Serverseitige Komponenteninfrastrukturen – Ein Überblick

Markus Völter, voelter@acm.org, www.voelter.de
Alexander Schmid, Eberhard Wolff

Wiederverwendung ist das mittlerweile fast das „ewig" Versprechen neuer Technologien: strukturierte Programmierung sollte dies durch Funktionsbibliotheken erreichen und Objekt-orientierung durch Vererbung. Damit sollte die Produktivität der Entwickler deutlich verbessert werden. Mittlerweile ist man bei den Komponenten angekommen, die Wiederverwendung als das zentrales Element begreifen. Komponenten jenseits von einfachen GUI Elementen bedeutet heutzutage Wiederverwendung von Komponenten bei Serveranwendungen. Aber wie erreicht man das und welche Ansätze in der Industrie gibt es? Hier soll eine Antwort versucht werden.

Noch eine kurze Vorwarnung: Ein in der Informatik häufig anzutreffendes Problem ist, dass Begriffe nicht einheitlich definiert bzw. verwendet werden. Dies ist auch zu beobachten bei dem Begriff Komponente. Derzeit werden verschiedenste Technologien mit dem Begriff Komponente beschrieben. Eine einheitliche Definition des Begriffs ist dadurch nicht möglich, da sie entweder so weit gefasst werden muss, dass sie nicht mehr sinnvoll einschränkt, oder viele der als Komponenten eingeführten Technologien nicht mehr erfassen kann. Wir beschränken uns in diesem Artikel deshalb auf Komponenten die in Komponenteninfrastrukturen wie EJB, CCM, COM+ und auch .Net zum Einsatz kommen. Diese erfüllen unter anderem als wichtigen Punkt den Aspekt der Verteilung.

Um sie genauer voneinander abgrenzen und beschreiben zu können, ist es sinnvoll die verschiedenen Teilaspekte ihrer Architekturen differenziert zu analysieren und zu beschreiben.

Grundlegende Prinzipien

Die Architekturen von Komponenteninfrastrukturen wurden anhand einiger grundlegender Prinzipien entworfen. Diese Prinzipien erklären die meisten der getroffenen Designentscheidungen und sollten deshalb vorab gesondert beschrieben werden:

Das Kernprinzip von Komponenteninfrastrukturen ist die **Trennung von Belangen:** es sollen technische und funktionale Belangen getrennt werden. Technische Belange sind z.B. Verteilung, Sicherheit, Transaktionen, Nebenläufigkeit, Ressourcenmanagement - funktionale Anforderungen sind die fachlichen Anforderungen des Systems. Sinn dieser Trennung ist es, die beiden Anliegen unabhängig voneinander - und meist auch von

verschiedenen Personen – (weiter-) entwickeln zu können. Hier liegt bereits ein deutlicher Unterschied zur objektorientierten Entwicklung vor. Bei der objektorientierten Entwicklung macht diese Trennung zwar auch möglich und sinnvoll, ist aber nicht automatisch und explizit vorgegeben. Komponenteninfrastrukturen sind also Systeme, die die technischen Belange eines Systems (oder einer Systemfamilie) implementieren. Dies führt zwar zwangsläufig zu einer Einschränkung des Lösungsbereichs, da bestimmte Dinge bereits vorgegeben sind (z.B. wie Transaktionen oder Sicherheit zu handhaben sind), bringt aber auch mit sich, dass die technischen Belange nicht mehr selbst entwickelt werden müssen, sondern von der Stange gekauft werden können. Innerhalb einer Anwendungsdomäne, die ähnliche Charakteristika aufweist, z.B. bezüglich des Transaktionsverhaltens, ist dies ein großer Vorteil. Vor allem findet der größte Teil der Software-Entwicklung im Bereich der Geschäftsanwendungen statt, die typischerweise Lösungen in bestimmten Bereichen wie Transaktionen, Sicherheit oder Persistenz erfordern.

Des Weiteren sollten komponentenbasierte Systeme aus Gründen der Ausfallsicherheit, Lastverteilung und Skalierbarkeit als **Mehrschichtige Systeme** aufgebaut werden. Eine spätere Verteilung der einzelnen Schichten auf verschiedene Maschinen wird damit ermöglicht. Außerdem erhöht dies die Möglichkeiten zur Wiederverwendung und erleichtert auch Weiterentwicklung bzw. Wartung des Systems. Die hier betrachteten Komponenteninfrastrukturen unterstützen daher die Aufteilung eines Systems in mehrere Schichten.

Als drittes Prinzip ist die **Variabilität** zu nennen. Um die Möglichkeit der Wiederverwendung einzelner Komponenten zu erhöhen, sollten Komponenten einen gewissen Grad an Flexibilität aufweisen. Nur so können sie in unterschiedlichen Kontexten verwendet werden ohne jedes mal programmiertechnisch anzupassen.

Bestandteile von Komponenteninfrastrukturen

Im Mittelpunkt eines komponentenbasierten Systems steht natürlich die **Komponente**. Diese besitzt einen genau definierten Verantwortungsbereich, die Gesamtfunktionalität der Anwendung wird also in einzelne Komponenten aufgeteilt. Da Komponenten verschiedene Aufgaben erfüllen müssen, werden verschiedene Komponententypen unterschieden: Zum einen gibt es **Entitäts-Komponenten**, die persistente Daten repräsentieren. Zusätzlich gibt es **Session-Komponenten**, die Zustand innerhalb einer Benutzersitzung speichern können. Zuletzt existieren noch **Service-Komponenten**, die Dienste zur Verfügung stellen, die innerhalb eines Aufrufs abgearbeitet werden können. Dabei kann man Entitäts Komponenten zur Abbildungen von Entitäten aus dem Geschäftsbereich verwenden. Entsprechenden können Session Komponenten für einfache Geschäftsprozesse verwendet werden, die ohne Zustand auskommen. Längere Prozesse, die einen Zustand mitführen müssen, können dann durch Service Komponenten implementiert werden.

Kommentar: Ich finde das eine wirklich komische Formulierung

Damit die Komponenten miteinander arbeiten können ohne auf konkrete Implementierungen angewiesen zu sein, wird das **Komponenteninterface** eingeführt. Dieses definiert die Signatur[1] der Operationen, die eine Komponente anbietet. Eine Komponente hat ausschliesslich Abhängigkeiten zu den *Interfaces* anderer Komponenten, nicht zu deren Implementierung. Die Implementierungen der Komponenten können dadurch beliebig unabhängig voneinander weiterentwickelt werden. Natürlich kann eine Komponente auch mehrer Interfaces zur Verfügung stellen.

Auch dies ist wieder ein deutlicher Unterschied zur Objektorientierung. Komponentenbasierte Systeme verwenden ausschließlich Blackbox-Reuse, also Wiederverwendung basierend auf der Komponentenschnittstelle beispielsweise durch Delegation. Vererbung zwischen Komponenten ist nicht vorgesehen, da es sich dabei um Whitebox-Wiederverwendung (mit all den bekannten Problemen) handeln würde (siehe u.a. [3]). Bei der Vererbung sind nämlich Implementierungdetails wie Instanzvariablen sichtbar.

Komponenten implementieren, wie erläutert, nur die funktionalen Belange der Anwendung. Damit die Gesamtanwendung aber richtig funktioniert, müssen auch die technischen Belange irgendwie realisiert werden: Diese implementiert der **Container**, eine Ablaufumgebung für Komponenten. Er kümmert sich um Transaktionen, Sicherheit, Threading, etc., was auch immer als „technische Belange“ in der entsprechenden Domäne identifiziert wird, wobei die hier vorgestellten Technologien auf Geschäftsanwendungen abzielen. Der Komponentenentwickler muss diese also nicht selbst implementieren. Container werden üblicherweise als fertiges Produkt gekauft.

Um nun mehrschichtige Anwendungen realisieren zu können, müssen die Komponenten im Container über das Netz von Clients oder anderen Komponenten aus erreichbar sein. Ein **Komponentenbus** abstrahiert die Kommunikationsinfrastruktur (Netzwerk, Protokoll, etc.). Er transportiert Operationsaufrufe über's Netz und bedient sich dabei üblicherweise OO-RPC Mechanismen wie CORBA/IIOP, RMI/JRMP oder DCOM/RPC.

Nun können nicht zu jeder Zeit alle Instanzen einer Komponente aktiv im Speicher des Containers sein, da dies zu Ressourcenproblemen führen würde. Daher arbeitet ein Container üblicherweise mit **virtuellen Instanzen:** Die logische Identität einer Komponente und die physikalische Komponenteninstanz werden konzeptionell getrennt. Der Container kann einer physikalischen Instanz nacheinander beliebig viele logische Instanzen zuweisen (**Pooling**), oder nicht gebrauchte Instanzen temporär aus dem Speicher entfernen (**Passivierung**). Ein **Komponentenproxy** dient dabei als Statthalter für eine oder mehrere gerade nicht aktive logische Instanzen. Der Client (bzw. eine als Client agierende

[1] Interfaces definieren derzeit bei praktisch allen Systemen nur die Signatur, also die Syntax und die Typen, der Operationen. Semantische Spezifikationen sind schwierig, Ansätze wie Pre- und Postconditions (siehe Eiffel oder OCL) werden in der Praxis kaum verwendet.

andere Komponente) kommuniziert grundsätzlich nur mit dem Komponentenproxy, was dem Container erlaubt, nach eigenem Gutdünken die Identitäten zu wechseln oder Komponenten zu reaktivieren. Der Komponentenproxy sieht dabei nach außen aus wie die Komponente selbst – er stellt also das gleiche Interface zur Verfügung.

Damit all dies funktionieren kann, muss der Container die Komponente, genauer gesagt ihren Lebenszyklus, kontrollieren. Komponenten sind passive Einheiten, damit der Container eine Komponente in ihrem Lebenszyklus steuern kann, muss sie dem Container **Lebenszyklusoperationen** zur Verfügung stellen. Diese ruft der Container auf, um bspw. einen Wechsel der logischen Identität einer Instanz herbeizuführen, oder sie z.B. in einen persistenten Speicher auszulagern. Der Implementierer der Komponente muss sich nicht um das Management des Lebenszyklus kümmern, er muss lediglich die Lebenszyklusoperationen korrekt implementieren.

Des Weiteren müssen Komponenteninstanzen erzeugt, zerstört, oder - bei Komponenten mit logischer Identität - wieder gefunden werden. Diese Vorgänge können beliebig komplex sein und hängen üblicherweise eng mit der Implementierung des Containers zusammen: Dinge, , die den Client oder den Komponentenimplementierer nicht interessieren sollten. Es handelt sich schließlich um einen technischen Belang. Daher benötigt man ein **Komponenten-Home**, welches dem Client entsprechende Operationen zur Verfügung stellt. Das Komponenten-Home wird zum Grossteil vom Container implementiert.

Der Client oder eine Client-Komponente muss allerdings irgendwie in den Besitz eines solchen Komponenten-Homes kommen, er kann es sich ja nicht einfach „selber bauen“: Dazu wird üblicherweise ein **Namensdienst,** wie man ihn auch aus CORBA oder Java’s RMI kennt, verwendet. Dort wird eine Komponente, genauer gesagt deren Komponenten-Home, registriert und der Client kann dort dieses anhand eines dem Client bekannten Namens erfragen. Damit bleibt der tatsächliche Ort einer Komponente (also der Server) für den Client transparent.

Leider kann das Ideal, dass sich der Komponentenentwickler nur um die funktionalen Belange kümmern muss, nicht ganz erreicht werden. Der Entwickler muss zumindest sagen, *was* der Container für ihn tun soll (nicht aber, *wie*!): Er muss also beispielsweise angeben, auf welche Operationen welcher Benutzer zugreifen darf, oder wo eine Transaktion notwendig ist. Er muss dies nicht selbst programmieren; aber er muss **Anmerkungen** schreiben, in denen deklarativ vermerkt ist, welches Verhalten er sich vom Container erwartet. Der Container muss diese Anmerkungen dann umsetzen - damit die Performance stimmt, wird dann daraus üblicherweise eine **Glue-Code Schicht** generiert. Es wird meist also Code generiert statt die Anmerkungen immer wieder zu interpretieren, was aber natürlich auch ein gangbarer weg wäre. Teil dieser generierten Schicht ist auch der oben erwähnte Komponentenproxy.

Dieser Glue-Code muss natürlich irgendwann erzeugt werden, allerdings erst nachdem die Konsistenz und Korrektheit der Anmerkungen überprüft wurde. Daher ist ein expliziter **Installationsschritt** notwendig. Hier wird die Komponente (inkl. den Anmerkungen) dem Container übergeben, dieser überprüft sie und führt, wenn alles in Ordnung ist, alle notwendigen Schritte durch, um die Komponente im Container laufen lassen zu können. Dabei wird sie auch mit dem Komponentenbus verbunden, und ihr Home wird im Namensdienst registriert.

Wenn die Komponente dann im Container läuft kann sie dies nicht ohne einen gewissen Kontakt zum Container tun. Sie muss zum Beispiel auf Ressourcen zugreifen können, oder muss den Container über den Status der aktuellen Transaktion unterrichten können. Dazu wird ihr vom Container ein **Komponentenkontext** zur Verfügung gestellt. Ressourcen sollten nicht von der Komponente selbst verwalten werden. Vielmehr ist die **Verwaltung von Ressourcen**, also z.B. das Anlegen von entspr. Pools, Aufgabe des Containers. Dieser kann dann Konzepte wie Ressourcen-Pooling umsetzen und damit Skalierbarkeit und Performanz des Gesamtsystems steigern.

Um dem Anspruch an Variabilität gerecht zu werden, muss eine Komponente Zugriff auf **Konfigurationsparameter** haben. Diese können bei der Installation der Komponente mit den passenden Werten versehen werden. Der Komponentenkontext bietet der Komponente dann zur Laufzeit Zugriff auf diese Parameter. Auf diese Weise kann sichergestellt werden, dass sich Komponenten auch unter verschiedenen Einsatzszenarien verwenden lassen.

Zu guter letzt muss noch sichergestellt werden, dass bei der Ausführung einer Operation einer Komponente der Container Zugriff auf alle für die technischen Aspekte nötigen Informationen besitzt. Z.B. muss er Zugriff auf die aktuelle Transaktion oder Sicherheitsinformationen haben. Beim Aufruf von Operationen durch den Client, oder durch einen andere Komponente, muss also mehr übertragen werden als nur die reine Operation und deren Parameter: dazu wird ein **Aufrufkontext** benötigt. In ihm sind eben diese Informationen wie Transaktionskontext usw. enthalten. Er wird transparent bei jedem Aufruf an eine Komponente mit übertragen.

Dies sind die wichtigsten Bestandteile moderner Komponenteninfrastrukturen. Tatsächlich existieren aber noch einige mehr. Diese können hier aber aus Platzgründen nicht alle aufgeführt werden. Sie sind in [2] vollständig in Form einer Mustersprache beschrieben und genauer erläutert. Dort werden auch die Beweggründe die zu den hier aufgeführten Bestandteilen geführt haben diskutiert. Das Buch diskutiert weiterhin eine Projektion der Patterns in die Technologien EJB, CCM und COM+. Tabelle 1 zeigt kurz welche der Patterns in den Technologien EJB, COM+, .Net und CCM zu finden sind.

Eine Implementierung: EJB

Im Folgenden soll eine kurze Projektion der oben beschriebenen Patterns auf EJB durchgeführt werden, da EJB sicherlich die Komponenteninfrastruktur ist, die für Lesern das Java Magazins am wichtigsten ist. Damit soll gezeigt werden, wie die oben beschriebenen Konstrukte tatsächlich implementiert wurden.

Im Rahmen von EJB (Version >=2.0) unterscheidet man zwischen vier Arten von **Komponenten**: Stateless Session Beans, Stateful Session Beans, Entity Beans, und Message Driven Beans. Stateless Session Beans stellen dabei eine Implementierung der Service Komponenten dar, also Komponenten ohne Zustand, die Ihre Aufgaben innerhalb eines Methodenaufrufes abarbeiten müssen. Stateful Session Beans haben einen Zustand und können dementsprechend auch komplexere Prozesse abdecken. Entity Beans haben einen persistenten Zustand, der in einer Datenbank abgelegt wird und entsprechen damit dem Entitäten Konzept.

Während die ersten drei Komponentenarten also einfach die Implementierungen der oben beschriebenen Konzepte sind, stellt sich die Frage, wie in diesem Zusammenhang Message Driven Beans einzuordnen sind. Im Prinzip sind diese eine Ausprägung von Service Komponenten, mit dem Unterschied, dass sie nicht synchron sondern asynchron aufgerufen werden. Aufgrund der Integration mit JMS benötigen sie auch kein Komponenten Interface im eigentlichen Sinn, Die Schnittstelle wird innerhalb der MDB programmiert, indem die empfangenen Messages interpretiert werden.

Alle anderen Komponententypen haben auch in EJB ein **Interface**, genauer gesagt, sie können sogar zwei haben: Das Remote Interface und das Local Interface. Das Remote Interface definiert die Operationen, die auf einer Bean von entfernten Clients aufgerufen werden können, das Local Interface definiert die Operationen, die nur innerhalb desselben Application Servers, also von anderen Beans, aufgerufen werden können. Diese Aufspaltung der Interfaces wurde hauptsächlich aus Performanzgründen vorgenommen. Neben der Netzwerkkommunikation ist bei einem entfernten Aufruf nämlich auch ein Umkopieren der Parameter und Ergebnisse (Marshalling/Unmarshalling) notwendig.

Die technischen Belange stellen in EJB Bean-Typ spezifische **Container** zur Verfügung. Diese laufen innerhalb eines J2EE Application Servers. Die Container kümmern sich um Security, Transaktionen, virtuelle Instanzen, Persistenz und einiges mehr, je nach Bean-Typ. Der **Komponentenbus** basiert auf RMI, welches wiederum verschiedene Kommunikationsprotokollen verwenden kann – üblicherweise IIOP. Teils bieten Applikationsserver aber auch proprietäre Protokolle an, die erhöhte Sicherheit, Performanz usw. bieten können (ein Beispiel ist BEA's t3 Protokoll).

Um Ressourcen zu schonen, verwendet auch EJB das Prinzip, dass logische und physikalische Identitäten einer Komponenteninstanz unterschieden werden (**Virtuelle Instanzen**), und zwar folgendermaßen:

- Bei Stateless Session Beans und Message Driven Beans legt der Container einfach einen Pool von Instanzen an. Jeder Request wird dann an eine der Instanzen im Pool weitergeleitet – da sie stateless sind, sind alle gleich und der Container kann sich für einen Methodenaufruf eine beliebige Instanz aussuchen.
- Bei Stateful Session Beans geht der Container anders vor: Wann immer der Client einen neue Instanz anfordert (durch aufrufen einer create() Operation auf dem Home), wird auch wirklich physikalisch einen neue angelegt. Diese Instanzen werden passiviert (also auf Platte ausgelagert), wenn sie für einen bestimmten Zeitraum nicht mehr benötigt wurden oder die Ressourcen auf dem Server knapp werden – wenn wieder ein Aufruf für die Instanz ankommt, werden sie wieder aktiviert.
- Bei Entity Beans geht der Container nochmals anders vor. Wie bei Stateless Session Beans legt er einen Pool von Instanzen an. Wenn ein Aufruf für eine bestimmte logische Instanz (also eine bestimmte Entität) beim Server ankommt, ruft er bestimmte Lebenszyklusoperationen auf (z.B. *ejbLoad()*), sodass die Instanz auch wirklich die benötigte Identität – und deren persistenten Zustande kommt.

Damit all dies funktioniert, wird beim Deployment ein **Komponentenproxy** generiert, der, oft per Reflection, die Operationen auf der Instanz aufruft, nachdem er sich vorher um Transaktionen und Security gekümmert, und die nötigen Lebenszyklusoperationen auf der Instanz aufgerufen hat. Diese **Lebenszyklusoperationen** sind in den *SessionBean* und *EntityBean* Interfaces spezifiziert. Jede Bean-Implementierungsklasse muss diese Operationen korrekt implementieren. Das **Komponenten-Home** wird durch das Home Interface (bzw. durch das Local Home, im lokalen Fall) realisiert. Dies wird vom Entwickler formal nie implementiert, es wird vom Container automatisch in der generierten **Glue-Code Schicht** implementiert.

Der **Namensdienst** wird bei EJB über das JNDI (Java Naming and Directory Interface) angesprochen, das dann CORBANaming, LDAP oder andere Namensdienste verwenden kann. Üblicherweise bieten Application Server aber ihre eigene Implementierungen.

Auch in EJB benötigt man **Anmerkungen** um den Container bei der Realisierung seiner technischen Aspekte zu steuern. Diese Anmerkungen sind in EJB XML Dateien und heißen Deployment Deskriptoren. Ein Deployment Descriptor enthält Angaben darüber, welche Attribute persistent sein sollen (bei CMP), für welche Operationen Transaktionen notwendig sind, welche Benutzer welche Operationen aufrufen dürfen und mehr. Daraus generieren die meisten Applikationsserver die **Glue-Code Schicht**. Der Installationsschritt heißt bei EJB Deployment, und er wird üblicherweise durch Application-Server spezifische Tools unterstützt.

Beans können ihre Umgebung ansprechen indem sie entweder den *SessionContext/EntityContext* verwenden, ein **Komponentenkontext** den ihnen der Application Server zu Beginn ihres Lebenszyklusses zur Verfügung stellt. Sie können aber

auch direkt Lookups im JNDI Kontext durchführen. Dort können sie zum Beispiel *DataSources* finden, welche die Rolle der **Managed Resources** für Datenbanken spielen. **Konfigurationsparameter** sind auch im Deployment Descriptor angegeben, sie sind zur Laufzeit von der Bean aus einem speziellen Teil des JNDI-Kontextes auslesbar.

Zu guter Letzt sei erwähnt, dass auch EJB einen **Aufrufkontext** unterstütz, der die Transaktions- und Security Tokens enthält.Dieser kann entweder nativ mit IIOP oder manuell per RMI übertragen werden.

Warum sind Komponentenmodelle erfolgreich?

Dafür gibt es mehrere Gründe:

- Die meisten Applikationen in diesem Umfeld haben eine ähnliche Struktur: Eine Datenbank die Business-Entitäten enthält, Prozesse die diese verändern, und das ganze üblicherweise transaktional speichern. Durch diese Gemeinsamkeit ist es leicht, technische und funktionale Aspekte zu trennen und die technischen Aspekte wiederzuverwenden.
- Entwickler können sich auf die fachlichen Dinge konzentrieren und müssen sich nicht um Lastverteilung, Transaktionen, etc. kümmern. (Oder, wenn sie beim Applikationsserver Hersteller arbeiten, die fachlichen Dinge ignorieren und nur technische Aspekte implementieren.)
- Es liegen Standards vor, die es Herstellern ermöglicht kommerziell Container-Implementierungen anzubieten die Kunden nicht an proprietäre Lösungen fesseln. Dies erhöht die Kundenakzeptanz und vergrößert dadurch den Markt. Aus Anwender Sicht gibt es zu Standart Technologien mehr Knowhow im Markt und Investitionsschutz da man nicht mehr an die wirtschaftliche Entwicklung eines Zulieferers gekoppelt ist.

Diese Gründe haben bereits den Relationalen Datenbanken geholfen eine marktdominierende Stellung zu gewinnen. Zusammenfassend kann man sagen, dass die Komponenten basierte Entwicklung sicherlich große Vorteile bietet. Trotzdem ist auch sie keine „Silver Bullet“ und je weiter das zu lösende Problem von dem von der Infrastruktur vorgesehenen Problembereich entfernt ist, um so weniger macht es meist Sinn komponentenbasierte Entwicklung einzusetzen. Neben der reinen Anwendung dieser Frameworks ist es deshalb besonders wichtig die zugrunde liegenden Prinzipien, wie die Trennung von technischen und funktionalen Belangen und die eingesetzten Mechaniken wie zum Beispiel Virtuelle Instanzen zu verstehen, um diese auch sinnvoll jenseits der EJB Welt einsetzen zu können.

Vergleich von EJB, CCM und COM+

Tabelle 1 zeigt die Patterns (oder auch „Bausteine“, oder Features) die in den verschiedenen Architekturen zum Einsatz kommen. Die Grundbausteine sind dieselben, es gibt aber schon auch Unterschiede.

Zu EJB müssen wir hier wohl nicht mehr viel sagen. CCM kann man als eine auf CORBA basierende und daher sprach- und plattformunabhängige Obermenge von EJB sehen (es ist auch wirklich aufwärtskompatibel). Es gibt bisher experimentelle Implementierungen (zumindest von Teilen), kommerziell verfügbar ist bisher aber noch nichts. Aufgrund der großen Mächtigkeit und Komplexität von CCM, und aufgrund der Marktdominanz von EJB außerhalb der MS-Welt darf bezweifelt werden, ob CCM wirklich jemals die Bedeutung von EJB oder COM+ erreichen wird, zumal CORBA im Moment nicht mehr im Mittelpunkt des Interesses der meisten Entwickler liegt.

COM+ ist das Komponentenmodell von Microsoft, aufsetzend auf dem Microsoft Transaction Server (der seit Windows 2000 Teil des Betriebssystems ist, Stichwort „Komponentendienste“). COM+ unterstützt von allen Komponentenmodellen die wenigsten Features. Nichts desto trotz hat COM+ eine erhebliche Verbreitung, was zeigt, daß man auch mit einer reduzierten Menge an Features gute Systeme bauen kann. Beispielsweise unterstützt COM+ technisch gesehen nur zustandslose (Service-) Komponenten, was zunächst nach einer Einschränkung klingt. Wirklich große Systeme versuchen aber auf Gründen der Skalierbarkeit sowieso nur mit zustandslosen Komponenten auszukommen. Dadurch können nämlich die Komponenteninstanzen ohne Probleme gegeneinander ausgetauscht werden, was auch über Server hinweg funktioniert und damit Clustering recht gut unterstützt. Wenn dann ein Server ausfällt, kann man ebenfalls auf die Komponenten auf dem Server verzichten und auf identische auf einem anderen „Umschalten“. Bei Komponenten mit Zustand ist dies weniger Einfach, da man diesen verliert. Letzendlich können die Service Komponenten die Daten direkt aus der Datenbank lesen und ablegen, so dass eine Verwaltung persistenter Daten möglich ist, ohne dass der Server selber einen Zustand hat. Dass COM+ weniger Features hat, bedeutet also in erster Linie, dass es ein anderes Design verfolgt, aber lässt noch keine Rückschlüsse auf die Leistungsfähigkeit zu.

Noch ein Wort zu .NET

.NET ist ja *die* Konkurrenz zu Java (oder J2EE, oder Sun ONE, oder was auch immer...). Es ist für eine Version 1.0 auch schon recht brauchbar und komplett. Im Bereich der Komponenteninfrastrukturen sieht es allerdings nicht ganz so glorreich aus: .NET definiert an sich kein Komponentenmodell im hier definierten Sinne. Es gibt zwar GUI Komponenten, und oft werden auch Assemblies als Komponenten bezeichnet – aber das sind eben Beispiele dafür, daß mit dem Begriff *Komponente* verschiedenste Dinge bezeichnet werden. Um in den Genuß einer serverseitigen Komponenteninfrastruktur zu kommen, müssen .NET (Remote-) Objekte im Rahmen einer COM+ Anwendung deployed

werden. Dies ist zwar aufgrund entsprechender Adapter und Wizards kein großes Problem, aber es ist der Grund warum wir in der Vergleichstabelle .NET nicht extra aufgeführt haben. Die Features sind identisch mit denen von COM+.

Webservices als Web-Komponenten?

Dieser Tage hört man ja oft die Aussage, Webservices seien die nächste Generation von Komponenten (und implizit: EJB, CCM oder COM+ sind out!). Aufgrund der obigen Diskussion dürfte offensichtlich sein, daß wir das nicht so sehen.

Wenn man als Definition von Komponente nur einfach die Tatsache verwendet, dass der Softwarebaustein ein wohldefiniertes Interface hat und damit austauschbar wird, dann ist ein Webservice sicherlich eine Komponente (oder ein Modul, oder ein Subsystem, oder was auch immer...). Wenn man allerdings berücksichtigt, dass es bei Komponenteninfrastrukturen immer auch einen Container gibt, der sinnvolle, wiederkehrende Aufgaben für die Komponenten erledigt, dann ist klar, das Webservices keine Komponenteninfrastruktur in diesem Sinne sind, sondern allenfalls ein Kommunikationsprotokolle, um Komponenten anzusprechen. In der Praxis wird dies ja auch schon daran deutlich, daß z.B. im Falle von Java Webservices im EJB 2.1 Standard als zusätzliches Interface zu EJBs gesehen werden (natürlich nur auf Stateless Session Beans, weil Webservices immer auf einem zustandslosen Kommunikationsmodell aufbauen). Vielleicht gibt es eines Tages einmal native Webservice-Container, wobei noch nicht absehbar ist, wie so etwas aussehen könnte.

Andere Komponentenmodelle

Wir haben uns in diesem Artikel (und auch in [2]) ausschliesslich auf serverseitige Komponentenmodelle zur Abbildung der Businesslogik konzentriert. Kernprinzip dabei war die Trennung von funktionalen und technischen Belange und die Erledingung der letzteren durch einen Container. Dieses Vorgehen – ein Container der für die eigentlichen „Programme“ bestimmte Dienste erbringt – wird auch an anderer Stelle verwendet, beispielsweise für Servlets. Inwieweit man diese Technologien deshalb als Komponentenmodelle bezeichnen will ist letztlich Ermessenssache[2]. Unseres Erachtens muss der Container eben schon nennenswerte Dienste erbringen und nicht nur einen *start()*, *stop()* oder *service()* Callback aufrufen, oder ein bischen Thread-Handling beisteuern. Hinzu kommt, dass Servlets oder JSP typischerweise keine Business Logik implementieren, sondern dies beispielsweise an EJBs delegieren. Ein serverseitiges Komponentenmodell sollte jedoch gerade dies unterstützen.

Aus diesen Gründen sehen wir z.B. Servlets nicht als Komponentenmodell, obwohl im Rahmen von J2EE von einen Servlet-Container die Rede ist.

[2] Wir haben uns entschieden, dies nicht zu tun.

Fazit

Komponentenbasierte Entwicklung auf dem Server hat mit Sicherheit große Vorteile. Zum einen ermöglicht es dem Entwickler, sich weniger um technische Belange wie Transaktionen, Sicherheit oder vor allem auch Skalierbarkeit kümmern zu müssen. Zum anderen ist damit die Basis für einen Komponentenmarktplatz geschaffen, der (vielleicht) eines Tages dazu führt, dass typische Objekte in den Domänen für ein Projekt eingekauft werden können.

Will man entscheiden, welche Komponenteninfrastruktur die „beste" ist, sollte man sich darüber im klaren sein, dass hier üblicherweise weniger technische Fragen eine Rolle spielen. EJB (und damit J2EE) sind die einzigen Java Komponentenarchitekturen und damit bedeutet die Entscheidung für Java auch eine Entscheidung für EJB/J2EE. Eine Entscheidung für die Microsoft Plattform impliziert genauso eine Entscheidung für COM+, wobei Microsoft eben nur seine eigene Windows Plattform unterstützt. .NET bietet, wie oben erläutert, kein eigenes Komponentenmodell in unserem Sinne und verläßt sich dazu auf COM+. Natürlich bietet .NET andere Vorteile (größere Offenheit, Webservices, etc.) aber die stehen hier nicht zur Debatte.

Referenzen

[1] B. Cox, A. Novobilski: *Object-Oriented Programming*, Addison-Wesley, 1986.

[2] M. Völter, A. Schmid, E. Wolff: *Server Component Patterns – Component Infrastructures illustrated with EJB*, Wiley, 2002.

[3] J. Bloch: *Effective Java*, Addison-Wesley, 2001.

Pattern	*EJB*	*COM+*	*CCM*
Komponente	👍	👍	👍
Service-Komponente	👍	👍	👍
Session-Komponente	👍	👎 (können emuliert werden)	👍
Entitäts-Komponente	👍	👎	👍
Komponenten Interface	👍	👍	👍

Container	Container sind Bestandteil eines Applikationservers	Container ist integriert mit dem Betriebssystem	👍
Komponenten Bus	👍	👍	👍
Virtuelle Instanz	👍	👍	👍
Pooling	👍	👍	👍
Passivierung	👍	👍	👍
Komponenten Proxy	👍	👍	👍
Lebenszyklus-operationen	👍	👍	👍
Komponenten-Home	👍	👍	👍
Namensdienst	👍	👎	👍
Anmerkungen	👍	👍	👍
Glue-Code Schicht	👍	👍	👍
Installationsschritt	👍	👍	👍
Komponenten Kontext	👍	👍	👍
Verwaltung von Resourcen	👍	Teil von z.B. OLEDB, kein Bestandteil von COM+	👍
Konfigurations-parameter	👍	👍	👍
Aufrufkontext	👍	👍	👍