debugmogelijkheid is het uitprinten van waarden.
Wijzigingen in de control flow kunnen centraal in de controller Servlet doorgevoerd worden. Dit is in feite niet anders dan het aanpassen van een deployment descriptor. Dit maakt het applicatieonderhoud veel overzichtelijker.
Een webapplicatie-framework als Struts "dringt je" eenvoud op. Presentatie en businesslogica zijn met het gebruik van Struts volledig onafhankelijk van elkaar. Bovendien verzorgt het framework bijna alle implementatie voor de communicatie tussen de presentatie en de businesslogica. Dit moet bij grote projecten enorme tijdwinsten opleveren. Wat dat betreft zou het ook als een stuurmiddel bij het criterium "Realisatie & onderhoud" kunnen staan.

10.3.2 Implementatie onafhankelijkheid

Eén van de belangrijkste argumenten voor het gebruik van J2EE in plaats van concurrerende producten als .NET is de mogelijkheid om van productleverancier te veranderen [C7]. Daarbij is het aanbod groot (zie Appendix C: J2EE Licentiehouders) en zitten er grote namen tussen de leveranciers.
Net zoals bij de initiële softwarekeuze moet er een degelijk onderzoek gedaan worden naar goede alternatieve aanbieder. Daarbij moet tevens weer een kostenafweging gemaakt worden. Ook speelt het "tool integratieprobleem[36]" weer op. Deze factoren spelen een belangrijke rol in de benodigde tijd en kosten van de overstap.
Op het technische vlak is het geen kwestie van "plug & play". Het klopt in beginsel dat er geen code-aanpassingen nodig zijn in de componenten, wanneer geen gebruik gemaakt is van functionaliteit die buiten de J2EE-specificatie valt. Maar ook wanneer er alleen gebruik gemaakt wordt van functionaliteit die binnen de specificatie valt, is er werk aan de winkel bij een overstap. Het probleem zit hem in het feit dat de J2EE-specificatie wel aangeeft welke functionaliteit aangeboden moet worden, maar niet hoe deze functionaliteit geïmplementeerd moet zijn. De API's staan vast, maar de DTD's van de deployment descriptors niet. Aanbieders kunnen extra elementen toevoegen aan de standaard deployment descriptors of eigen descriptors definiëren. Zo zijn de elementen in de deployment descriptors voor het beveiligingsmodel, datatype-mapping, en JNDI naming contexten anders bij elke aanbieder.
Nu hoeft dit alles niet te betekenen dat een overstap naar een andere J2EE-productleverancier een lang traject is. Naar eigen schatting is dit eerder een traject van dagen of weken dan van maanden. Het zal met name afhangen van de kwaliteit van de documentatie.

10.3.3 Gedistribueerde objecttechnologie en naming

Het communiceren met een enterprise bean op een andere server is nauwelijks anders dan met een enterprise bean binnen dezelfde virtuele machine. Het verschil zit in de servernaam en het gebruik van de remote interface.
In Figuur 10-3 is de code te zien die nodig is om de optimizer bean te lokaliseren en te gebruiken vanaf de client. De JNDI initial context bevat de informatie voor JNDI voor het lokaliseren van de server. De "provider-url" specificeert de servernaam en het poortnummer. Vervolgens hoeft er alleen nog op de naam van het server-component gezocht te worden. Een ervaren RMI-programmeur verwacht dat de client een stub nodig heeft om te kunnen compileren[37], maar dat is niet nodig. Doordat de deployment tools automatisch de stubs en skeletons aanmaken [A7], heeft de client tijdens de ontwikkelfase alleen de home en remote interface nodig.

[36] Het "tool integratieprobleem" staat beschreven in paragraaf 10.2.2 Opstartfase
[37] Zie paragraaf 4.3.3, gedistribueerde objecttechnologie

optimizeClient | optimizer | optimizerBean | performOptimize

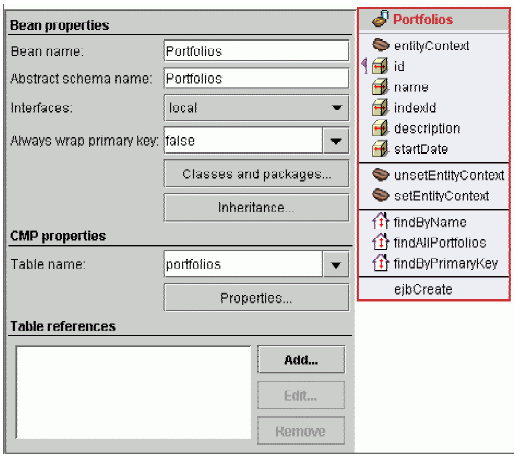
```
public double doOptimize(double p1, double p2)
{
  double result = 0;
  try
  {
    Context jndiContext = getInitialContext();
    Object ref = jndiContext.lookup("optimizer");
    optimizerHome home = (optimizerHome) PortableRemoteObject.narrow(ref, optimizerHome.class);
    optimizer r = home.create();
    result = r.optimize(p1,p2);
  }
  catch (Exception e)
  {
    e.printStackTrace();
  }
  return result;
}

public static Context getInitialContext() throws NamingException
{
  Properties p = new Properties();
  p.put(Context.INITIAL_CONTEXT_FACTORY, "org.jnp.interfaces.NamingContextFactory");
  p.put(Context.PROVIDER_URL, "p-539:1099");
  p.put(Context.URL_PKG_PREFIXES, "org.jboss.naming:org.jnp.interfaces");
  return new javax.naming.InitialContext(p);
}
```

Figuur 10-3: applicatiecode voor het gebruik van JNDI en RMI

RMI en JNDI zorgen ervoor dat het benaderen van remote objecten nauwelijks moeilijker is dan het benaderen van objecten in hetzelfde proces.

10.3.4 Database onafhankelijkheid

Het wisselen van een database in een J2EE-applicatie is in feite niet meer dan het vervangen van de JDBC-driver[38]. De keuze is ook groot, want voor bijna alle databases bestaat een JDBC-driver. Bij container managed persistentie moeten de regels voor het “mappen” van de datatypen nog gespecificeerd worden voor de nieuwe database. Paragraaf 10.5.2 geeft hier meer informatie over.



Figuur 10-4: Database mapping in Jbuilder

[38] We laten de problemen met het overstappen naar een andere database die los staan van het J2EE-platform buiten beschouwing.

Dan het datamodel. De verandering van een tabelnaam of veldnaam heeft geen grote gevolgen voor de gekoppelde entity beans. In ons geval was het een kwestie van een andere tabelnaam toekennen aan een entity bean. Figuur 10-4 laat zien voor een 'portfolios'-bean. In de CMP-sectie van het scherm kan uit een lijst met bestaande tabellen gekozen worden.

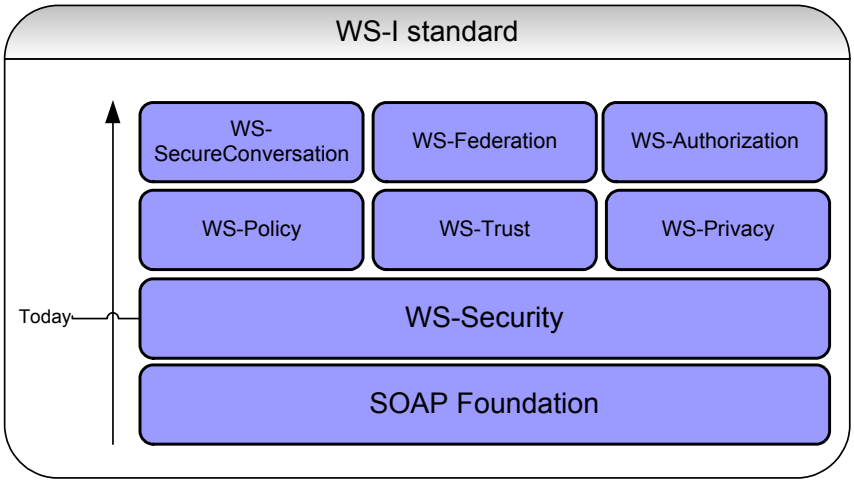
10.3.5 Web services

Web services zijn "hot" en de toekomst voor web services is groot. Daar is de literatuur het wel over eens [A13, B15, B14, C6]. Behalve expliciete waarde oordelen zijn er ook impliciete aanwijzingen, doordat bedrijven bereid zijn te investeren in web services. Een voorbeeld is de primaire focus van J2EE 1.4 op web services [B7].
Het nut van web services bij business-to-business integratie heeft met name geleid tot de grote populariteit van web services. Overigens worden op dit moment web services vooral gebruikt voor het integreren van interne systemen. Onderzoek geeft aan dat de markt voor web services zal groeien tot 21 miljard dollar in 2007 [B14]. Daarbij zijn J2EE en .NET de favoriete ontwikkelplatformen. Deze houden elkaar in evenwicht, waarbij grote bedrijven een voorkeur hebben voor J2EE en kleinere bedrijven een voorkeur hebben voor .NET. Volgens de Gartner groep zal de strijd tussen de platformen in ieder geval tot 2008 niet beslist zijn.
De reden waarom web services de toekomst zijn voor integratie en communicatie heeft volgens de literatuur te maken met een aantal eigenschappen van web services [B15, C6]:

- Gebruik van XML als data-standaard
- Gebruik van het standaard lightweight SOAP protocol
- Vereisen geen uitgebreide technische infrastructuur[39]
- Platformonafhankelijk
- Taalonafhankelijk
- Synchrone en asynchrone communicatiemogelijkheden
- Dynamisch te "ontdekken"
- Beschrijven zichzelf voor de buitenwereld (WDSL)

Ondanks al deze voordelen zijn er nog wat donkere wolken aan de horizon. Meer dan twintig procent van de ontwikkelaars ziet beveiliging en onduidelijkheid in standaarden als de grootste obstakels voor het ontwikkelen van web services. Beveiliging is een belangrijk aspect van web services. Ze draaien in een open omgeving en kunnen door een firewall methoden aanroepen van een applicatie [C6]. Een extra WS-Security laag boven op de SOAP-fundering zorgt voor een op XML-gebaseerde beveiliging voor digitale handtekeningen en encryptie. Zoals Figuur 10-5 laat zien is dit de huidige technologische stand van zaken. WS-security zal onderdeel gaan uitmaken van een nog groter dienstenpakket voor web services, genaamd "WS-I".

[39] zoals een JVM of een CORBA-infrastructuur



Figuur 10-5: De WS-I standaard

Dat er onduidelijkheid bestaat over standaarden lijkt in eerste instantie vreemd, aangezien web services juist populair zijn dankzij het gebruik van standaarden als SOAP, XML en WSDL. Dit zijn echter technische standaarden. De standaardisering op bedrijfsprocesniveau is nog een gevecht [C5]. Twee kampen[40] zijn bezig elk een standaard op te zetten voor complexere webdiensten, zoals een samenwerkingsverband tussen drie partijen. Het unificatieproces van deze zogenaamde *choreografiestandaarden* is gestopt na onenigheid over de vraag of er licentiekosten gevraagd mogen worden voor het intellectuele eigendom van de ontwikkelde kennis. Nu dreigen er twee overlappende standaarden op de markt te komen.

10.4 Schaalbaarheid & Performance

10.4.1 Eigen bevindingen

Deze paragraaf beschrijft de algemene indruk die met name Enterprise JavaBeans achtergelaten heeft op het gebied van performance. De resultaten zijn gebaseerd op een situatie met één gebruiker.
Tijdens het uitvoeren van de testprojecten viel op dat het gebruik van instance pooling voor een betere performance zorgt dan het handmatig instantiëren van beans. Hoewel dit eigenlijk als schalingsmiddel bedoeld is, is het verschil ook al merkbaar bij één gebruiker. De overhead van het aanmaken en verwijderen van component-instanties is dus niet verwaarloosbaar en het nut van instance pooling is daarmee ook bewezen.
Een terechte vraag is hoe groot de overhead van entity beans is ten opzichte van een directe databasebenadering met behulp van SQL. Voor elk record moet immers een instantie van een entity bean gevuld worden. We voerden een kleine test uit, die alle fondsen uit de database inleest en afdrukt. Dit levert een resultaatset op van bijna 1500 records. Het blijkt dat het gebruik van entity

[40] Eén partij wordt gevormd door Microsoft, IBM en BEA. De andere partij bestaat uit Sun, Oracle, Intalio, SAP, W3C en eveneens BEA.

beans bijna *dertig* keer langzamer is dan het gebruik van SQL[41]. Voor een OO-laag om de database wordt dus duur betaald. Ontwikkelaars zullen afwegingen moeten maken wanneer ze entity beans gebruiken en wanneer niet. Overigens heeft de grootte van de "instance pool" bijna geen invloed op de resultaten. Dit zal echter wel verschil maken, wanneer er sprake is van meerdere gelijktijdige gebruikers.
Tenslotte nog een opmerking over de performance van typen JDBC-drivers. Iets wat in paragraaf 5.4.2 besproken is als een "best practice". Bij het testproject waarbij we overstapten naar een MS-Access database (9.3.4), waren we verplicht een JDBC-ODBC driver te gebruiken. De performance van deze driver is werkelijk onacceptabel en daarmee beamen we het advies om dit type driver te vermijden.

10.4.2 Verschillen tussen J2EE-implementaties

Omdat de J2EE-specificatie leveranciers zelf laat bepalen hoe ze verplichte functionaliteit implementeren, kunnen grote verschillen optreden in de performance en schaalbaarheid van de J2EE-servers. *ECperf* is een Enterprise JavaBeans benchmark, bedoeld om de schaalbaarheid en performance van J2EE-servers en containers te meten [B23]. De benchmark is ontwikkeld door Sun en een tiental leveranciers.
ECperf bestaat uit een specificatie en een "Kit" met alle benodigde code (voor de J2EE referentie implementatie). De gebruikte kit kent een aantal verplichte eigenschappen om de benchmark een realistisch systeem te maken en een showcase voor de gedistribueerde, transactiegeoriënteerde enterprise beans.
De eerste versie van ECperf was bedoeld voor J2EE 1.2 servers. De resultaten van de deelnemende leveranciers staan gepubliceerd op Internet[42]. De performance van de configuraties wordt gemeten in *Bbops / min,* wat staat voor *Benchmark Business OPerationsS per minute.* De resultaten bevatten ook informatie over de gebruikte configuratie en de bijbehorende kosten. Het is noemenswaardig om te vermelden dat Oracle in deze benchmark als beste presteert, gevolgd door de implementaties van respectievelijk IBM en BEA. Ook in de prijs/performance meting scoort Oracle het beste. De performanceverschillen tussen de configuraties lopen op tot een factor tien. Daarbij speelt de gebruikte hardware ook een rol.
Helaas staat versie 1.1 van ECperf, bedoeld voor J2EE 1.3 servers, niet toe dat de resultaten gepubliceerd worden.

10.4.3 J2EE vs .NET

Het is nuttig om te zien hoe de verschillende implementaties zich tot elkaar verhouden op het gebied van performance, maar minstens even interessant is de vergelijking tussen J2EE en haar concurrentie. Nu is er eigenlijk maar één grote concurrent van J2EE: Microsofts .NET. Waarom komt deze vergelijking alleen aan de orde bij "performance & schaalbaarheid"? Het antwoord daarop is simpel: performance en schaalbaarheid blijken de meest interessante

[41] De resultaten zijn een gemiddelde van honderd runs, om de initiële overhead van het connectiemanagement te kunnen verwaarlozen.
[42] Dit zijn alleen leveranciers van commerciële J2EE-implementaties. Open-source implementaties zoals JBoss staan niet bij de resultaten.

strijdwapens in de praktijk. Performance benchmarks geven met een enkel getal een oordeel. Vandaar dat hier ook de “marketing-oorlog” mee gevoerd wordt.
De benchmark-strijd tussen .NET en J2EE gaat met behulp van de “Pet Store Demo”. Dit was oorspronkelijk een J2EE-tutorial implementatie van Sun Microsystems, dat laat zien hoe de best practices van gedistribueerde applicaties toegepast zouden moeten worden (Appendix B: Core J2EE Patterns). In feite was het een “leer-tool”, die nu ook gebruikt wordt als “performace benchmark suite” [B22].
Om een lang verhaal kort te maken: Microsoft implementeerde haar eigen versie van de “Pet Store” met behulp van .NET, dat sneller was dan de geoptimaliseerde J2EE-versie van de *Middleware Company*. Echter, J2EE-aanhangers hadden kritiek op de “geoptimaliseerde” J2EE-versie, en sindsdien is er een eindeloos moddergevecht aan de gang tussen de beide kampen. Het is nog wat te vroeg om een uitspraak te doen over de ontwikkelplatformen op basis van de bestaande ervaringen. Voor een eerlijke vergelijking tussen de platformen zul je zelf een empirische test moeten uitvoeren.
Wat wel uit de strijd blijkt is dat het .NET-platform ook schaalbare applicaties kan leveren, iets wat voorheen niet mogelijk geacht werd vanuit de J2EE wereld. De Java-literatuur is het er dan ook wel over eens dat .NET een serieuze bedreiging is voor het J2EE-platform of dat zal worden. Op Windows-platformen lijkt het waarschijnlijk dat .NET de performancewinnaar zal worden [B22].

10.5 Betrouwbaarheid

10.5.1 High availability

Guy Damian, Allan Packer en Tom Daly testen in hun artikel “High Availability for J2EE Platform-Based Applications” [B20] de availability en failover mogelijkheden van een viertal J2EE-configuraties.

1. Een enkelvoudige J2EE-server
2. Een enkelvoudige webserver en een enkelvoudige EJB-server
3. Een enkelvoudige webserver en twee EJB-servers
4. Twee webservers en twee EJB-servers

Als applicatie gebruiken ze een iets aangepaste versie van de ECperf-Kit (paragraaf 10.4.2).
Uit de test blijkt dat het plaatsen van de webserver en de applicatieserver op aparte machines ervoor zorgt dat HTTP-sessiedata blijft bewaard bij een crash van de applicatieserver. Andersom, wanneer de webserver crasht, blijft de status van stateful session beans bewaard in de EJB-server.
Het artikel geeft verder aan dat serverclustering en load-balancing technieken een positieve invloed hebben op het failover percentage. Voor de exacte resultaten verwijzen we door naar het artikel zelf.

10.5.2 Persistentie

Bij alle testprojecten is gebruik gemaakt van “container managed persistentie”. De EJB-container is dan verantwoordelijk voor de mapping tussen de entity beans en de database. Enterprise beans gebruiken Java datatypen, terwijl een database zijn eigen formaten kent. De container moet weten hoe hij hoe hij de datatypen moet “omzetten”. De datatype mapping voor een specifieke database staat beschreven in de deployment descriptor. Als een database niet in de lijst is opgenomen moet de ontwikkelaar de datatype mapping toevoegen aan de deployment descriptor.
Het CMP-model is flexibel en makkelijk te gebruiken. Wanneer de datatype mapping gespecificeerd is, bestaat de rest uit “het betere klikwerk”. Met een druk op de knop kan het hele datamodel van de database worden geïmporteerd. Veranderingen in het datamodel zijn snel door te voeren in de entity beans (zie 10.3.4) en volgens de theorie is container managed persistentie ook sneller dan bean managed persistence. Weinig reden dus om geen container managed persistentie te gebruiken.

10.5.3 Transactiemanagement

Bij het gebruik van Enterprise JavaBeans krijg je transactiemanagement “cadeau”. Het invullen van de transactiesemantiek kan eenvoudig met behulp van deployment descriptors. De container zorgt voor de atomaire eigenschap van een transactie. Een vraag uit de praktijk is of J2EE goed kan omgaan met de isolatie-eigenschap van een transactie, vooral in combinatie met externe resource managers. Het is mogelijk om isolatielevels te definiëren, maar het is de vraag of dit voldoende is. Bij dit rapport is hier verder geen onderzoek naar gedaan.
De ondersteuning voor gedistribueerd transactiemanagement is niet verplicht. Dit is nog altijd het paradepaardje van dure commerciële implementaties. Open-source implementaties zoals JBoss zijn wel bezig dit te ontwikkelen.

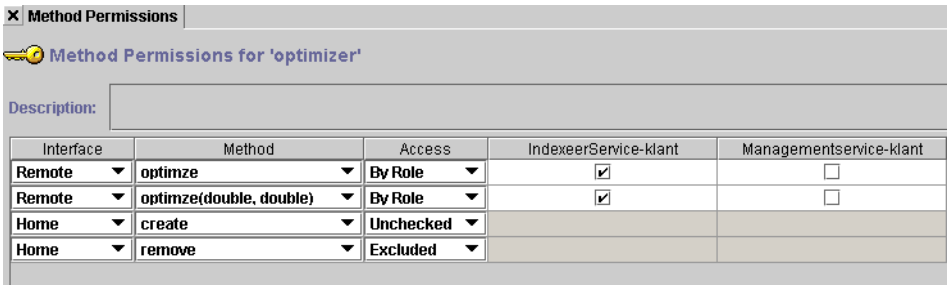
10.6 Beveiliging

10.6.1 Authenticatie

Voor web-clients worden drie authenticatiemechanismen aangeboden (zie paragraaf 7.2.1). Authenticatie voor applicatie-clients valt buiten de J2EE-specificatie. Daardoor is bij applicatie-clients de ontwikkelaar verantwoordelijk voor de implementatie van authenticatiemechanismen. Als je gebruik wilt maken van een web-authenticatiemechanisme, dan kun je dat in de deployment descriptor aangeven. Bij bijvoorbeeld “basic-authenticatie” krijgen gebruikers dan automatisch een inlogscherm te zien wanneer ze een beveiligde webpagina bezoeken.
Het definiëren van het beveiligingsmodel is aanbiederspecifiek. Voor het volledig definiëren van het beveiligingsmodel moeten meerdere deployment descriptors van de aanbieder aangepast worden. Ook is het aan de aanbieder om te bepalen hoe de gebruikersnamen en wachtwoorden worden opgeslagen. Dit kan bijvoorbeeld in een database of in een tekstbestand.

10.6.2 Autorisatie

Het definiëren van beveiligingsrollen en autorisatieregels kan met behulp van de ontwikkelomgeving. Voor het testproject hebben we twee rollen aangemaakt. Figuur 10-6 laat zien hoe de autorisatieregels voor de optimizer declaratief ingesteld kunnen worden.



Figuur 10-6: Declaratief vaststellen van autorisatieregels

Het declaratieve beveiligingsmodel van J2EE komt vooral bij de autorisatie tot zijn recht. We hebben niet onderzocht of het propageren van identiteitsgegevens naar bijvoorbeeld een informatiesysteem goed verloopt, maar binnen het J2EE-model is het toepassen van autorisatie geen kunst.

11 Conclusie & Discussiepunten

11.1 Inleiding

Op basis van de besproken theorie, de praktijkresultaten van de testprojecten en de ervaringen van anderen kunnen we een algemeen kwalitatief oordeel geven over het Java 2, Enterprise Edition ontwikkelplatform. Dit gebeurt in de volgende paragraaf aan de hand van de centrale probleemstelling en de opgestelde criteria voor enterprise-applicaties.
Door de omvangrijkheid van het J2EE-platform en gedistribueerde applicatieontwikkeling in het algemeen, is dit onderzoek te beperkt om een kwantitatieve uitspraak te kunnen doen. Het onderzoek naar het J2EE-model en gedistribueerde applicatieontwikkeling is met dit document dan ook niet afgerond. Ook roept het onderzoek nieuwe vragen op. Dit hoofdstuk sluit daarom af met een aantal discussiepunten die gebruikt kunnen worden als uitgangspunt voor verder onderzoek.

11.2 Conclusie

Voor de conclusie nemen we de centrale probleemstelling uit hoofdstuk 1 erbij.

Voldoet Java 2, Enterprise Edition als ontwikkelplatform voor enterprise-applicaties?

Om deze vraag gefundeerd te kunnen beantwoorden stelden we in hoofdstuk 2 een vijftal criteria voor enterprise-applicaties vast, die als leidraad dienden bij het verdere onderzoek. Het ontwikkelplatform moet namelijk de *mogelijkheid* bieden om applicaties te creëren die aan deze criteria voldoen. Paragraaf 11.2.1 geeft de conclusie over deze basisfunctionaliteit van J2EE.
De toegevoegde waarde van een ontwikkelplatform is minder concreet meetbaar dan de vereiste basisfunctionaliteit. Het wordt met name bepaald door de indruk die het platform achterlaat wanneer je er in de praktijk mee werkt. Paragraaf 11.2.2 geeft de conclusie over de toegevoegde waarde van het J2EE-platform aan de hand van de vragen uit hoofdstuk 1.

11.2.1 Basisfunctionaliteit

- ***Realisatie & Onderhoud***
  Het ontwikkelen van een gedistribueerde applicatie is complexer dan het ontwikkelen van een niet-gedistribueerde applicatie. Het J2EE-platform biedt ondersteuning door standaard Java API's te leveren, die een laag vormen boven aanbiederspecifieke API's. De verplichte Java-

programmeertaal is een elegante en robuuste taal met een hoog abstractieniveau.
De kracht van objectgeoriënteerd programmeren komt naar voren in de componentenmodel-architectuur van het J2EE-platform. Componenten kunnen onafhankelijk van elkaar worden vernieuwd en vervangen. Het J2EE-model levert ook een aantal diensten aan de componenten. Deze diensten implementeren theoretische concepten zoals transactiemanagement. Dankzij deze diensten kan een ontwikkelaar zich meer richten op de applicatielogica, dan op de systeemtechnische implementatie.
Het enkelvoudige applicatiemodel vermindert het heterogene karakter van gedistribueerde systemen, waardoor het koppelen van afzonderlijke applicatieonderdelen makkelijker wordt. De mogelijkheid om applicatiegedrag te sturen met behulp van "deployment descriptors" geeft ruimte voor specialisatie. Het assembleren en deployen van een enterprise-applicatie vereist geen wijzigingen in de code en hoeft daardoor niet door ontwikkelaars gedaan te worden.
In de praktijk blijkt dat J2EE een hoog initieel kennisniveau vereist. Dit heeft te maken met de omvangrijkheid van een ontwikkelplatform als J2EE. Het aantal diensten, API's en design patterns vereisen de nodige kennis voordat ze gebruikt kunnen worden. Ook kennis van Java en gedistribueerde applicatieontwikkeling in het algemeen is nodig.
De opstartfase wordt verder bemoeilijkt door de heterogeniteit van de J2EE-implementaties. Tools werken niet in bepaalde configuraties en het ontbreekt in de documentatie nog wel eens aan een eenduidige richtlijn. Dat laatste geldt met name voor concrete documentatie voor een specifieke configuratie. Overigens kunnen tools, wanneer ze goed functioneren, een heleboel werk uit handen nemen van de ontwikkelaar.
Al met al kan gesteld worden dat het J2EE-platform veel steun biedt bij het ontwikkelen van enterprise-applicaties. Zowel bij de realisatie als het onderhoud. Echter, er blijft een hoop expertise nodig voor het overzicht en ontwikkelaars zullen te maken hebben met opstartproblemen bij een nieuwe configuratie.

- ***Openheid***
  Een J2EE-applicatie kan verschillende typen clients hebben, variërend van web-browsers tot mobiele telefoons. Door het gebruik van design patterns en application frameworks blijft een applicatie in deze situatie onderhoudbaar dankzij een goede scheiding tussen presentatie en businesslogica.
  Java wordt uitgevoerd op een Java virtuele machine. Deze virtuele machine bestaat voor meerdere platforms, wat Java in grote mate platformonafhankelijk maakt. Doordat J2EE-containers ook op een Java virtuele machine draaien, zijn zij eveneens niet gebonden aan een specifiek platform.
  J2EE is geen product maar een specificatie. Er bestaan zo'n vijftig implementaties van de J2EE-specificatie van verschillende leveranciers. Hierdoor zijn ontwikkelaars niet gebonden aan een specifieke aanbieder.
  Voor de communicatie met andere systemen biedt J2EE een aantal mogelijkheden. Voor synchrone communicatie kan gebruik gemaakt

worden het CORBA IIOP-protocol, terwijl voor asynchrone communicatie een willekeurige JMS-provider gebruikt kan worden. Met de ondersteuning van web services zal de integratie met andere systemen nog eenvoudiger worden. Web services vereisen geen complexe infrastructuur en maken gebruik van standaarden als XML en TCP/IP.
Integratie met bestaande informatiesystemen gaat met behulp van connectors. Voor elk informatiesysteem is één connector nodig. Communicatie met relationele databases kan met behulp van de populaire JDBC API. Bijna elke database heeft een JDBC-driver. Java-applicaties kunnen zo op een standaard manier communiceren met verschillende databases. Het verwisselen van de driver is voldoende om over te stappen naar een andere database.
Het J2EE-model maakt gebruik van vele standaard interfaces en protocollen, waardoor integratie tussen componenten van verschillende aanbieders mogelijk en vaak ook makkelijk is. Op dit punt scoort J2EE dan ook heel goed.

- ***Performance & Schaalbaarheid***
  Specifieke J2EE-performancevoordelen zijn moeilijk te noemen. Het genereren van dynamische webpagina's gaat sneller met behulp van JSP dan met CGI. Er bestaat de mogelijkheid om load-balancing toe te passen bij servers, maar dit is geen onderdeel van de J2EE-specificatie.
  De belangrijkste performanceafweging op de applicatieserver is het wel of niet gebruiken van Enteprise JavaBeans. Deze techniek is vooral bedoeld voor het maken van schaalbare applicaties, en kan door haar architectuur de performance van kleinschalige applicaties verminderen.
  Uit benchmarks voor performance en schaalbaarheid blijkt dat de performance van verschillende J2EE-implementaties nogal uiteen lopen. Dit komt omdat de J2EE-specificatie niet vaststelt hoe de verplichte functionaliteit geïmplementeerd moet zijn. Performance is daarmee een aandachtspunt bij de keuze van een J2EE-productleverancier.
  Het is moeilijk om een goed oordeel te vellen over de performance van het J2EE-platform ten opzichte van haar grootste concurrent .NET. Dit heeft te maken met het feit dat het .NET-platform nog erg nieuw is. De informatie hierover op het Internet is te bevooroordeeld om hier een beroep op te doen. Wel kan uit deze documentatie afgeleid worden dat het vooral het J2EE-platform is, dat zich zal moeten bewijzen op het gebied van performance en schaalbaarheid.

- ***Betrouwbaarheid***
  De beschikbaarheid van een J2EE-systeem kan verhoogd worden door het gebruik van redundante clusters. Dit is helaas geen onderdeel van de J2EE-specificatie en daardoor ook niet gestandaardiseerd. Wel bieden de grote J2EE-aanbieders allemaal ondersteuning voor clustering.
  Op het gebied van fouttolerantie biedt J2EE persistentiemechanismen en transacties. Persistentie heeft te maken met het synchroniseren van data in objecten en data in een database. Entity beans vormen een OO-laag voor de benadering van een database. De container coördineert automatisch de

synchronisatie tussen de status van de bean en de data in de database[43]. Met behulp van grafische tools is de mapping tussen de entity beans en het datamodel eenvoudig te realiseren. Ook zijn aanpassingen in het datamodel hiermee snel door te voeren.
De Enterprise JavaBeans architectuur biedt transactiemanagement als een standaard dienst aan. De transactiesemantiek kan met transactieattributen declaratief ingesteld worden. Daarmee kunnen onder andere de transactiegrenzen bepaald worden. De EJB-container kan de verantwoordelijkheid dragen voor de commit en roll-back beslissingen. De standaard ondersteuning voor transacties is een krachtig hulpmiddel, zeker omdat ook gedistribueerde transacties mogelijk zijn.

- ***Beveiliging***
  De literatuur onderscheidt vijf beveiligingsmaatregelen (security services): authenticatie, autorisatie, data confidentiality, data integriteit en accountability. J2EE biedt een uniform beveiligingsmodel aan, dat ondersteuning biedt voor de eerste vier beveiligingsmaatregelen. Het gebruik van auditing als mechanisme voor accountability is op dit moment geen onderdeel van de J2EE-specificatie.
  Authenticatiemechanismen voor webapplicaties worden binnen J2EE standaard aangeboden. De implementatie van het beveiligingsmodel is hierbij aanbiederspecifiek. J2EE biedt de mogelijkheid om identiteiten te propageren naar enterprise informatiesystemen en J2EE-servers.
  J2EE kent voor de autorisatie een declaratief beveiligingsmodel. De beveiligingsstructuur kan daarmee buiten de applicatie zelf worden gedefinieerd. Dit maakt het mogelijk om de autorisatieregels na de applicatieontwikkeling vast te stellen of te wijzigen. Naast declaratieve beveiliging ondersteunt J2EE ook geprogrammeerde beveiliging. Deze vorm van beveiliging is nuttig wanneer declaratieve beveiliging alleen niet voldoende is.
  J2EE-containers implementeren de data confidentiality en integrity mechanismen op de transportlaag. Deployers configureren de containers om aan te geven voor welke netwerken en berichten deze mechanismen toegepast moeten worden. Het verkrijgen van sleutelcertificaten valt buiten het J2EE-model.

11.2.2 Toegevoegde waarde

We gebruiken de vragen uit hoofdstuk 1 om een beeld te geven van de toegevoegde waarde van J2EE als ontwikkelplatform.

Welk abstractieniveau biedt het J2EE-platform bij het ontwikkelen van enterprise-applicaties?

Een platform als J2EE slaat de brug tussen de theorie en de praktijk. Veel van de geïmplementeerde theoretische concepten zijn niet nieuw. De algoritmen en ideeën met betrekking tot deze concepten bestonden al in de jaren tachtig of

[43] Het is ook mogelijk om de synchronisatie handmatig te doen.

begin jaren negentig [A2]. Maar het feit dat ze als dienst geïmplementeerd zijn en ontwikkeld tot een standaard is wel nieuw. Dit komt ook overeen met de gedachte dat er een cyclisch verband bestaat tussen de ervaring met bestaande systemen, vervolgens het formuleren van nieuwe concepten en eisen aan systemen en het ontwikkelen van deze nieuwe systemen [A2]. Samengevat kan gesteld worden dat de meeste concepten waarover je las in begin jaren negentig, nu aangeboden worden als dienst in het J2EE-ontwikkelplatform.

Hoe flexibel is dit ontwikkelplatform voor de verschillende soorten applicaties?

Het J2EE model schrijft geen verplichte architectuur voor. Met name de multitier-structuur maakt het mogelijk om met J2EE meerdere applicatiescenario's te ondersteunen. Dit kan variëren van een webapplicatie tot een business-to-business systeem.
Op "openheid" scoort het J2EE-model goed en dat is belangrijk voor de integratie met andere systemen. J2EE complementeert daarbij het bestaande systeem in plaats van de functionaliteit ervan te kopiëren. De ondersteuning van web services zal communicatie met andere systemen verder vergemakkelijken.

Hoe groot is de draagkracht van J2EE in de markt?

Op dit moment is J2EE het meest volwassen platform. De ruim vijftig leveranciers doen elk hun deel van de marketing, evenals de rest van de community. Dit heeft geleid tot een groot aanbod aan tools, implementaties en documentatie voor het J2EE-platform. De komst van open-source implementaties heeft de populariteit van het platform verder vergroot. Door de investering van met name de grote aanbieders, zal de ondersteuning voor J2EE niet "zomaar" verdwijnen.

Wat zijn de toekomstige verwachtingen rondom J2EE?

Alle aandacht is nu gevestigd op web services. Met name de business-to-business integratie zal een belangrijke rol gaan spelen in de toekomst, omdat deze integratie met web services veel eenvoudiger wordt dan voorheen.
Anderzijds probeert Sun steeds meer zaken op te nemen in de specificatie, zoals een standaardisering van het deploymentproces. Dit maakt het makkelijker voor ontwikkelaars om gebruik te maken van een andere leverancier. Ook standaarden op het gebied van bijvoorbeeld clustering en beveiliging ontbreken nog. Er is echter ook een commercieel aspect. Hoe meer Sun zal vastleggen in de J2EE-specificatie, hoe minder ruimte leveranciers hebben om zich te onderscheiden van hun concurrenten. Bovendien eisen open-source implementaties zoals JBoss een steeds groter deel van de taart op. Een aantal bedrijven is vanwege de harde concurrentie gestopt met hun J2EE-implementatie. Naar verwachting zal in de toekomst het aantal aanbieders eerder afnemen dan toenemen.

11.3 Discussiepunten

Als er één ding duidelijk is geworden bij het onderzoek naar het Java 2, Enterprise Edition ontwikkelplatform, dan is dit wel de omvangrijkheid ervan. Dit document heeft kort de onderliggende technieken besproken, terwijl er over de meeste technieken complete boeken geschreven zijn. Ook zijn sommige veelbelovende technieken en standaarden op dit moment nog in ontwikkeling, wat ze interessant maakt voor verder onderzoek. Er zijn een aantal aandachtspunten die in ieder geval wat meer aandacht verdienen.

.

11.3.1 J2EE in de praktijk

Het onderzoek naar het J2EE-platform in de praktijk is beperkt in dit rapport. We hebben de theorie 'getoetst', maar dit heeft vooral laten zien dat J2EE de benodigde basisfunctionaliteit kan leveren. Er zijn meer praktijkervaringen nodig om een realistisch beeld te krijgen van de toegevoegde waarde van het platform. Dit kan door het bouwen van een volledige enterprise-applicatie, die aan de "design goals" van dit rapport voldoet. Een andere optie is het evalueren van bestaande enterprise-applicaties binnen organisaties. Hierbij is het interessant om ook een vergelijking te maken tussen applicaties die wel en die niet met het J2EE-platform gerealiseerd zijn. Wat kun je zeggen over het implementatietraject van de applicatiecriteria? Hoe belangrijk is het gebruik van een enkelvoudig applicatiemodel zoals J2EE bij gedistribueerde systemen in de praktijk?

11.3.2 J2EE vs. .Net

In dit document hebben we maar één marktspeler beschreven. Microsoft lanceerde onlangs de eerste "volwassen" .NET-server. Door de grote invloed van Microsoft en de veelbelovende resultaten van .NET, zal het een geduchte concurrent worden van J2EE.
De twee platformen zullen volgens de verwachtingen bijna de hele applicatieserver-markt in handen krijgen. En het gaat hierbij om een miljardenmarkt. Dat maakt een grondige vergelijking tussen deze reuzen erg interessant.
Bij een dergelijke vergelijking zijn naast de technische aspecten ook de bedrijfsaspecten van belang. Waarom kiest een organisatie voor een bepaald platform? Met welke (verborgen) kosten heeft een organisatie te maken?

11.3.3 Web services

Web services zijn de toekomst voor het integreren van systemen. Dat schrijft de literatuur en roepen de ICT-bedrijven. Ze lijken de ideale oplossing voor business-to-business scenario's, geavanceerde mobiele telefoonapplicaties, en "personalized" applicaties [C6]. Het feit dat ze bij het rijtje met ontwikkelingen zoals Java en XML gezet worden, schept hoge verwachtingen.
In dit document hebben web services een ondergeschoven rol gekregen onder "openheid", maar het is een technologie op zich. De hype spreekt van SOA-applicaties, waarbij SOA staat voor *Service Oriented Architecture*. Mensen moeten niet meer denken in objecten, maar in volledige diensten als

bouwstenen van een systeem. Kan dit als de opvolger van de OO-gedachte gezien worden?

11.3.4 Gevolgen van enterprise-applicatie ontwikkelplatformen

Een wat meer filosofisch discussiepunt is de vraag welke gevolgen de evolutie van ontwikkelplatformen zoals J2EE en .NET op de ontwikkeling van enterprise-applicaties heeft.
J2EE biedt de gebruiker nu een homogeen applicatiemodel en een bepaald abstractieniveau, waardoor een ontwikkelaar niet alles meer hoeft te weten van de onderliggende systeemarchitectuur. De komst van een concurrent als .NET en de verdere ontwikkelingen op dit gebied zullen het ontwikkelen van enterprise-applicaties steeds makkelijker maken. Dit geeft te denken.
Is het straks voor een ontwikkelaar nog duidelijk wat er "achter de schermen" gebeurt? Kunnen "leken" straks complexe gedistribueerde applicaties bouwen? Wat voor gevolgen heeft dit voor de kwaliteit van de applicaties? Verschuift een deel van de verantwoordelijkheid hiermee van de ontwikkelaar naar het ontwikkelplatform?

Appendix A: J2EE Required API's

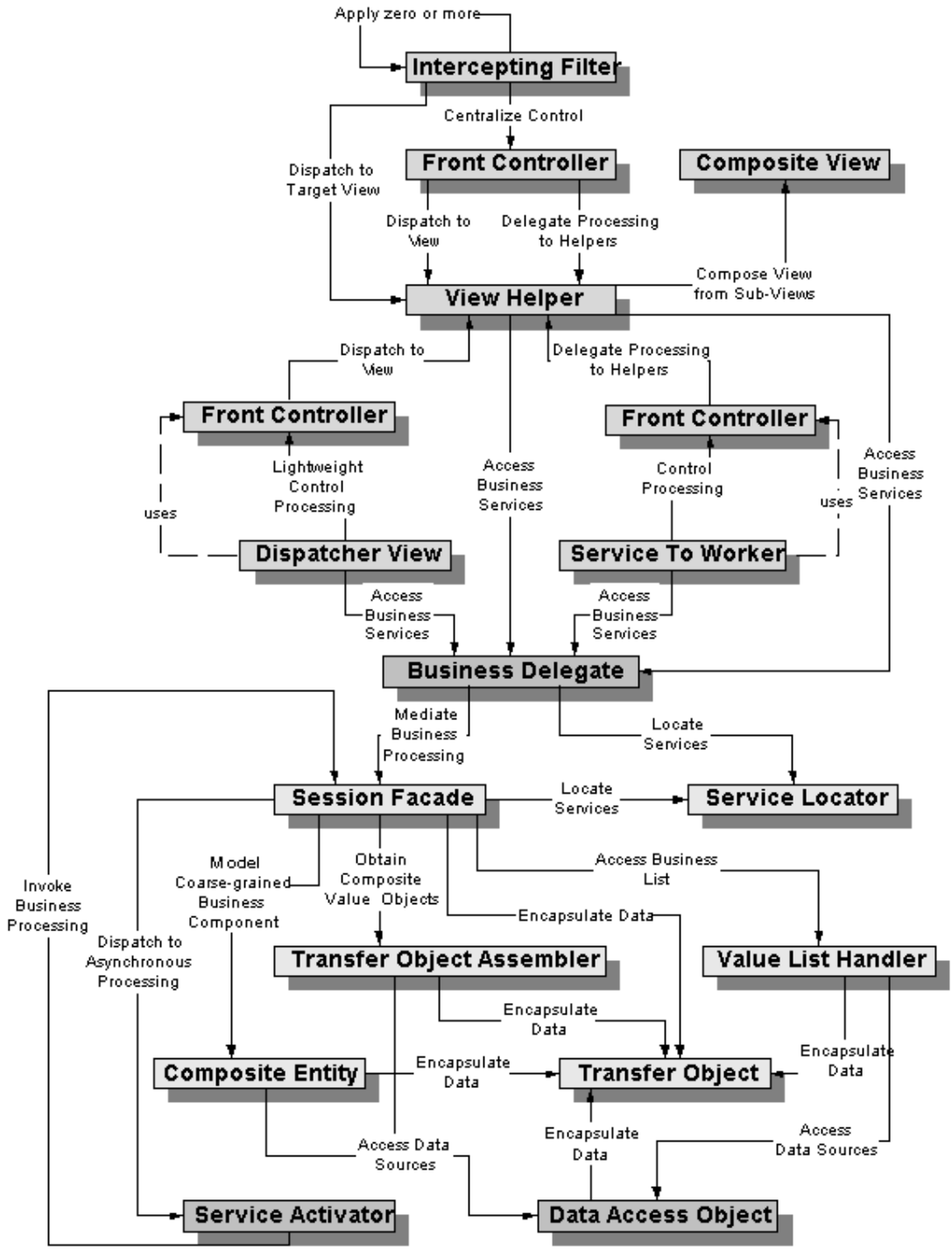
Naam API	Acroniem	Versie
Enterprise JavaBeans	EJB	2.0
Servlets		2.3
JavaServer Pages	JSP	1.2
HTTP & HTTPS		
Java RMI-IIOP		
Java RMI-JRMP		
Java IDL		
Java DataBase Connectivity	JDBC	2.0
Java Naming and Directory Interface	JNDI	1.2
JavaMail API		1.2
Java Activation Framework	JAF	1.0
Java Message Service	JMS	1.0.2
Java API for XML Parsing	JAXP	1.1
J2EE Connection Architecture	JCA	1.0
Java Authentication and Authorization Service	JAAS	1.0
Java Transaction API	JTA	1.0.1

Tabel 6: J2EE 1.3 API's

Naam API	Acroniem	Versie
Enterprise JavaBeans	EJB	2.1
Servlets		2.4
JavaServer Pages	JSP	2.0
HTTP & HTTPS		1.1
Java RMI-IIOP		
Java RMI-JRMP		
Java IDL		
Java DataBase Connectivity	JDBC	3.0
Java Naming and Directory Interface	JNDI	1.2
JavaMail API		1.3
Java Activation Framework	JAF	1.0
Java Message Service	JMS	1.1
Java API for XML Parsing	JAXP	1.2
J2EE Connection Architecture	JCA	1.5
Java Authentication and Authorization Service	JAAS	1.0
Java Transaction API	JTA	1.0.1
Web Services for J2EE		1.1
Java API for XML-based RPC	JAX-RPC	1.0
SOAP with Attachments API for Java	SAAJ	1.1
Java API for XML Registries	JAXR	1.0
J2EE Management API		1.0
Java Management Extensions	JMX	1.2
J2EE Deployment API		1.1
Java Authorization service provider Contract for Containers	JACC	1.0

Tabel 7: J2EE 1.4 API's

Appendix B: Core J2EE Patterns



Appendix C: J2EE Licentiehouders

Licentiehouder	J2EE-applicatieserver	Compatibiliteit
ATG	Dynamo Application Server	1.3
BEA Systems	WebLogic Server	1.3
Borland Corp.	Enterprise Server	1.3
BroadVision		
Brokat		
Cape Clear Software		
Compaq		
DataDirect Technologies		
Fujitsu	INTERSTAGE Application Server	1.3
Fujitsu Siemens Computers		
Hewlett-Packard	Total-e-Server	1.2
Hitachi	Cosminexus Server	1.2
IBM	WebSphere Application Server	1.3
Interworld		
IONA Technologies	Orbix E2A Application Server	1.3
Macromedia	JRun	1.3
MERANT		
NEC	WebOTX	1.3
Nokia		
Oracle Corporation	9i Application Server	1.3
Persistence Software, Inc.	PowerTier Application Server	1.2
Pramati	Server & Studio	1.3
SAP	Web Application Server	1.2
SAS Institute, Inc.	AppDev Studio	1.3
Secant		
SilverStream	eXtend App Server	1.3
Sonic Software Corporation		
SpiritSoft	SpiritSoft	1.3
SUN Microsystems	ONE Application Server	1.3
Sybase, Inc.	EAServer	1.3
TIBCO Software Inc.		
Tmax Soft	JEUS	1.3
Trifork Technologies	Application Server	1.3

Tabel 8: J2EE licentiehouders en producten

Bron: SUN microsystems website http://www.java.sun.com/licensees

Literatuurlijst

Boeken

[A1] Schneider, J. e.a., Special Edition using Enterprise Java
1st edition, Que, jun 1997

[A2] Coulouris, G. e.a., *Distributed Systems – Concepts and Design*
2nd edition, Addison Wesley, 1994

[A3] Monson-Haefel, R., *Enterprise JavaBeans*
3rd edition, O'Reilly & Associates, sep 2001

[A4] Fowler, M. en Scott, K., *UML beknopt*
Addison Wesley, jun 2000

[A5] Horstmann, C.S. en Cornell, G., *Core Java 2, Volume I – Fundamentals*
5th edition, Sun microsystems Press, jan 2001

[A6] Horstmann, C.S. en Cornell, G., *Core Java 2, Volume II – Advanced Features*
5th edition, Sun microsystems Press, dec 2001

[A7] Kassem, N. e.a., *Designing Enterprise Applications with the J2EE Platform*
2nd edition, Sun microsystems Press, dec 2001

[A8] O'reilly Java Authors, *Java Enterprise Best Practices,*
O'Reilly & Associates, dec 2002

[A9] Bergsten H, *JavaServer Pages*
2nd edition, O'Reilly & Associates, sep 2002

[A10] Stallings W., *Network Security Essentials, Applications and Standards*
2nd edition, Prentice Hall, jan 2003

[A11] Monson-Haefel R. e.a., *Java Message Service*
1st edition, O'Reilly & Associates, jan 2001

[A12] Alur D. e.a, *Core J2EE Patterns: Best Practices and Design Strategies*
1st edition, Prentice Hall, jun 2001

[A13] Farley J. e.a, *Java Enterprise in a Nutshell: A Desktop Quick Reference*
2nd edition, O'Reilly & Associates, apr 2002

[A14] Beasley, J.E., N. Meade en T-J. Chang: *Index tracking*
The Management School, Imperial College, London, 1999

[A15] Meade, N. en G.R. Salkin: *Index funds - construction and performance measurement*
Journal of the Operational Research Society, 1989

[A16] *Enterprise JavaBeans Developer's Guide*
Borland Software Corporation, 2001

World Wide Web

[B1] Sun Microsystems, *Simplified Guide to the Java 2 Platform, Enterprise Edition*
http://java.sun.com/j2ee, sep 1999

[B2] OMG, *Unified Modeling Language Specification version 1.3*
1st edition, http://www.omg.org, mrt 2000

[B3] Pawlan, M., The J2EE Tutorial version 1.3
draft 5, http://www.javasoft.com, apr 2001

[B4] The Java 2, Enterprise Edition Developer's Guide
versie 1.2.1, http://www.java.sun.com/j2ee , mei 2000

[B5] DeMichiel L. e.a., Enterprise JavaBeans Specification, Version 2.0
final release, http://www.java.sun.com/ejb, aug 2001

[B7] Shannon, B., *Java 2 Platform Enterprise Edition Specification version 1.4*
proposed final draft 3, http://www.javasoft.com, apr 2003

[B8] Sun Microsystems, *JDBC Data Access API overview*
http://www.java.sun.com/jdbc, 2002

[B9] Sun Microsystems, *JDBC 2.1 API Specification*
versie 1.1, http://www.java.sun.com/j2ee/docs, okt 1999

[B10] Sun Microsystems, *JDBC 3.0 Specification*
final release, http://www.java.sun.com/j2ee/docs, okt 2001

[B11] Sun Microsystems, *J2EE Connector Architecture Specification 1.5*
proposed final draft 2, http://www.java.sun.com/connector, jul 2002

[B12] Reinshagen, D., *Connect the enterprise with the JCA, Part 1*
http://www.javaworld.com, nov 2001

[B13] Sun Microsystems, *J2EE Design Patterns*
http://www.java.sun.com/blueprints/patterns, 2002

[B14] Kawamote, D. e.a., *Web services market up for grabs*
http://www.businessweek.com, feb 2003

[B15] Monson-Haefel, R., *Monson-Haefel's Guide to Enterprise JavaBeans*
http://www.theserverside.com, jun-dec 2002

[B16] Christensen, E. e.a., *Web Services Description Language (WDSL) 1.1*
http://www.w3.org/TR/wsdl, mrt 2001

[B17] Christensen, E. e.a., *Simple Object Access Protocol (SOAP) 1.1*
http://www.w3.org/TR/SOAP, mei 2000

[B18] Shannon, B., *Java 2 Platform Enterprise Edition Specification version 1.3*
final release, http://www.java.sun.com/j2ee, jul 2001

[B19] Kang, A., *J2EE clustering*
http://www.javaworld.com, feb 2001

[B20] Guy, D. e.a., *High Availability for J2EE-platform-based Applications*
http://java.sun.com, jan 2002

[B21] LaMonica, M., *Java servers feel the open-source heat*
http://www.businessweek.com, feb 2003

[B22] McMillan, R., *Two worlds collide*
http://www.javaworld.com, jan 2003

[B23] Sun Microsystems,*ECPerf Specification*
final release, http://www.java.sun.com/j2ee/ecperf, apr 2002

Overige literatuur

[C1] Thomas, A., *Java 2 Platform, Enterprise Edition:Ensuring Consistency, Portability and Interoperability*
Patricia Seybold Group, 1999

[C2] Ford N., *Strutting your stuff*
Java Developers Journal, nov 2001

[C3] van Draat, L.F., *Index tracking met fondsselectie en transactiekosten*
Vrije Universiteit Amsterdam, aug 2000

[C4] Walsh, A.E., *J2EE 1.4 Web Services*
Dr. Dobb's Journal, apr 2003

[C5] *Getrouwtrek om standaarden web services*
Automatiseringsgids, 28 mrt 2003

[C6] Yuan, M.J., *XMLSecurity for Web Services*
Univeristy of Texas at Austin, nov 2002

[C7] Vawter, C. e.a., *J2EE vs. Microsoft .NET*
The Middleware Company, jun 2001

[C8] de Boer, R.C., *A Generic Architecture for Fusion-Based Intrusion Detection Systems*
Erasmus Universiteit Rotterdam, nov 2002