In dit artikel kijkt Bert Ertman naar de implicaties die de introductie van Enterprise JavaBeans 3.0 zal hebben voor de manier waarop we in de nabije toekomst Java Enterprise applicaties zullen ontwikkelen. De beschouwing begint met een blik terug in het verleden. Vervolgens analyseert de auteur de huidige stortvloed van kritiek op de standaard.

Enterprise JavaBeans 3.0 komt eraan

Terugblik en analyse

Om de huidige situatie te begrijpen, is het goed om eens terug te kijken naar hoe die eigenlijk is ontstaan. De geschiedenis van dit verhaal begint in 1997 waar eigenlijk voor de eerste keer sprake was van de term 'Enterprise JavaBeans'. Dat is ruim twee jaar voordat de eerste J2EE-standaard het levenslicht zag. Er was in die tijd een verbeten strijd gaande tussen de objectmodellen CORBA (OMG), JavaBeans en ActiveX (Microsoft). In navolging van de component based gedachte introduceerde iedere leverancier tevens zijn eigen componenten-framework. CORBA introduceerde een dienstenmodel rondom de Object Request Broker en Microsoft kwam met het Component Object Model, kortweg COM, met de Microsoft Transaction Server (MTS) als runtime-omgeving. De eerste versie van Enterprise JavaBeans was, om het netjes te zeggen, heftig geïnspireerd door beide concurrenten. Het container/server-model was geleend van MTS en het dienstenmodel van CORBA. Enterprise JavaBeans was oorspronkelijk bedacht met drie doelstellingen. Ten eerste als framework (API) voor het implementeren van componenten in Java. Ten tweede als framework voor het implementeren van Object/Relational Mapping en de derde oorspronkelijke doelstelling van Sun was om een nieuwe markt te openen voor enterprise-componenten in navolging van het succes van vergelijkbare markten met bijvoorbeeld ActiveX-componenten. Dit laatste bleek echter een grote vergissing. De ActiveX componenten markt die zo succesvol was, bleek een markt voor GUI-componenten die je in ieder project en over totaal verschillende soorten opdrachten heen, zo kon hergebruiken. In de praktijk bleek dat vrijwel niemand business logica of domeincomponenten als 'Klant' of 'Order' over projecten heen wilde hergebruiken. De enterprise-componentenmarkt stierf daarmee een zachte dood, of heeft misschien wel niet eens bestaan.

Kritiek op Enterprise JavaBeans Over het onderwerp Object/Relational Mapping (ORM) wordt al jarenlang heftig gediscussieerd. De discussie gaat meestal over de moeizame mapping tussen relationele databases en de objectgeoriënteerde wereld en beperkt zich dus niet alleen tot EJB, of Entity Beans. In de afgelopen jaren heb ik tijdens het verzorgen van trainingen over dit onderwerp en in talloze architectuur- en design-discussies met klanten en collega's vaak de vraag gehoord: "Zijn Entity Beans eigenlijk wel bruikbaar?" Een terechte vraag die diverse malen met 'nee' moest worden beantwoord voor dat specifieke project, of die specifieke toepassing. Het succes van EJB wordt in grote mate bepaald door de manier waarop je het gebruikt. Dit geldt in het bijzonder voor Entity Beans. Het simpelweg afwijzen van de gehele EJB-specificatie kan me in zekere mate ergeren. Laten we vooral niet vergeten dat een groot gedeelte van alle ophef voornamelijk door misbruik van de technologie wordt veroorzaakt. Voorbeelden hiervan zijn het toepassen van simpele 1-op-1 mappings van tabellen naar Entity Beans, het rechtstreeks benaderen van Entity Beans vanuit client-technologie, of het willens en wetens gebruiken van remote interfa-

ces terwijl er totaal geen noodzaak voor distributie in de applicatie bestaat. Niet voor niets werd aan JavaBeans de toevoeging 'Enterprise' meegegeven. We hebben het hier over toepassingen waarbij honderden of duizenden gebruikers tegelijkertijd actief zijn in een heftig gedistribueerd systeem wat zeer robuust en fouttolerant moet zijn. Kortom, het klinkt niet als een beschrijving van applicaties die je doorgaans ziet. Misschien ligt een gedeelte van de schuld aan het verkeerd gebruiken van EJB's ook wel bij de evangelisten van Sun zelf. Immers, jarenlang hebben zij de EJB-technologie gepusht en werden kleine, simpele applicaties als voorbeeld gebruikt. Zo is er de beruchte Java Pet Store, een geliefd doelwit van bespotting in benchmarks en in vergelijkingen met bijvoorbeeld Microsoft-technologie.

Natuurlijk zijn er dingen voor verbetering vatbaar aan de huidige versie van de Enterprise JavaBeans standaard. Wat ooit als een mooie oplossing is gelanceerd, is op basis van voortschrijdend inzicht of veranderende omstandigheden aan evolutie onderhevig. Bekijk de EJB-specificatie echter wel met het respect die deze verdient. Ikzelf ben in ieder geval van mening dat het doel waarvoor EJB's oorspronkelijk zijn bedacht, realistisch was en nog steeds is, zij het voor een beperkt type applicaties.

Alternatieven Vanwege aanhoudende kritiek, in het bijzonder op Entity Beans, ontstond de behoefte aan alternatieven, zoals Hibernate, TopLink en JDO. Het bijzondere aan JDO ten opzichte van de andere frameworks is dat het ook een standaardisatie is die binnen het Java Community Process (JSR-12) heeft plaatsgevonden. Daarmee werd dus een tweede 'echte' persistency-standaard voor het Java-platform gecreëerd. Niet alleen voor persistency ontstonden er alternatieven, ook de plaats van EJB in het applicatie-framework werd ter discussie gesteld, bijvoorbeeld door het Spring-framework. De zogenaamde 'rebel frameworks' bieden weliswaar een alternatief met een verfrissende, nieuwe aanpak, maar zijn ook niet meer dan dat: een populaire, maar niet gestandaardiseerde manier van applicatie-ontwikkeling. Daarmee wil ik zeker niet beweren dat je ze daarom links moet laten liggen, kijk maar naar het succes wat Struts (ook geen standaard) heeft. Waak er echter voor om zonder meer de hype rond deze frameworks te volgen. Bepaal voor iedere specifieke situatie wat de beste oplossing is.

De moord op JDO Noem het een complottheorie, maar een feit is dat een aantal van de grote applicatieserver-vendors niet al te blij was met het succes van JDO en de dalende belangstelling voor Entity Beans. Een verklaring hiervoor is misschien dat de runtime-omgeving voor Entity Beans, de EJB-container, het hart vormt van hun enterprise application server producten die nogal wat omzet genereren. Zij zitten er zeker niet op te wachten dat een eenvoudige webcontainer in combinatie met JDO deze inkomsten teniet zou doen. Deze redenering zou heel goed een verklaring kunnen zijn voor het stemgedrag van bijvoorbeeld IBM, Oracle en BEA in de stemronde waarin over de toekomst van JDO 2.0 werd beslist. De officiële lezing van het verhaal is dat de beschikbaarheid van twee standaarden voor hetzelfde doel de transparantie van het Java-platform zou schaden. Of die lezing nu klopt of niet, de tegenstemmen dwarsboomden de toekomst van JDO grotendeels en daarmee is de JDO-specificatie nu zo goed als ten dode opgeschreven. De weg naar EJB 3 was daarmee vrij.

EJB 3 Mijn eerste kennismaking met EJB 3.0 was tijdens de JavaOne van 2003, vrij kort na het formeren van de JSR-220 expert group. De voorlopige plannen die daar werden gepresenteerd waren nog lang niet zo ingrijpend als het uiteindelijke resultaat zoals geschetst in de huidige proposed final draft uit december 2005, maar dat er iets zou gaan veranderen, dat was wel duidelijk. De belangrijkste doelstelling van EJB 3.0 is om het ontwikkelen van enterprise Java-applicaties fors te versimpelen en, na het JDO-debacle, het creëren van een gestandaardiseerde en algemeen geaccepteerde Java Persistence API. De EJB-specificatie bestaat hierdoor feitelijk uit twee onderdelen: de zogenaamde 'Simplified Components' specificatie en de Java Persistence API. Eerstgenoemde is een sterk versimpeld contract tussen clients en containers, dat veel meer in het voordeel van de ontwikkelaar is uitgevallen dan in het verleden. De laatstgenoemde beperkt zich niet specifiek tot EJB, maar is ook bruikbaar buiten de context van een container of applicatieserver. Het is niet mijn bedoeling om in dit artikel een in-depth technische introductie van EJB 3 te geven. In het kader 'Referenties' vindt u een aantal nuttige links naar een meer gedetailleerde technische kennismaking met EJB 3.0.

Java 5 Enterprise JavaBeans 3.0 heeft een grote afhankelijkheid met Java 5 in verband met het toepassen van metadata annotations voor het overbodig maken van expliciete configuratie met deployment descriptors. Zoals gezegd is de belangrijkste doelstelling van EJB 3.0 het versimpelen van het ontwikkelen van enterprise componenten (Ease-of-Development). Dat is te realiseren door gebruik te maken van de nieuwe Java 5 Annotations implementatie. Ook wordt er gebruik gemaakt van andere nieuwe taalfeatures, bijvoorbeeld Generics voor het implementeren van relaties. Deze afhankelijkheid met Java 5 technologie zou wel eens een mogelijke hobbel in de acceptatie van Java EE 5 kunnen veroorzaken. De realiteit is namelijk dat er op dit moment nog maar weinig omgevingen op Java 5

draaien. Hier zit een wederzijdse afhankelijkheid. De belangrijkste reden dat er nog niet overal Java 5 draait, is omdat er nog vrijwel geen grote applicatieserver-releases zijn die Java 5 ondersteunen. Voor veel applicatieservers zal Java 5-ondersteuning pas komen in een Java EE 5 compatibele release van het product. Dit betekent dus zowel een upgrade van de taal, als de enterprise runtime-omgeving. Een vrij heftige stap, die niet door alle beheerafdelingen van IT-organisaties met veel gejuich in een punt nul-versie zal worden ontvangen.

EENVOUDIGER? Een terugkerend element in discussies over EJB-technologie is het feit dat de productiviteit van ontwikkelaars ernstig zou lijden onder de mate van configuratie en boilerplate die noodzakelijk zijn voor het fabriceren van een EJB. Het is echter de vraag of dit probleem nu opgelost is. Op het eerste gezicht hoeft er in ieder geval veel minder expliciet geconfigureerd te worden en is de boilerplate-code grotendeels verdwenen, of verhuisd naar annotations. Het geheim hiervan zit hem in de verzameling default configuratiekeuzes die de expert-group heeft voorgeschreven. 'Configuration by exception', zoals de expertgroup het noemt, komt in ieder geval ten gunste aan de productiviteit. Deze zal voor gemiddeld gebruik ongetwijfeld omhoog gaan, maar zijn de implicaties van defaultkeuzes daarmee gemakkelijker te overzien? Dat is een lastige vraag. Onder de motorkap blijft EJB een complex verhaal. Als ontwikkelaar hoef je om er in eerste instantie mee aan de slag te kunnen in ieder geval veel minder details vanaf te weten dan in het verleden. Het te eenvoudig voorstellen van complexe zaken zou echter wel eens voor een onplezierige rekening kunnen zorgen.

Wat zeker gemakkelijker zal worden is het unit-testen van Enterprise JavaBeans. Daar waar voorheen nog een toevlucht gezocht moest worden tot mocking en obscure in-container test frameworks, zal het nu een eenvoudige opgave zijn om voor een Plain-Old-Java-Object (POJO) een normale JUnit test te schrijven. Ook beginnen heel voorzichtig, maar toch steeds meer, invloeden vanuit het Aspect Oriented Programming (AOP) tot Java door te dringen. Java 5 annotations zijn daar een eerste begin van en nu wordt dat uitgebreid met de zogenaamde 'interceptors' die bij een EJB op class of methodeniveau kunnen worden aangebracht. Een interceptor stelt je in staat om rondom (@AroundInvoke) de aanroep van een methode in te grijpen en ofwel pré- of post-conditie aan te passen, of een common operatie uit te voeren, zoals bijvoorbeeld logging of security-checks.

VAN SERVICE LOCATOR NAAR DEPENDENCY INJECTION Met de introductie van EJB 3 en de onderliggende technieken zullen een aantal van de bekende J2EE Blueprint design patterns gaan verdwijnen, zoals de Service Locator en de het Transfer Object. De Service Locator valt ten prooi aan het nieuwe mechanisme van Dependency Injection. Met dit mechanisme zal de container (externe) afhankelijkheden naar bijvoorbeeld datasources, message queues of andere EJB's automatisch oplossen. Het zelf uitvoeren van JNDI-operaties of het downcasten van RMI-IIOP stubs behoren daarmee tot het verleden. Het Transfer Object, vaak een (samengestelde) kopie van data uit één of meerdere entity-beans zal ook van het toneel verdwijnen. De Java Persistence API hanteert namelijk een onderscheid tussen 'managed' en 'unmanaged' objecten, waarmee het mogelijk is om referenties naar de echte entity door te spelen tussen Java EE applicatie-lagen. Als er onderweg aan deze objecten iets verandert, dan kunnen deze met behulp van een 'merge'-operatie weer worden gesynchroniseerd met de managed data.

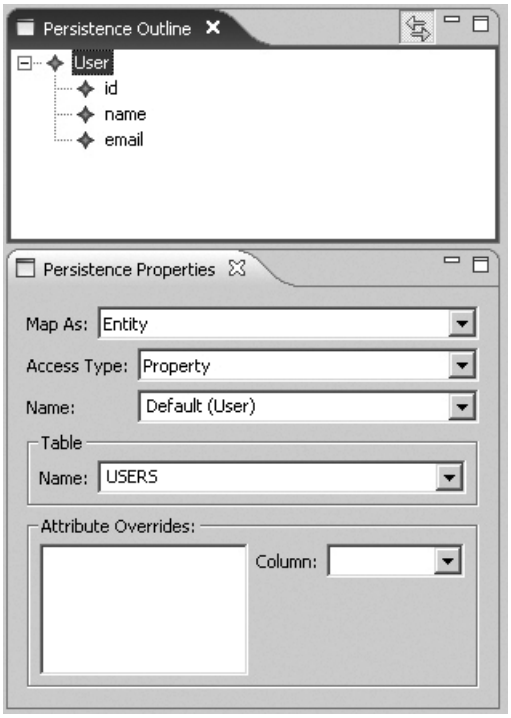
COMPATIBILITEIT EN MIGRATIE Wat slechts weinig mensen weten en wat mijns inziens ook te weinig wordt benadrukt is dat onder EJB ook de volledige EJB 2.x specificatie nog steeds geldig is. Daarmee bedoel ik niet alleen dat het backwards-compatible is, maar ook dat het in principe nog steeds toegestaan is om op basis van EJB 2.x-technologie nieuwe componenten te ontwikkelen. Ook zijn alle EJB-generaties door elkaar heen te gebruiken. Zo kan een client op de EJB 2.0 manier een EJB 3.0 Stateless Session Bean benaderen, die op zijn beurt weer gebruik maakt van een EJB 2.1 Entity Bean. Ook binnen de context van een transactie kunnen de verschillende generaties beans door elkaar heen worden gebruikt. Het is wel belangrijk oplettend te zijn wanneer zowel EJB 2.x als EJB 3.0 entity's dezelfde data benaderen. Vanwege verschillende onderliggende persistency-mechanismen en data caching-implementaties die geen weet van elkaar hebben, kan dit wel eens vervelende gevolgen hebben.

Toch is het geen slecht idee om te gaan denken over een al dan niet toekomstige migratiestrategie. Sommige EJB-typen laten zich gemakkelijker migreren dan anderen. Een slimme aanpak voor migratie zou kunnen zijn om te beginnen met het vervangen van de bestaande Service Façade laag met nieuwe EJB 3.0 Session Beans. Dit is een relatief simpele ingreep. Van daaruit kun je vervolgens de achterliggende componenten gefaseerd vervangen door nieuwe componenten, zonder dat de interface naar de clients hoeft te worden aangepast. Het is zelfs mogelijk om het gemak van de nieuwe dependency injection te gebruiken in bestaande oudere generatie beans.

TOOLING Voor het maken van een testrit met de nieuwe EJB-technologie is in feite niet veel nodig. Een Java IDE met ondersteuning voor Java 5 features, zoals Annotations en Generics is aan te bevelen. Sommige IDE's bieden zelfs al wizards voor EJB 3. Op dit moment

zijn er een aantal applicatieservers beschikbaar die een preview van Java EE 5 technologie bieden. De officiële Reference Implementation van Java EE 5 is Project GlassFish, een open source applicatieserver die op initiatief van Sun wordt gehost op java.net. Ook JBoss heeft een preview beschikbaar. De derde runtime die nogal in het oog springt is de Oracle-applicatieserver. De spierballentaal waarmee Oracle dit product omgeeft moeten we echter wel met een korreltje zout nemen. Hoewel ze pretenderen met hun EJB- en JSF-technologie ver op de markt vooruit te lopen, zijn ze ook vooral ver op zichzelf vooruit. Hoewel er diverse preview-versies van allerlei technologie verschijnt, is er nog maar zeer recent een productierijpe versie van een J2EE 1.4 applicatieserver beschikbaar. Het zal dus nog wel even duren voordat een Java EE 5-versie op de markt komt. Positief en bijzonder interessant is echter wel de moeite die Oracle, samen met BEA en JBoss in het open source-project Dali stopt. Doelstelling van Project Dali is om tooling te leveren voor EJB 3 Object/Relational mapping, met andere woorden 'the persistence of (objects in) memory', een verwijzing naar een beroemd schilderij van de bekende Spaanse schilder Salvador Dali.

Conclusie In Java Magazine nr 1/2004 schreef ik dat Java 5 metadata een grote bijdrage zouden gaan leveren aan de versimpeling van (Enterprise) Java. De huidige toepassing van annotaties binnen EJB, bijvoorbeeld als vervanger van expliciete configuratie met deployment-descriptors, is een perfecte illustratie van wat ons de komende tijd nog te wachten staat. De trend om van complexe zaken een POJO-gebaseerd programmeermodel te maken, zal ongetwijfeld een heftige impact gaan hebben op de productiviteit van enterprise Java-ontwikkeling. Met EJB 3 is tevens een verwoede poging gedaan om de wildgroei aan persistency-oplossingen binnen het Java-platform te uniformeren. Persoonlijk denk ik dat men hier heel aardig in is geslaagd. De toekomst zal echter uitwijzen hoe lang EJB-critici de mond is gesnoerd.



De Dali plugin voor Eclipse biedt mapping wizards en probleemdetectie voor EJB 3.0 entities

Referenties

- JSR-220: Enterprise JavaBeans 3.0 - http://www.jcp.org/en/jsr/detail?id=220
- Project Dali - http://www.eclipse.org/dali/
- Project GlassFish – https:// https://glassfish.dev.java.net/
- Diverse EJB 3 en Java EE 5 gerelateerde artikelen - https://glassfish.dev.java.net/public/TipsandBlogs.html

Ing. Bert Ertman is als IT-architect werkzaam bij Info Support B.V. te Veenendaal en sinds 1996 actief op het vlak van Java development, advisering en training. Binnen Info Support is hij inhoudelijk verantwoordelijk voor de activiteiten van het Competence Center Java. Meer informatie over EJB 3.0 en andere Java gerelateerde onderwerpen is ook te vinden op zijn weblog: http://blogs.infosupport.com/berte. Wilt u reageren op dit artikel? Mail naar berte@infosupport.com.