Reich der Mitte

Die Komponententechnologien COM+, EJB und „CORBA Components“

Michael Stal

Vor wenigen Jahren noch beschränkten sich Untersuchungen komponentenbasierter Softwareentwicklung entweder auf Präsentationskomponenten oder auf Infrastrukturen für verteilte Objekte. Analog zu vielen anderen Bereichen der Softwaretechnik hat das Internet auch hier zu einer Verschiebung des Weltbilds geführt. Mittlerweile gehören Präsentationskomponenten à la „JavaBeans“ oder „ActiveX Controls“ fast schon zur Pflichtübung professioneller Softwareentwickler. Der Mittelpunkt des Entwicklerinteresses hat sich längst vom Desktop zum Server verlagert. COM+, „CORBA Components“ und EJB heißen die modernen Objekte der Begierde. Der Artikel stellt die generellen Konzepte server-basierter Komponenteninfrastrukturen vor und beleuchtet die wichtigsten Technologien, ihre Gemeinsamkeiten und Unterschiede.

Eine f r alles: Multi-Tier-Architekturen

Ein Blick auf die notwendige Infrastruktur für E-Commerce-Systeme genügt, um die Gründe für diese Entwicklung zu verstehen (siehe Abb. 1): Web-basierte Systeme benötigen inhärent eine Softwarearchitektur, die sich in mehrere unabhängige Ebenen zergliedert. In der Präsentationsebene laufen Web-Browser und andere GUI-basierte Applikationen, die auf Funktionalität im Webserver zugreifen, etwa auf *Servlets* oder *Active Server Pages*. Von dort erfolgt eine Delegation an eine Ebene von Geschäftsobjekten, die sowohl Abläufe als auch persistente Informationen modellieren. Für die eigentliche Persistenz sorgen die in der vierten logischen Ebene beheimateten Datenbanksysteme. Das Ganze ist unter dem Begriff „Multi-Tier Systeme“ verbreitet, ein Schlagwort, das manche IT-Anbieter gerne als universelle Marketingkeule benutzen.

Problemzone Server

Aus Entwicklersicht sind gerade die Ebenen der Geschäftsobjekte und Datenbanksysteme von entscheidender Bedeutung. Zum einen hat sich dort im Gegensatz zu den web-basierten Technologien noch kein universeller Standard etabliert. Zum anderen ist ein Entwickler gerade im heterogenen Serverbereich mit diffizilen Integrationsproblemen konfrontiert. Für nicht-triviale Anwendungen genügt es nämlich nicht, einen objektorientierten Kommunikationsmechanismus wie *Remote Method Invocation (RMI)* oder *Common Object Request Broker Architecture (CORBA)* zu nutzen. Vielmehr ergeben sich bei deren Anwendung zusätzliche Probleme:

- Wer ist für das sogenannte Lebenszyklusmanagement von Objekten verantwortlich, also beispielsweise für das Kreieren, Aktivieren oder Deaktivieren von Objekten?
- Auf welche Art und Weise erhält eine Applikation effizienten Zugriff auf die Geschäftsobjekte des Middle-Tiers?
- Wie sind die diversen Sicherheitsanforderungen zu erfüllen?
- Ist eine werkzeuggestützte Installation und Administration von Funktionalität möglich?
- Verteilte Anwendungen verfügen über einen verteilten Zustand. Wie lässt sich dieser Zustand persistent speichern und gleichzeitig in Fehlerfällen konsistent halten?
- Ist es möglich, dem Entwickler Unabhängigkeit von den Spezifika der verwendeten Systemplattform zu gewährleisten?
- Wie soll die Einbindung ins Web erfolgen?

Ein möglicher und sehr häufig anzutreffender Ansatz besteht darin, für jedes dieser Teilprobleme eine möglichst perfekte Einzellösung zu erarbeiten. Das Entwickler hat dann für die Komposition der zahlreichen Einzellösungen zu einem harmonischen

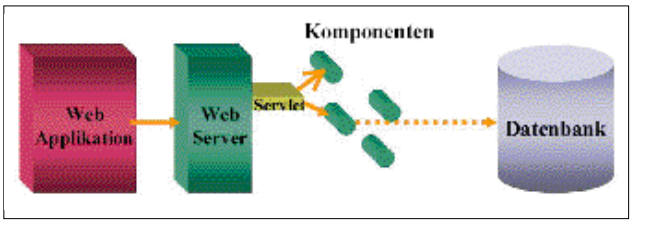

Abb. 1: Multi-Tier Architekturen (Web-basierte Systeme lassen sich inh rent in die logischen Ebenen Pr sentation, Webserver, Gesch ftsobjekte und Datenbank zerlegen.)

Michael Stal ist Mitarbeiter der Zentralabteilung Technik der Siemens AG in M nchen. Er ist Mitglied der OMG, Chefredakteur von JavaSpektrum, Pr sident des „International Java Club“ und Koautor des Buches „Pattern-Oriented Software Architecture – A System of Patterns“.

Ganzen zu sorgen. Erfahrungsgemäß führt dieser Ansatz dazu, dass über 50% des Entwicklungsaufwandes in die Lösung von Integrationsaufgaben fließen statt in die Bereitstellung von Geschäftslogik. Hier tritt konsequenterweise nicht nur ein Zeitproblem zu Tage, sondern auch ein Kostenproblem.

Wie sieht der Ausweg aus dieser Komplexitäts- und Kostenspirale aus? Die Antwort liegt geradezu auf der Hand. Statt dem Entwickler die Integration der genannten Aufgaben aufzubürden, sollte dies die Aufgabe der Infrastruktur sein. Das Stichwort heißt somit „Rationalisierung durch Automatisierung". Womit wir bei den Middle-Tier-Komponententechnologien angelangt sind, die genau diesen Ansatz verfolgen.

Die Idee

Zentrale Idee der Middle-Tier-Komponententechnologien ist neben der notwendigen Bereitstellung eines verteilungstransparenten Kommunikationsmechanismus, wie DCOM oder CORBA, die Integration aller sonst benötigten Dienste.

Sogenannte Container bilden einen Schutzwall um Komponenten und verbergen Systemabhängigkeiten durch eine homogene Schicht von Programmierschnittstellen (siehe Abb. 2). Über diese Schnittstellen können Komponenten Informationen über ihre Laufzeitumgebung erfragen oder deren Verhalten steuern. Container sorgen des Weiteren für das Lebenszyklusmanagement der Komponenten, indem sie diese auf Bedarf kreieren, aktivieren oder „passivieren". Damit Komponentenentwickler die notwendige Funktionalität für die verschiedenen Phasen im Lebenszyklus einer Komponente bereitstellen können, implementieren Komponenten Rückrufschnittstellen, über die sie der Container beim Eintreffen der entsprechenden Ereignisse benachrichtigt.

Zudem verdeckt der Container den Zugriff auf die grundlegenden Dienste der Infrastruktur, wie zum Beispiel Transaktionen, Ereignisdienste, Datenbankzugriffe oder Sicherheitsmechanismen. Er stellt den Komponenten einen programmatischen Zugriff auf die Dienste zur Verfügung, erlaubt aber auch ein deskriptives Vorgehen. In letzterem Fall beschreibt der Entwickler in speziellen textuellen Deskriptoren, beispielsweise unter Nutzung von XML, welche Laufzeitumgebung eine bestimmte Komponente benötigt. Dazu gehören etwa die Beschreibung aller Personen und Gruppen, die Zugriff auf die Komponente haben sollen, ebenso wie der persistent zu speichernde Zustand der Komponente sowie ihr Verhalten bei Transaktionen. Die Aufgabe des Containers besteht darin, abhängig vom Inhalt der Deskriptoren automatisch die gewünschte Laufzeitumgebung bereitzustellen, also z. B. eine Transaktion zu starten oder Sicherheitsbedingungen zu überprüfen. Allerdings gibt es mit diesem Lösungsansatz ein gravierendes Problem: Wie kann ein Container transparent die richtige Laufzeitumgebung bereitstellen, wenn ein externer Client direkt auf die Komponente zugreift? Die Antwort heißt *Interception*: Der Client erhält keinen direkten Zugriff auf Komponenten, mag dies aus seiner Sicht auch so erscheinen. Stattdessen fängt der Container die Aufrufe der Clients ab und arbeitet zusätzliche Funktionalität ab, bevor er schlussendlich den Aufruf an die eigentliche Komponente delegiert.

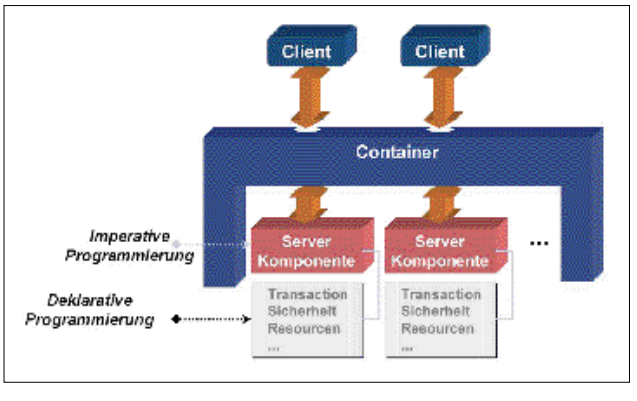

Abb. 2: Container (Ein Container fungiert als Schutzh lle f r Komponenten. Diese spezifizieren ihre Anforderungen an die Laufzeitumgebung deklarativ. Aufgabe des Containers ist es, durch „Interception"-Mechanismen automatisch f r die entsprechende Laufzeitumgebung zu sorgen.)

Zum Kreieren neuer und Finden existierender Komponenten sehen die zu Grunde liegenden Infrastrukturen externe Schnittstellen für Clients vor. Auch dort kommt ein *Interception*-Mechanismus zum Einsatz. Kreiert ein Client eine neue Komponente, so fängt die Infrastruktur den entsprechenden Aufruf ab. Dadurch ist es möglich, Komponenten erst bei wirklichem Bedarf zu instanziieren (*Just-in-Time-Activation*) oder vorsorglich anzulegen (*Pooling*). Beide Ansätze führen zu verbesserter Performanz und Ressourcennutzung.

Programmiersprachenunabhängige Komponententechnologien stellen Hochsprachen zum Im- und Export von Komponentenschnittstellen zur Verfügung. Aus solchen *IDL*-Spezifikationen (*Interface Definition Language*) erzeugen Generatoren sowohl den notwendigen Code zur Integration der Komponentenimplementierung in den Container als auch den Code zur Kommunikation zwischen Clients und Komponenten.

Bei Ressourcenknappheit und anderen Anlässen kann der Container Komponenten temporär deaktivieren, um sie bei Bedarf später wieder zu reaktivieren. Grundlegende Voraussetzung hierfür ist die Speicherung von Komponentenzuständen auf persistenten Medien. Das Laden und Speichern des vom Entwickler festzulegenden Zustands kann manuell durch die Komponente (komponenten-verwaltete Persistenz) oder automatisch durch den Container (container-verwaltete Persistenz) erfolgen. Der Container benachrichtigt die Komponente zu diesem Zweck über deren Rückrufschnittstellen sowohl bei Aktivierungs- als auch bei Deaktivierungsereignissen. Komponenten mit selbst verwalteter Persistenz schreiben daraufhin ihren Zustand in eine Datenbank oder restaurieren ihn von dort, während sich bei container-verwalteter Persistenz die Komponente um nichts zu kümmern braucht.

Für unterschiedliche Anwendungsfälle existieren unterschiedliche Komponententypen. Zu unterscheiden ist zum Beispiel, ob Komponenten ausschließlich einem Client zugeordnet sind oder mehrere Clients durch Bereitstellung eines Primärschlüssels Zugriff auf Komponenten erlangen können. Zusätzlich sind die Persistenz von Komponenten sowie die Kopplung ihrer Lebensdauer an die des Containers ein wichtiges Differenzierungsmerkmal. Je nach Komponententyp lassen sich unterschiedliche Ausprägungen der Schnittstellen zwischen Komponente und Container sowie zwischen Client und Komponente bereitstellen.

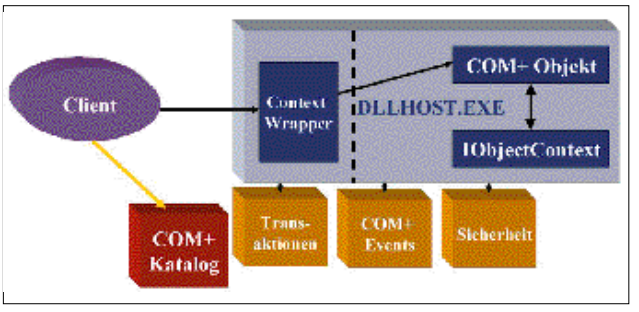

Abb. 3: Microsoft COM+ (In COM+ werden COM+-Komponenten paketweise in einen Surrogate-Prozess „DLLHOST.EXE" integriert. Dieser Container stellt über den Kontext-Wrapper ein „Interception"-Proxy bereit. COM+ Objekte müssen ihren Zustand selbst verwalten und können dabei über die Schnittstelle „IObjectContext" auf die Mechanismen des Transaktionsmonitors zugreifen.)

Die Auslieferung von Komponentenimplementierungen und aller benötigten Ressourcen erfolgt in binären Paketen, die zusätzlich Metainformation enthalten. Dadurch sind Entwicklungsumgebungen in der Lage Komponentenerstellung und -verwendung visuell zu unterstützen und sie in den Zielcontainer zu integrieren.

Im Gegensatz zu herkömmlichen Präsentationskomponenten gestaltet sich die Rollenverteilung bei Serverkomponenten wesentlich komplexer. Neben dem Komponentenlieferanten und dem Anwendungsentwickler gibt es den für Installation und Konfiguration zuständigen *Deployer*, den Lieferanten des Containers, den Lieferanten der Betriebssystem- und Middleware-Infrastruktur, und zu guter Letzt den für den laufenden Betrieb zuständigen Systemadministrator.

Die Wettbewerber

Das digitale Nervensystem: Microsoft COM+

Mit dem *Component Object Model (COM)* hat Microsoft vor vielen Jahren zunächst einen erfolgreichen Standard für Präsentationskomponenten auf Windows-Plattformen etabliert und damit den Startschuss für Komponentenarchitekturen abgegeben. Die wachsende Bedeutung des Enterprise-Computing hat Microsoft frühzeitig auf den Plan gerufen. Mittlerweile lassen sich COM-Objekte auch zwischen Maschinengrenzen nutzen.

Zu den primären Eigenschaften von COM zählt die Fähigkeit Funktionalität über mehrere transiente Schnittstellenzeiger nach außen zu exportieren. Da Clients nur auf Schnittstellenzeiger und niemals auf die eigentliche Komponente zugreifen, existiert eine Basisschnittstelle IUnknown, von der sich sämtliche COM-Schnittstellen ableiten. In dieser Basisschnittstelle ist neben Methoden zum Lebenszyklusmanagement die Methode QueryInterface zum Navigieren zwischen COM-Schnittstellen einer Komponente vorhanden. Die Implementierung von Komponentenklassen erfolgt entweder in dynamischen Bibliotheken, die in den Adressraum des Clients geladen werden, oder über lauffähige *EXE*-Server. Jeder Server – egal ob Bibliothek oder ausführbares Programm – stellt eine oder mehrere Fabriken (*Factories*) zur Instanziierung von Komponenten bereit, meistens implementiert durch die Standardschnittstelle IClassFactory. Sowohl die Referenzierung der Komponentenklassen als auch die der Schnittstellen geschieht über eindeutige Identifikatoren. Die Aktivierungsmechanismen des Laufzeitsystems sorgen für eine transparente Aktivierung von COM-Implementierungen und die Verbindung des Clients zu den entsprechenden Fabriken. Die dazu notwendigen Informationen sind in der *Registry* abgelegt. Mit Hilfe von *Proxies* und *Stubs* erfolgt eine nahtlose und ortstransparente Integration von Clients und Servern.

Vor wenigen Jahren hat Microsoft durch den *Microsoft Transaction Server (MTS)* zusätzlich für eine Trennung von Komponentenverhalten und -zustand gesorgt. Dort können COM-Objekte im Kontext einer Transaktion zusammenarbeiten. In Fehlersituationen sorgt MTS dafür, dass der ursprüngliche Zustand der Objekte wieder restauriert wird, wobei sich der Zustand von Objekten in persistenten Speichermedien, wie „SQL Server", oder transienten Ressourcemanagern, wie z. B. dem *Shared Memory Dispenser*, befinden kann. MTS-Objekte müssen demzufolge selbständig für das Laden und Speichern ihres Zustands sorgen.

Windows 2000 löst sowohl COM als auch MTS mit der aufwärtskompatiblen Technologie COM+ ab. Letztendlich handelt es sich hier um eine Verschmelzung von COM mit MTS, zusätzlich angereichert um weitere Basisdienste. Die Kooperation der MTS- bzw. COM+-Objekte erfolgt in Form sogenannter Aktivitäten (*Activities*), die sich als logischer, über Maschinengrenzen agierender Thread betrachten lassen. Die eindeutige Identifikation von Aktivitäten bewerkstelligt COM+ über automatisch mitgelieferte Identifikatoren, sogenannte *Casuality IDs*.

Zur Synchronisation paralleler Aktivitäten stehen diverse Sperrmechanismen zur Verfügung. Zusammengehörige COM+-Objekte werden in Form von *Dynamic Linked Libraries (DLLs)* zu Paketen (*Packages*) zusammengefasst und dem Laufzeitsystem über ein Administrationswerkzeug beziehungsweise MMC-SnapIn (Microsoft Management Console, COM+-Explorer) bekannt gemacht. Einträge über neue COM+-Objekte und ihre benötigten Anforderungen sind dabei im separaten COM+-Katalog abgelegt. Der Ablauf der Pakete geschieht geschützt in einem Container mit dem Namen DLLHOST.EXE (siehe Abb. 3). Notwendige Vorbedingung hierzu ist die Existenz von Typbibliotheken, die dem Container alle relevanten Informationen über die jeweiligen COM+-Objekte zur Verfügung stellen. Der Container hängt sich zwischen Client und Komponentenimplementierung. Mittels *Interception* ist er in der Lage, Instanziierungswünsche von Clients abzufangen, um Objekte bei Bedarf zu instanziieren oder auf Vorrat zu halten. Zu diesem Zweck erhält der Client niemals Zugriff auf das eigentliche Objekt, sondern stets nur auf ein Zwischenobjekt, den sogenannten Kontext-*Wrapper*, der die gleichen Schnittstellen wie das Objekt anbietet und die Funktion eines *Interception*-Proxies ausübt. Dadurch kann der Container dem Objekt stets die richtige Laufzeitumgebung offerieren. Um Zugriff auf die Laufzeitumgebung zu erlangen, also etwa um Transaktionen zu steuern, greifen COM+-Objekte mittels einer globalen API-Methode auf die Schnittstelle IObjectContext zu und können dadurch die Funktionalität des umgebenden Containers nutzen. Im Detail lassen sich dort Sicherheitsabfragen stellen und Transaktionen steuern. Implementiert ein COM+-Objekt die Rückrufschnittstelle IObjectControl, ist der Container in

der Lage dem Objekt mitzuteilen, sobald er es aktivieren oder deaktivieren möchte.

In COM+ können sich alle Objekte mit identischen Anforderungen an die Laufzeitumgebung einen gemeinsamen Kontext – beziehungsweise in „traditioneller" COM-Terminologie – ein Apartment teilen. Zwischen Kontexten existieren Proxies, um bei Methodenaufrufen vermitteln zu können. Das hat eine reduzierte Performanz zur Folge, weil ein Kontextwechsel letztendlich auch einen Threadwechsel impliziert. In COM standen bislang nur das *Single-threaded Apartment (STA)* für den serialisierten Zugriff durch einen einzigen Thread und das *Multi-threaded Apartment (MTA)* für gleichzeitige Nutzung durch mehrere Threads zur Verfügung. STAs zeichnen sich durch geringere Performanz aus, MTAs durch höhere Performanz bei größerem Implementierungsaufwand. In COM+ hat nun auch das *Neutral Apartment* Einzug gehalten, das beide Modelle miteinander vereint. Ein neutrales Apartment läßt sich von unterschiedlichen Threads durchlaufen (wie ein MTA), aber zur gleichen Zeit kann sich nur ein Thread im Besitz des Apartments befinden (wie bei STAs). Neutrale Apartments lassen sich somit von beliebigen Clients nutzen, ohne einen Threadwechsel zu erfordern.

Neben der Standardfunktionalität bietet COM+ zwei neue Basisdienste:

- Mittels *Queued Components* lassen sich Methodenaufrufe asynchron und zeitentkoppelt versenden. Der zu Grunde liegende „Microsoft Message Queue Server" (MSMQ) sorgt mit seiner *Store-and-Forward*-Architektur für die reibungslose Übermittlung und Zwischenspeicherung von Aufrufen.
- Der *COM+ Event Service* schließt eine weitere Lücke. Waren bislang keine effizienten und standardisierten Mechanismen zur Ereignisübermittlung zwischen verteilten Komponenten möglich, ändert sich dies mit dem COM+-Dienst, der sowohl die Registrierung von Ereignislieferanten als auch das Abonnement ihrer Ereignisse durch interessierte Teilnehmer ermöglicht. Dabei lassen sich gewünschte Ereignistypen auf beiden Seiten filtern. Das Broadcasting von Ereignissen an Hunderte oder Tausende von Abonnenten ist aufgrund der Implementierung des *COM+ Event Service* über ein Unicast-Protokoll heute noch nicht zu empfehlen.

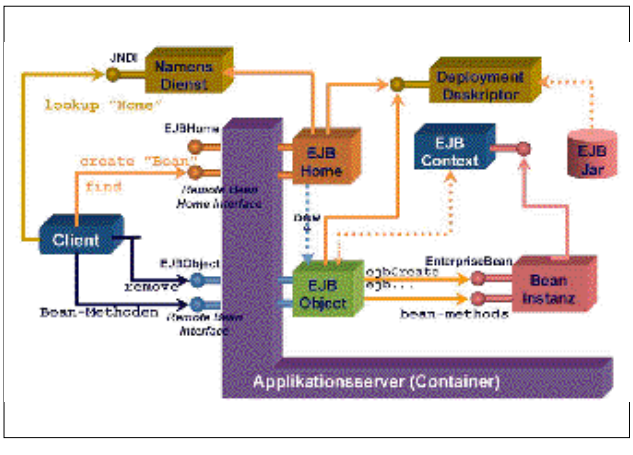

Abb. 4: EJB (Das komplizierte Bild t uscht: In EJB stellt der Entwickler lediglich die Applikationsschnittstelle, ein Home-Interface und die Implementierung der Applikationslogik bereit. Den Rest erledigen Programmierumgebung und Container.)

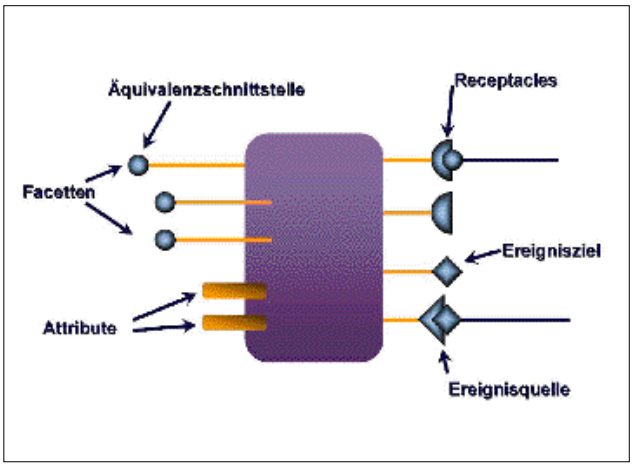

Abb. 5: Ports (CORBA-Komponenten k nnen im Gegensatz zu EJB mehrere Schnittstellen im- bzw. exportieren. Die quivalenzschnittstelle stellt dabei die ausgezeichnete Schnittstelle einer Komponente dar.)

Geschäftsbohnen: Enterprise JavaBeans 1.1

Mit ihren Namensvettern, den JavaBeans, haben *Enterprise JavaBeans (EJBs)* nahezu keine Gemeinsamkeit. Als Bestandteil der *Java 2 Enterprise Edition (J2EE)* stellen sie einen zentralen Baustein für die Entwicklung verteilter Applikationen dar. Dem Client präsentieren sich EJB-Komponententypen zunächst über ein sogenanntes *Home-Interface*, mit dessen Funktionalität sich Komponenteninstanzen erzeugen (create), löschen (remove) oder – bei Vorhandensein eines optionalen Primärschlüssels – auch gezielt finden lassen (find). Eine Referenz auf das *Home* eines gewünschten Komponententyps erhält der Anwender dabei durch die Programmierschnittstelle des Verzeichnisdienstes – JNDI (siehe Abb. 4). Auf eine frisch instanziierte oder eine bereits existierende Komponente greift der Client über deren einzige Schnittstelle, das *Remote Interface*, zu. Sowohl *Home*- als auch *Remote-Interface* sind aus Gründen der Uniformität als *RMI*-Schnittstellen (*Remote Methode Invocation*) zu definieren. Dies impliziert allerdings keine Nutzung von RMI, sondern nur eine Anwendung der RMI-Programmierkonventionen. Grundsätzlich ist jede Art von Kommunikationsmechanismus möglich.

Der Komponentenentwickler stellt neben den genannten Schnittstellen auch eine Java-Klasse zur Verfügung, die sowohl die Anwendungslogik als auch weitere Funktionalität implementiert, zum Beispiel die genannten create- und find-Methoden des *Home-Interface*.

Aufgabe des verwendeten EJB-Produkts ist die Einbettung der Komponententypen in den Container. Dazu liest ein Generator die *Class-Files* des *Remote*- und des *Home-Interface* und erzeugt daraus konkrete Objektimplementierungen in Form einer EJBObject- und einer EJBHome-Instanz. Der externe Client greift auf diese Instanzen zu, die sich ihm gegenüber als das tatsächliche *Home* und die tatsächliche Komponentenschnittstelle gebärden. In Wirklichkeit implementieren EJBObject und EJBHome den bereits vorgestellten *Interception*-Mechanismus. Sie fangen Aufrufe der Clients ab, stellen für die Komponente die notwendige Laufzeitumgebung bereit und delegieren

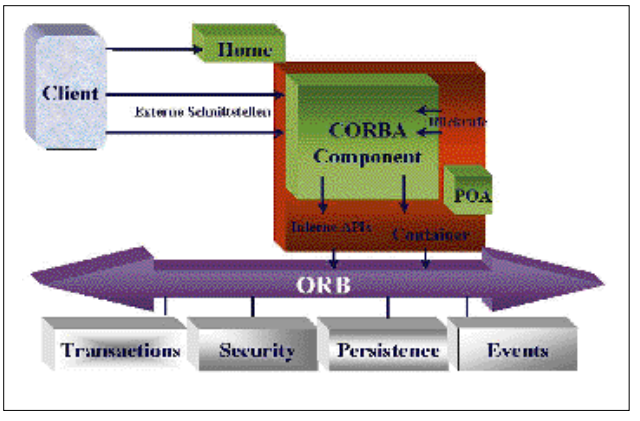

Abb. 6: CORBA-Komponentenarchitektur (CORBA-Komponenten sind mit EJB nahe verwandt. CORBA-basierte Container beruhen auf den Mechanismen des POA. Interne Schnittstellen dienen dem Zugriff auf Funktionalit t des Containers. ber R ckrufschnittstellen kann der Container die Komponente ber Ereignisse in deren Lebenszyklus informieren.)

den Aufruf dann an die vom Entwickler bereitgestellten Implementierungsklasse.

Damit ein Container Komponenten über wichtige Ereignisse informieren kann, muss die Komponenten-Implementierungsklasse diverse Rückrufmethoden bereitstellen. Deren Aufruf erfolgt zur Laufzeit automatisch durch den Container. Zum Beispiel teilt ejbPassivate() der Komponente mit, dass der Container sie deaktivieren möchte, während ejbActivate() über eine (Re-)Aktivierung informiert. In den genannten Methoden kann der Entwickler alle erforderlichen Maßnahmen durchführen, also beispielsweise bei der Deaktivierung einer Komponente zuvor allokierte Dateideskriptoren freigeben.

Auch die umgekehrte Richtung, der Zugriff auf Funktionalität des Containers durch die Komponente, findet Unterstützung. Nach der Instanziierung einer Komponente durch den *Interception*-Mechanismus des Containers übergibt dieser der Komponente ein Kontextobjekt. Das Kontextobjekt gewährt Zugriff auf die Programmierschnittstellen des Containers. Dort lassen sich etwa Informationen über Aufrufer abrufen oder Transaktionen steuern.

Die EJB-Spezifikation unterscheidet zwei Arten von Komponenten:

- „Session Beans“ und
- „Entity Beans“

Session Beans sind mangels Primärschlüssel einem Client fest zugeordnet und in ihrer Lebenszeit an den Container gekoppelt. Diese Komponentenart untergliedert sich wiederum in zwei Varianten: *Stateless Session Beans* merken sich ihre Vorgeschichte nicht, während *Stateful Session Beans* aufgrund von Methodenaufrufen ihren durch die Datenfelder gegebenen Zustand ändern, der auch die Ergebnisse nachfolgender Methodenaufrufe beeinflusst. Da dieser Zustand durch Konversation zwischen dem Client und der Komponente zustande kommt, ist von *Conversational State* die Rede. Bei der Deaktivierung einer Session-Komponente und einer anschließenden Reaktivierung sorgt der Container automatisch für die Sicherung beziehungsweise Restauration des *Conversational State* der Komponente, beispielsweise durch Nutzen der Serialisierungsschnittstelle. Darüber hinaus kann die Komponente selbständig Informationen in Datenbanken verwalten, was dem Container allerdings verborgen bleibt. *Stateful Session Beans* können zudem eine spezielle Schnittstelle implementieren, mit der ein Container sie bei Transaktionen über den Fortlauf informiert.

Im Gegensatz zu *Session Beans* lassen sich *Entity Beans* durch mehrere Clients nutzen und fungieren als persistente Komponenten, deren Lebenszeit nicht an die des Containers gekoppelt ist. Zu diesem Zweck ist *Entity Beans* ein Primärschlüssel zugeordnet, der eine eindeutige Identifikation von Komponenten gestattet und sich üblicherweise aus einer Kollektion von Datenfeldern zusammensetzt. Im *Home-Interface* sind neben den bei Session-Komponenten üblichen create-Methoden auch find-Methoden zur Suche existierender Komponenten verfügbar. Der Zustand einer Entity-Komponente, also der Inhalt ihrer Datenfelder, ist mit einem oder mehreren Einträgen in Datenbanken assoziiert. Die Aufgabe des Containers besteht darin, bei Aktivierung oder „Passivierung“ von Entity-Komponenten beziehungsweise bei deren Partizipation an Transaktionen für den konsistenten Abgleich zwischen persistentem Speichermedium und Komponente zu sorgen. Der Zugriff auf das Speichermedium kann entweder automatisch durch den Container erfolgen (container-verwaltete Persistenz) oder durch die Komponenten selbst (komponenten-verwaltete Persistenz).

Zur Auslieferung einer Komponentenimplementierung muss der Entwickler die class-Dateien für Schnittstellen und Komponentenimplementierung und alle zugehörigen Ressourcedateien in eine komprimierte Archivdatei (Endung: ejb-jar) verpacken. Zusätzlich stellt er dort einen XML-basierten *Deployment*-Deskriptor bereit, der sämtliche Metainformationen enthält – etwa über Struktur der Komponente, gewünschtes Verhalten bei Transaktionen, Sicherheitsanforderungen oder persistente Speicherung. Diese Information nutzt der EJB-Container, um der Komponente die gewünschte Laufzeitumgebung zu bereiten. EJB bedient sich übrigens des CORBA-Transaktionsdienstes, um Transaktionsmonitore mit 2-Phasen-Commit-Protokoll zu integrieren. Auch hinsichtlich der Kompatibilität ergibt sich eine enge Verzahnung mit CORBA, was der nachfolgende Abschnitt näher beleuchtet.

Free CORBA: CORBA-Komponentenmodell

CORBA Components (CORC) repräsentieren den zentralen Bestandteil des CORBA-3-Standards und sind sowohl für unterschiedliche Programmiersprachen als auch für unterschiedliche Betriebssysteme verfügbar. Das *CORBA Component Model (CCM)* orientiert sich dabei derart stark an EJB, dass sich beide Komponententechnologien als unterschiedliche Seiten derselben Medaille betrachten lassen. An dieser Stelle sollen daher die durch CCM eingebrachten Erweiterungen im Vordergrund stehen.

Im Gegensatz zu EJB können CORBA-Komponenten mehrere Schnittstellen, sogenannte *Ports*, spezifizieren. Über diese exportieren sie Funktionalität nach außen oder importieren externe Funktionalität (siehe Abb. 5). *Facettes* dienen dem Export von Funktionalität, *Receptacles* definieren Eingänge zum Import von externen Schnittstellen. Des Weiteren kann eine Komponente als *Emitter* Ereignisse an einen einzelnen Empfänger oder als *Publisher* sogar an beliebig viele Empfänger senden. Mittels *Subscriber*-Schnittstellen fungiert eine Kom-

ponente selbst als Ereignisempfänger. Jede CORBA-Komponente implementiert beliebig viele Schnittstellen, muss aber in jedem Fall eine sogenannte Äquivalenzschnittstelle anbieten, die gleichsam die Komponente selbst repräsentiert. Teil der Äquivalenzschnittstelle sind zum Beispiel Methoden, um zu anderen Schnittstellen zu navigieren. Jede CORBA-3-Schnittstelle enthält zudem die Methode get_component(), mit der eine Rücknavigation zur Äquivalenzschnittstelle möglich ist – vorausgesetzt, es handelt sich wirklich um eine Komponente. Durch Ableiten einer *Facette* von einer speziellen Navigationsschnittstelle lässt sich auch von dort zu jeder anderen Schnittstelle navigieren. Zur Spezifikation einer Komponente dient die Hochsprache „Component IDL".

Neben der Applikationsschnittstellen eines Komponententyps existiert auch in CORBA ein *Home-Interface*. Der Client lokalisiert zunächst unter Nutzung der globalen Operation resolve_initial_references() den ComponentHomeFinder und beschafft sich dort eine Referenz auf das gewünschte *Home*. Analog zu EJB dient das *Home* primär zur Instanziierung neuer oder zum Zugriff auf existierende Komponenten.

Die notwendige *Interception*-Strategie erfolgt bei CORBA-Komponenten im Rahmen des bewährten *POA*-Modells (*Portable Object Adapter*) (siehe Abb. 6). Ein POA repräsentiert einen Lebensraum von CORBA-Objekt-Implementierungen, den sogenannten *Servants*.

	CCM	EJB	COM+
Architekturkonzept	Container & Interception	Container & Interception	Container & Interception
Muster zum Finden/Kreieren	Finder & Factory	Finder & Factory	Factory
Schnittstellen pro Komponente	Mehrere Interfaces	Ein Interface	Mehrere Interfaces
Aktivierung durch ...	Container & POA	Container	Container & Laufzeitproz.
Komponentenarten	Session, Service, Process, Entity	Session, Entity	„Stateless" Session
Kommunikationsprot.	IIOP	IIOP/RMI	DCE/SOAP
Persistenz	PSS	Serializat.	Datenbank
Ereignisdienst	Notification Dienst	Nicht unterstützt	COM+ Events
Transaktionsmonitor	OTS	JTS/OTS	„MTS"
Sicherheitssupport	CORBA Security	Java Security Package	COM+ Security
Asynchronous Komm.	AMI/TII	JMS	Queued C.
Verzeichnisdienst	CosNaming	JNDI	ADSI
Skalierbarkeit	++	+	0
Verfügbare Interoperab.protokolle	RMI-IIOP/COM	RMI-IIOP	IIOP
Integrationsmgl. mit ...	EJB	CCM	n.a.
Installation und Konfiguration	Installations Deployment Prozess	Abhängig von Hersteller	COM+ Explorer + Registry
Metadaten	XML	XML	Proprietär
Adminstrationssupp.	--	--	--
Ad Hoc Networking	Nicht vorh.	Jini	UPnP

Tabelle 1: Vergleich (Die drei Komponentetechnologien weisen sowohl gravierende Unterschiede als auch große Gemeinsamkeiten auf.)

Zu unterscheiden ist zwischen dem CORBA-Objekt (einem virtuellem Konzept) und seiner konkreten Implementierung, dem *Servant*. Im Laufe der Lebenszeit eines CORBA-Objekts können verschiedene *Servants* dasselbe CORBA-Objekt implementieren. Im selben Server können mehrere baumartig angeordnete POAs existieren. Jeder POA besitzt dabei einen eigenen Satz von standardisierten Richtlinien. Beispielsweise definiert eine der möglichen Richtlinien, ob ein gegebenes CORBA-Objekt persistent oder transient sein soll. Eine weitere Richtlinie legt fest, ob mehrere CORBA-Objekte vom selben *Servant* inkarniert werden dürfen oder nicht. Der POA ist unter anderem dafür verantwortlich *Servants* zu aktivieren und zu deaktivieren sowie eine eindeutige Abbildung von CORBA-Objektreferenzen auf die entsprechenden *Servants* vorzunehmen. Normalerweise merkt sich der POA die Abbildung von CORBA-Objekten auf *Servants* in einer Objekttabelle. Eine Implementierung kann aber auch einen sogenannten *Servant Manager* beim POA anmelden. Trifft ein Methodenaufruf eines Clients zu einem CORBA-Objekt ein, für das kein Eintrag in der Objekttabelle existiert, delegiert der POA diesen Aufruf an den registrierten *Servant Manager*. Im Fall der Variante *Servant Locator* kann die Implementierung alle Aufrufe abfangen. Fertig ist der CORBA-basierte Komponentencontainer mit integriertem *Interception*-Mechanismus.

Zur Interaktion zwischen Container und Komponenten definiert auch CCM Rückrufschnittstellen, über die der Container Ereignisse an Komponenten meldet, sowie ein Kontextobjekt, das der Container der Komponente nach deren Instanziierung übergibt und das dem Zugriff auf die Funktionalität des Containers dient.

Neben den EJB-Komponentenarten *Session Bean* und *Entity Bean* definiert das CORBA-Komponentenmodell zusätzlich Servicekomponenten und Prozesskomponenten. Erstere tragen keinen Zustand und sind transient; letztere besitzen einen persistenten Zustand und sind nur einem singulären Client zugeordnet.

Zur persistenten Speicherung von Komponentenzuständen steht in CORBA 3 der *Persistent State Service* zur Verfügung. Ähnlich wie CORBA-Schnittstellen abstraktes Verhalten modellieren, dienen *Storage*-Objekte dort der abstrakten Modellierung von Speicherobjekten. Aus einer abstrakten Beschreibung erzeugt ein Generator Zugriffsfunktionalität auf die Speicherobjekte. Für den Nutzer ist nicht sichtbar, ob sich dahinter eine relationale Datenbank oder ein anderes Speichermedium verbirgt. Ein Anwender spezifiziert für seine Komponenten *Segmente*, die eine Teilmenge der Schnittstellen zusammenfassen und ein gemeinsames Speicherobjekt zur Zustandsspeicherung nutzen. Weil Segmente separat aktivierbar sind, lässt sich die Ressourcennutzung optimieren. Wie in EJB kann eine Komponente für ihre Persistenz Verantwortung tragen oder dies dem Container überlassen.

Die Auslieferung von CORBA-Komponenten erfolgt ebenfalls in Archivdateien (Endung .car). Analog zu EJB definiert CORBA standardisierte XML-basierte Deskriptoren zur Beschreibung dieser Komponentenpakete. Zusätzlich definiert CCM einen kleinen Installationsprozess für Komponenten.

Dank des bereits reichhaltigen Satzes an standardisierten CORBA-Diensten nutzen CORBA-Komponenten-Container den CORBA-Transaktionsdienst zur Einbindung eines Trans-

aktionsmonitors, den CORBA-Sicherheitsdienst für Sicherheitsaspekte, den Benachrichtigungsdienst zur Ereignismeldung und den *Persistent State Service* für die Zustandsspeicherung. Mit *CORBA Messaging* steht zudem ein Standard für asynchrone und zeitversetzte Kommunikation zur Verfügung. Hinsichtlich paralleler Verarbeitung erlaubt das CORBA-Komponentenmodell im Gegensatz zu EJB auch parallele Verarbeitung.

Ying und Yang: CORC und EJB

Ziel der CORBA-Komponentenspezifikation war von Beginn an die problemlose Kompatibilität mit Enterprise JavaBeans. Daher sind java-basierte CORBA-Komponenten bei Beachtung kleinerer Einschränkungen problemlos in EJB-Containern nutzbar. Umgekehrt repräsentieren EJB-Komponenten gleichzeitig CORBA-Komponenten und sind demzufolge in CCM-Containern verwendbar. Wegen der häufigen Verwendung von CORBA als Kommunikationsbasis von EJB-Produkten ist auch die Interoperabilität auf dieser Ebene gewährleistet. Von der Integration von CORBA-Diensten in J2EE war bereits an anderer Stelle die Rede. Insgesamt lassen sich CORBA- und EJB-Komponenten somit als Zwillingsschwestern betrachten. Die enge Verwandtschaft hat auch Synergieeffekte zur Folge, sobald Produkthersteller ihre bisherigen EJB-Produkte zu vollständigen CCM-Implementierungen erweitern wollen.

Fazit

Galten CORBA, COM und RMI bis vor wenigen Jahren noch als mächtige und exotische Standards zur Entwicklung verteilter Systeme, hat sich die Situation inzwischen geradezu dramatisch verändert. Heute gehören die Programmierung verteilter Systeme und die Nutzung von Middleware zum Tagesgeschäft. Für heutige Anforderungen genügt ein entfernter objektorientierter Methodenaufruf jedoch nicht mehr. Es bedarf in der Regel zusätzlicher infrastruktureller Dienste, wie Transaktionen, Sicherheitsmechanismen oder Ereignisdienste. Serverbasierte Komponententechnologien – wie COM+, CORC oder EJB – repräsentieren unter dem Schlagwort „Applikationsserver“ integrierte Lösungen, die den Schwerpunkt speziell auf die Web-Entwicklung legen. Trotz ihrer architektonischen Übereinstimmung weisen die genannten Technologien essenzielle Unterschiede auf. COM+ unterstützt – als integraler Bestandteil von Windows 2000 – zwar mehrere Programmiersprachen, aber nur eine Betriebssystemplattform. EJB läuft auf nahezu allen Betriebssystemplattformen, kommt aber nur mit einer Programmiersprache zurecht. CORC – als vollständigstes und flexibelstes Modell – unterstützt zwar mehrere Programmiersprachen und Betriebssysteme, existiert vorläufig aber nur auf dem Papier. Durch gegenseitige Kompatibilität lassen sich EJB und CORC als zwei Seiten derselben Medaille betrachten, weshalb sich die Welt wie gewohnt in Microsoft und den „Rest der Welt“ unterteilen lässt (siehe auch Tabelle 1).

Bei aller Euphorie sind aber auch die Defizite der Technologien zu berücksichtigen. Für die Entwicklung von Echtzeitsystemen oder eingebetteten Systemen eignen sich aufgrund ihrer Ressourcenanforderungen sowie der fehlenden Vorhersagbarkeit ihres Laufzeitverhaltens weder EJB, noch CORC, noch COM+. Zur Verifikation der Erfüllbarkeit von Anforderungen in anderen Domänen bedarf es eines Prototyping-Ansatzes. Das Testen und Debuggen verteilter komponentenbasierter Systeme erfolgt nach wie vor mühsam und weitgehend manuell. Ebenso finden Administration und Konfiguration nur stiefmütterliche Behandlung seitens der Hersteller. Im CORBA- und Java-Bereich gibt es zwar eine große Palette an Produkten, die sich in der angebotenen Funktionalität gegenseitig zu übertreffen versuchen, was bei Portierungsaufgaben zu ungeahnten Überraschungen führen kann. Auch weisen die Spezifikationen noch einigen Raum für Verbesserungen auf. Die Interoperabilität zwischen COM+ und EJB beziehungsweise CORC ist heute nur über spezielle Lösungen möglich. Mit dem XML-basierten *Simple Object Access Protocol (SOAP)* gibt es aber einen Lichtblick am Horizont.

Trotz dieser Minuspunkte eignen sich viele Projekte für den Einsatz von Komponententechnologien. Wer heute noch auf Eigengewächse setzt, handelt meist fahrlässig und dürfte dies in naher Zukunft bereuen. Nicht zuletzt ist die Verwendung von Middle-Tier-Komponenten industrieweit unumstritten. Zu den „Sympathisanten“ gehört die Creme de la Creme des Internet-Zeitalters. Zu erwähnen sind hier etwa IBM, Sun, Microsoft, Corel, SAP, Novell und Oracle. Kurz und prägnant zusammengefasst: Es ist an der Zeit sich intensiv mit Komponententechnologien beschäftigen, sich dabei aber an den Realitäten und nicht am „Hype“ zu orientieren. Auch in diesem Kontext gilt das inzwischen sattsam bekannte Motto: Wer zu spät kommt, den bestraft das Leben.

Literatur

[Mat99] V. Mathena, M. Hapner, Enterprise JavaBeans 1.1, Sun Microsystems, 1999

[OMG99] OMG Dokument 99-02-05 (CORBA Komponentenspezifikation)

[Pla99] D.S. Platt, Understanding COM+, Microsoft Press, 1999

[Bus96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, Pattern-Oriented Software Architecture – A System of Patterns, Wiley and Sons, 1996

[Sch00] D.C. Schmidt, M. Stal, H. Rohnert, F. Buschmann, Pattern-Oriented Software Architecture 2 – Patterns for Distributed and Concurrent Objects, Wiley and Sons (erscheint 10/2000)

[Sta00] M. Stal, Episode 3 , Teil 1: Komponenten, in: iX-Magazin 4/2000