Javamagazin Sonderdruck aus JM 6.2003

c.a.r.u.s. cJEF:
J2EE-Framework zur effektiven Entwicklung serverseitiger Geschäftslogik

von Béla Bodnár

Bauen in Schichten

Bei der Anwendung der J2EE-Architektur gibt es eine Reihe von Gestaltungsfreiräumen und Grenzen, die zum Projektrisiko werden können. Die Anwendung der J2EE bedarf daher der Festlegung einer eigenen Architektur, die über die Standardarchitektur hinausgeht. Hier ist vor allem die Entscheidung über die Verwendung der EJB-Komponenten wichtig. Es stellt sich die entscheidende Frage, wie Geschäftslogik in einem Applikations-Server abgearbeitet und gleichzeitig effizient mit einer Präsentationsschicht zusammenarbeiten kann.

In diesem Artikel wird eine Architektur vorgestellt, die die serverseitige Implementierung von Geschäftslogik angemessen unterstützt. Außerdem wird ein Verfahren erläutert, mit dem der Komplexität der Entwicklung von verteilten Anwendungen entgegentreten werden kann. Mit dem Einsatz des Java Enterprise Frameworks (cJEF) der Firma c.a.r.u.s. wird zunächst ein Java „stand alone" Programm entwickelt und getestet. Unterstützt durch das Framework wird dabei eine logische Trennung in Präsentations-, Business- und Datenzugriffschicht vorgenommen. Die Trennung zwischen Präsentations- und Businesskomponenten stellt ein zentrales Konzept dar und erweitert das von Smalltalk her bekannte MVC-Entwurfsmuster [2] um die Eigenschaften, die für eine effektive Verteilbarkeit notwendig sind. Diese Schichtentrennung ist im Framework so ausgelegt, dass für die Verteilung des fertigen Programms auf mehrere Server dann lediglich ein Konfigurationsschritt notwendig ist.

Der Kern einer kommerziellen bzw. fachlichen Anwendung besteht heute üblicherweise aus der Modellierung von Anwendungsfällen und Geschäftsobjekten. Die J2EE-Architektur wird zurzeit als die Lösung aller technischen Hindernisse beim Aufbau einer gut strukturierten, verteilbaren Anwendungsarchitektur propagiert. Dabei werden häufig Session Beans als Abbildung von Anwendungsfällen und Entity Beans als Abbildung von Geschäftsobjekten gesehen. Diese Sichtweise reicht jedoch nach unserer Ansicht kaum aus, um die wirklichen Probleme komplexer, interaktiver Anwendungssoftware zu lösen. Dies hat zur Folge, dass viele größerer Projekte bei Anwendung der J2EE-Architektur mit der Komplexität Probleme bekommen. Gleiches gilt ebenfalls bei der Anwendung der .NET-Architektur von Microsoft.

Beim Aufbau einer Softwarearchitektur muss der fachliche Kern in den Mittelpunkt gestellt werden. Dieser besteht aus Anwendungsfällen (UseCases) und Geschäftsobjekten (Modelle). Es muss unterschieden werden zwischen logischer und physischer Trennung. Die logi-

Wandlung von Rich- zu Thin-Clients

Die physische Trennung bestimmt maßgeblich den gewählten Architektur-Oberbegriff. Hier hat sich in der Vergangenheit eine Wandlung (zumindest in der Zielsetzung) von sog. Rich-Clients (früher auch gerne Fat-Client genannt) zu Thin-Clients vollzogen. Die Reichhaltigkeit eines Clients muss man daran messen, in welchem Umfang fachliche Logik auf ihm ausgeführt wird. In der Praxis finden wir recht wenige Systeme, bei denen man wirklich von Thin-Clients sprechen kann. Dies liegt daran, dass meistens lediglich Datenzugriffe auf Applikations-Servern ausgeführt werden, während Geschäftsobjekte auf dem Client laufen. Dies gilt vor allem für GUI- (SWING), aber auch für Web-Anwendungen, bei denen der Webserver als Client auftritt.

© Copyright Software & Support Verlag

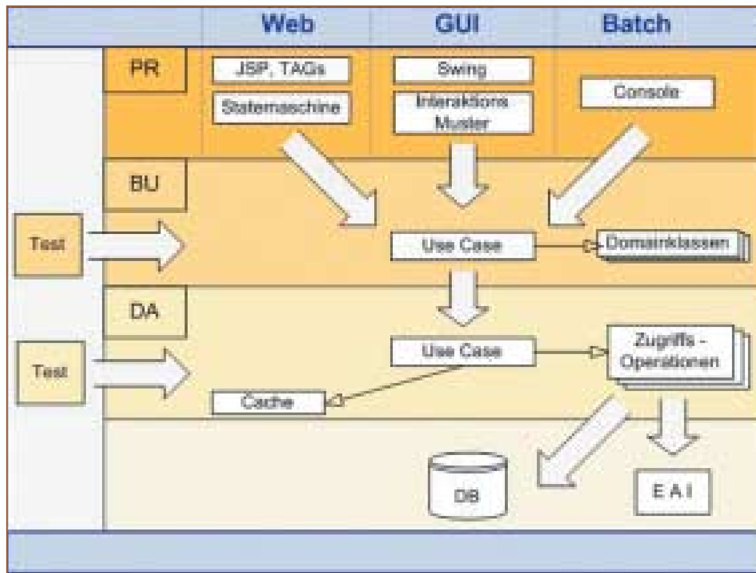

Abb.1: Schichtenmodell cJEF

sche Schichtung dient der sauberen Trennung verschiedener Verantwortlichkeiten. Hier ist mindestens eine Abgrenzung der Präsentations-, Geschäfts- und Datenzugriffslogik erforderlich. Die physische Schichtung ermöglicht die Verteilbarkeit einer Anwendung auf mehrere Rechner.

Die logische Trennung steht für den Anwendungsentwickler im Vordergrund, um die Softwarequalität zu sichern (Wartbarkeit, Erweiterungsfähigkeit). Die physische Trennung ist für den Betrieb wichtig, um die Betreibbarkeit zu gewährleisten (Skalierung, Performance, Ausfallsicherheit, Security, Deployment). Sie ist jedoch für den Anwendungsentwickler meist hinderlich, weil die Entwicklung und der Test verteilter Systeme wesentlich aufwändiger ist (Turn-Around-Zyklen, verteiltes Debugging etc.) als bei nicht verteilten Systemen, die sich in einem einzigen Prozessraum bewegen. Dies gilt auch, wenn die Verteilung auf einem Entwicklungsrechner in mehrere Prozesse vorgenommen wird (z.B. durch eine EJB-Testumgebung).

Eine moderne Anwendungsarchitektur muss die beiden Anforderungen zusammenbringen und die technischen Aspekte zur Verteilbarkeit transparent lösen. Gleichzeitig ist die Flexibilität notwendig, ein verteilbares System nicht zwangsläufig verteilen zu müssen. Idealerweise wird die Anwendungsarchitektur durch ein geeignetes Framework unterstützt bzw. weitestgehend darin implementiert. Durch Frameworkunterstützung und eine saubere Schichtentrennung entsteht eine Gesamtarchitektur, die neben den Softwarequalitätskriterien darauf ausgerichtet ist, die Entwicklungsproduktivität zu optimieren. Dabei gilt es, die wiederkehrenden, meist technischen Aufgaben, zentral zu implementieren, um einen maximalen Grad an Wiederverwendung zu erreichen. Für fachliche Funktionalität muss ein Rahmen geschaffen werden, der den Entwickler dabei unterstützt, denselben Wiederverwendungsgrad auch hier zu erreichen.

Einen solchen Ansatz verfolgt das c.a.r.u.s. Java Enterprise Framework (cJEF). Im Vordergrund steht dabei die Kernarchitektur und weniger die darüber hinaus umfangreich vorhandene Funktionalität in den verschiedenen Softwareschichten.

Architekturüberblick

Grundkonzept der Architektur ist ein Schichtenmodell. Die Schichtung erleichtert die Kapselung einerseits der fachlichen Logik, um komplexe Anwendungen erstellen zu können. Andererseits werden die technischen Aspekte der Verteilung so gekapselt, dass sie nicht mit fachlichen Aspekten vermengt werden müssen. Die Architektur wird in drei wesentliche Schichten aufgeteilt:

- Präsentation (PR-Schicht)
- Geschäftslogik (BU-Schicht)
- Datenzugriff (DA-Schicht)

Als Präsentationsarten werden unterstützt:

- Web-Anwendungen (basierend auf JavaServer Pages)
- GUI (basierend auf Swing)
- Batch-Anwendungen

Alle diese Anwendungstypen arbeiten mit einer einheitlichen Geschäftslogik und Datenzugriffsschicht zusammen. Abbildung 1 zeigt die Struktur einer geschichteten Anwendung in Bezug auf das Framework.

Als Ergebnis einer Anforderungsanalyse liegt in der Regel eine Sammlung von *Anwendungsfällen* (*Use Case*) vor, ergänzt durch Geschäftsregeln, die strukturelle und algorithmische Vorgaben enthalten. Die Anwendungsfälle sind ein zentraler Punkt für den Entwurf der Anwendung. Die Ablaufsteuerung der Anwendungsfälle und ihre Präsentation erfolgt in der Präsentationsschicht. Neben den Dialogkomponenten wird ein Anwendungsfall immer als präsentationsunabhängige UseCase-Klasse abgebildet. Aus den Anwendungsfällen wird ein fachliches Objektmodell entwickelt, dessen Implementierung in fachlichen Modellklassen erfolgt. Die fachliche Logik ist in der Geschäftslogikschicht angesiedelt. Die Persistenz von Modellklassen dient der permanenten Speicherung. Diese wird in der Regel in relationalen Datenbanken vorgenommen und ist Aufgabe der Datenzugriffsschicht.

Das Schichtenmodell ist so strukturiert, dass die einzelnen Schichten auf unterschiedlichen Rechnern laufen können. Dies ist vor allem für die Präsentationsschicht wichtig, um diese möglichst „schlank“ zu halten. Eine Verteilung ist jedoch nicht zwangsläufig notwendig und wäre während der Entwicklung kontraproduktiv. Daher ist die Verteilung optional, und eine zentral (d. h. auf einem Entwicklerrechner in einem einzi-

© Copyright Software & Support Verlag

gen Java-Prozess) entwickelte und getestete Anwendung kann später ohne Codeänderung in einer verteilten Umgebung laufen.

Anwendungskern

UseCase- und Modellklassen bilden den Kern einer Anwendung in der Geschäftslogikschicht. UseCases wirken wie eine Fassade auf die Modellklassen und dienen dazu, anwendungsfallspezifische Logik zu implementieren. Sie werden als transiente Klassen implementiert und halten den Bearbeitungszustand während der gesamten Dauer eines Anwendungsfalls z.B. für die Präsentationsschicht bereit. Modellklassen hingegen implementieren die anwendungsfallübergreifende Logik und werden in der Regel als persistente Klassen implementiert. Um beim Entwurf von UseCases eine maximale Freiheit bezüglich deren Granularität zu erzielen, können mehrere UseCases innerhalb eines UseCase-Managers gebündelt werden. Eine komplexe Benutzeraktion kann dadurch über eine Fassade auf mehrere UseCases und Modellinstanzen zugreifen. Dies entspricht den Empfehlungen, die in dem J2EE-Pattern Session Facade bezüglich der UseCase-Abbildung gegeben wird [3].

Während eines Anwendungsfalls werden Modellinstanzen neu erzeugt, geladen, verändert oder gelöscht. Diese Bearbeitung erfolgt zunächst im Hauptspeicher und wird erst mit der Beendigung des Anwendungsfalls in die Datenbank gespeichert. Dies ist notwendig, um die Konsistenz der Datenbank zu gewährleisten. Durch die Trennung in UseCase- und Modellklassen kann man dem Problem entgegenwirken, Modellklassen mit Funktionalität für viele Anwendungsfälle zu überladen, um einen stabilen und wartbaren Anwendungskern mit minimalen Abhängigkeiten zu erreichen. Zwischen den einzelnen Anwendungsschichten müssen vor allem die Daten von Modellinstanzen ausgetauscht werden. Diese Daten werden in spezielle Datenklassen ausgelagert, um die einzelnen Schichten möglichst wenig aneinander zu koppeln. Gleichzeitig ist durch die Modellierung von Referenzen als Object-IDs eine effiziente Serialisierung gegeben, die vor allem in einer verteilten Umgebung sehr wichtig ist. Ein ähnliches Muster finden wir in den J2EE-Patterns mit dem Data Access Object (DAO) [4] vor. In diesem Muster wird allerdings ausschließlich der Datenbankzugriff betrachtet.

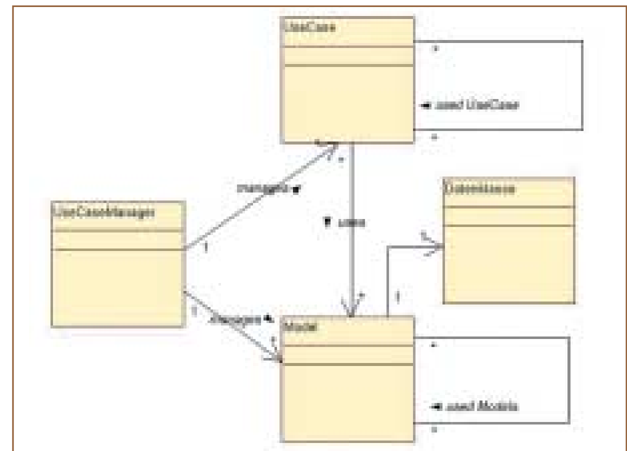

Abb.2: Klassenstruktur der fachlichen Modellierung

Es ergeben sich folgende wesentliche Klassen (Abb.2):

- Modellklassen: Basisklasse für die Abbildung aller fachlichen (persistenten) Anwendungsklassen.
- Datenklassen: Basisklasse für die Abbildung von Modelldaten.
- UseCase-Klassen: Basisklasse für die Abbildung eines Anwendungsfalls. Ein Anwendungsfall verwendet in der Regel eine Menge von Modellobjekten. Über die Benutzung anderer Anwendungsfallklassen kann die Funktionalität individuell aufgeteilt werden.
- UseCase-Manager: Verwaltet eine Benutzertransaktion und damit alle UseCase- und Modellinstanzen, die innerhalb einer Benutzertransaktion notwendig sind. Als Benutzertransaktion ist hier z.B. die Bearbeitung eines Auftrags mitsamt den zugehörigen Positionen und dem Kunden in einer Warenwirtschaft gemeint.

Schnittstellen zwischen den Schichten, physische Verteilung

Die logische Schichtung bildet die Grundlage für die physische Verteilung. Zunächst kann man zwischen den einzelnen Schichten eine Prozess- bzw. Maschinengrenze legen oder dies unterlassen. Dadurch hat man generell die Möglichkeit, die Präsentationsschicht auf einem Client zu betreiben, die Geschäftslogikschicht auf einem Applikations-Server und die Datenzugriffsschicht auf einem anderen Applikations-Server. Als Verteilungsgranularität ist der UseCase bzw. die Sammlung mehrerer kooperierender UseCases sinnvoll. Ein UseCase ist eine gut isolierbare Aufgabe, die zu ihrer Erfüllung normalerweise mit einer Vielzahl von Modellinstanzen zusammenarbeitet. Hier ist es nicht sinnvoll, diesen Komplex über mehrere EJBs zu verteilen, sondern zusammenzuhalten. Inhaltlich ergeben sich zwei Schnittstellen, eine zwischen der Präsentations- und Geschäftslogikschicht und eine zweite zwischen der Geschäftslogik- und Datenzugriffsschicht.

Dem Zugriff von der Präsentations- auf die Geschäftslogikschicht kommt dabei eine besondere Bedeutung zu, weil hier einerseits die geringste Kommunikationsgeschwindigkeit erwartet wird, andererseits vor allem bei komplexen GUI-Interaktionen eine intensive Kommunikation stattfindet. An der Schnittstelle zwischen Geschäftslogik- und Datenzugriffsschicht wird vergleichsweise selten kommuniziert. Dies passiert immer, wenn Daten gelesen oder geschrieben werden müssen. Die Schnittstelle ist weniger von der fachlichen Logik bestimmt als von den objektrelationalen Mappingalgorithmen.

© Copyright Software & Support Verlag

Die Trennung von Geschäftslogik- und Datenzugriffsschicht ist immer dann sinnvoll, wenn die Datenzugriffsschicht durch Synergien (Caching) zwischen möglichst vielen UseCases massiven Nutzen erlangt und die Applikations-Server aus Lastverteilungsgründen skaliert werden müssen.

Die Teilung von Präsentations- von Geschäftslogikschicht ist hingegen für GUI-Anwendungen sehr wichtig und für Webanwendungen häufig sinnvoll. Die Trennung wird in der Regel aufgrund der Betreibbarkeit (Security, Deployment, Skalierung, Lastverteilung) vorgenommen und hat erhebliche Auswirkung auf die Performance einer Anwendung. Richtig angewendet, kann durch Softwareverteilung die Performance gesteigert werden. Falsch angewendet, kann es zu dramatischen Einbußen gegenüber Rich-Client-Anwendungen kommen. An dieser Trennschicht liegt also der Schlüssel für den Projekterfolg einer J2EE-Anwendung.

Schnittstelle Präsentation und Geschäftslogik

Betrachtet man nun die Schnittstele genauer, so muss man gegenläufige Ziele feststellen. Die Präsentationsschicht soll möglichst keine fachliche Logik beinhalten. Gleichzeitig wachsen die Anforderungen an den Oberflächenkomfort gegenüber „alten“ Rich-Client-Anwendungen eher, als dass diese schrumpfen. Mindestens ist jedoch ein vergleichbarer Oberflächenkomfort notwendig. Dies bedeutet z.B., dass beim Verlassen von Feldern entsprechende Prüfungen erfolgen, dass nicht ausführbare Aktionen deaktiviert werden oder im Bearbeitungsablauf sinnvolle, zustandsabhängige Voreinstellungen gemacht werden. All dieser Komfort kann sich nur über eine intensive Kommunikation mit der Geschäftslogikschicht einstellen, um die notwendige Aufgabenteilung zu erhalten. Da sich die Geschäftslogikschicht aber nicht, wie bei Rich-Client-Architekturen, im selben Prozess befindet wie die Präsentationsschicht, sondern auf einem Applikations-Server, ergibt sich auch in schnellen Netzwerken ein Kommunikationsproblem. Es muss also zu häufiger Netzverkehr unterbunden und auch das Übertragungsvolumen beachtet werden. Dies kann nur erfolgreich durch eine Bündelung von Aktionen erfolgen, was zwangsläufig etwas Logik vom Serverprozess in den Clientprozess verschiebt. Hier ergibt sich im folgenden das Ziel, diesen Sachverhalt möglichst nicht mit der fachlichen Logik zu vermischen und dem Anwendungsentwickler ein möglichst transparentes Arbeiten zu ermöglichen.

Die Implementierung serverseitiger Geschäftslogik wird sinnvollerweise über die Verwendung eines zustandsbehafteten Servers vorgenommen. Dabei ist ein Vorhalten von Zuständen sowohl auf dem Client als auch auf dem Server notwendig. Der Client muss die Geschäftsobjekte enthalten, die in der Präsentationsschicht angezeigt werden und mit denen interagiert wird. Diese können nicht ausschließlich auf dem Server liegen, weil sonst die Kommunikation zu intensiv wäre. Der Server enthält alle Geschäftsobjekte, die für die Ausführung der Geschäftslogik notwendig sind. Dies sind in der Regel mehr, als im Client präsentiert werden. (Hieraus ergibt sich ein massiver Performancevorteil, weil Objekte, die nicht präsentiert werden, auf dem Server verbleiben können). Um ein derartiges Konzept zu implementieren, sind folgende Probleme zu lösen:

- Wie werden UseCases und Instanzen fachlicher Klassen identifiziert?
- Wie werden Methoden an UseCases oder fachliche Klassen delegiert?
- Wie wird dem Server mitgeteilt, welche Zustandsänderungen der Client vorgenommen hat?
- Wie wird dem Client mitgeteilt, welche Zustandsänderungen der Server vorgenommen hat?

Das cJEF-Framework implementiert das zustandsbehaftete Verteilungskonzept mit Hilfe von Statefut-Session-Beans, wobei UseCases und fachliche Klassen zunächst als EJB-unabhängige Klassen implementiert werden. Mit ihnen kann über eine EJB-Fassade kommuniziert werden. Hier kommt das J2EE-Pattern Session Facade zur Anbindung von UseCases zum Einsatz [3].

UseCases und fachliche Klassen können wegen ihrer EJB-Unabhängigkeit sowohl auf dem Applikations-Server, als

Fehler beim Einsatz der J2EE-Architektur

Die J2EE-Konzepte bilden eine sehr gute Grundlage für die Entwicklung serverseitiger Geschäftslogik. In der Praxis findet man jedoch häufig Probleme bei der Anwendung der EJBs, weil die Möglichkeiten und Grenzen nicht ausreichend beachtet werden. Grundlage von Überlegungen bildet meistens die Abbildung von Anwendungsfällen und Geschäftsobjekten (fachliche, persistente Modellklassen).

Entity Beans wird häufig die Möglichkeit zur Abbildung von Geschäftsobjekten zugedacht. Sie sind jedoch nicht vorrangig für diesen Zweck konzipiert worden. Entity Beans haben keinen „Bearbeitungszustand“, jegliche Datenänderung ist sofort persistent, bzw. befindet sich innerhalb einer geöffneten Transaktion. Es handelt sich hier also eher um eine Abbildung von Daten bzw. Datenbanktabellen und ist aus unserer Sicht konzeptionell in der Datenzugriffsschicht anzusiedeln.

Session Beans sind zum Vorhalten eines Bearbeitungszustand geeignet, wenn sie als *stateful* verwendet werden. Sie sind selbst nicht persistent. Session Beans können Geschäftsobjekte schlecht abbilden, da Geschäftsobjekte üblicherweise. Beziehungen zu anderen Geschäftsobjekten haben und selbst persistent sind. Stateful Session Beans eignen sich jedoch gut als Abbildung eines Anwendungsfalls und liefern damit eine geeignete Infrastruktur zur Anwendungsverteilung.

Für die Abbildung von Geschäftsobjekten ist es aus unserer Sicht notwendig, ein eigenes Konzept zu entwickeln. Hier kann man zwar auf J2EE-Patterns wie Data Access Object (DAO) [4] oder den JDO-Standard [5] zurückgreifen, jedoch steht bei beiden Konzepten vorrangig die Persistenz im Focus und weniger die Abbildung von Geschäftslogik.

© Copyright Software & Support Verlag

auch im Client instanziert werden. Die Client-Instanzen fungieren dann jeweils wie ein intelligentes Proxy, bei denen Methoden sowohl im Client als auch im Server ausgeführt werden können. Um Instanzen zu identifizieren, verwendet das Framework ein eigenes Identitätskonzept. Dadurch können lokal ausgeführte Methoden bei Bedarf an die richtige Instanz innerhalb des Servers delegiert werden. Es ist also möglich, eine komplexe Verarbeitung auf dem Server und die präsentationsnahe Logik z.B. eines Daten-Pflegeprozesses im Client ablaufen zu lassen. Das Framework sorgt hier für Transparenz, indem es die Veränderungen, die auf beiden Seiten entstehen können, immer korrekt miteinander synchronisiert. Zur Synchronisation werden gebündelte Modell-Änderungsnachrichten verwendet. Die Daten werden mithilfe von Datenobjekten serialisiert übertragen (Abb.3).

Ein Beispiel aus einem Auftragserfassungssystem könnte folgendermaßen aussehen. Im Dialog werden Auftrags- und Positionsdaten geändert (Lieferbedingung, Menge etc). Dies passiert im Client, wobei notwendige Prüfungen sofort vorgenommen werden. Dann erfolgt die Bestätigung der Positionserfassung, und auf dem Server erfolgt die Preisrechnung (Verwendung eines Konditionssystems) und die Bestandsreservierung (Verwendung eines Reservierungssystems). Dabei ändern sich wieder Positonsdaten (reservierte Menge, voraussichtlicher Liefertermin), diese werden wieder in den Client synchronisiert und dort dem Benutzer präsentiert.

Schnittstelle Geschäftslogik und Datenzugriff

Bei der Verteilung der Datenzugriffsschicht ist ebenfalls ein Synchronisationsmechanismus erforderlich, obwohl man dies auf den ersten Blick nicht vermuten würde. Bei lesenden Zugriffen werden Daten z.B. aus einem Datenbanksystem in die Geschäftslogikschicht transferiert, um daraus fachliche Instanzen zu bilden. Auf diesen Instanzen wird die Geschäftslogik ausgeführt, was als Ergebnis immer die Veränderung, Erzeugung oder das Löschen von Instanzen bedeutet. Diese Veränderung findet zunächst für einen UseCase im Hauptspeicher statt. Erst beim Abschluss des UseCase (i. d. R. ausgelöst durch ein Speicher-Kommando an der Oberfläche) werden die Änderungen in die Datenbank zurückgeschrieben. Man kann hier die nahe Verwandtschaft zum oben beschriebenen Synchronisationsmechanismus erkennen:

- Neue oder geänderte Instanzen werden bei Bedarf in die Datenzugriffsschicht transferiert.
- Änderungen werden gesammelt, bis eine bestimmte Aktion das Zurückschreiben aller vorgenommenen Änderungen veranlasst. Hier wird nur die Differenz, d. h. die wirkliche Veränderung kommuniziert.

Das Besondere an diesem Synchronisationsmechanismus ist, dass er sehr viel seltener stattfindet als der zwischen der Präsentations- und Geschäftslogikschicht. Die Implementierung der Datenzugriffsschicht führt dann bei der Verwendung von Datenbanken zum Lese- oder Schreibzugriff auf das Datenbanksystem.

In Bezug auf die Notwendigkeit eines zustandsbehafteten Servers ergibt sich hier ein anderes Bild als bei der Geschäftslogikschicht. Während der Ausführung eines UseCases wird in der Datenzugriffsschicht keine Logik ausgeführt, die auf dem aktuellen Zustand operieren muss. Es ist daher an dieser Stelle auch eine zustandslose Verarbeitung denkbar. Sinnvoll ist ein Zustand dann, wenn in der Implementierung einer Datenzugriffsoperation auf die ursprünglich gelesenen Daten zugegriffen werden muss. Ein erneutes Lesen aus der Datenbank hätte hier Performancenachteile und würde ggf. einen von anderen Benutzern bereits geänderten Satz liefern.

Synchronisation der Schichten im Detail

Ein Kernziel des cJEF-Frameworks ist die Entlastung des Anwendungsentwicklers

Zustandslose oder zustandsbehaftete Server

Bei der Lösung der geschilderten Probleme stellt sich nun die Frage, ob die Geschäftslogikschicht zustandslos oder zustandsbehaftet arbeiten sollte. Auch hier ergibt sich ein Zielkonflikt. Einerseits ist es aus Sicht einer Serverimplementierung günstig, zustandslos zu arbeiten, weil hier sowohl die Skalierbarkeit als auch die Ausfallsicherheit gesteigert werden kann. Andererseits muss man sich fragen, wie eine zustandslose UseCase-Implementierung sinnvoll die Geschäftslogik übernehmen kann. Hier gibt es zwei mögliche Lösungen:

- Implementierung der Geschäftslogik ausschließlich mit funktionalen Schnittstellen, die alle Daten, auf denen sie operieren, als Parameter vom (zustandsbehafteten) Client bekommt.
- Objektorientierte Implementierung der Geschäftslogik mit einer vorgeschalteten Komplettübertragung des UseCase-Zustands bei jedem Methodenaufruf.

Beide Lösungen scheinen hier ungeeignet. Die Anwendung funktionaler Schnittstellen wäre ein aus unserer Sicht nicht akzeptabler Weg, weil hier das Prinzip der Objektorientierung aufgegeben würde. Die Komplettübertragung des Zustands hingegen würde massive Nachteile bei der Performance bedeuten, weil einerseits Bandbreiten der Netzübertragung begrenzt sind und andererseits die intensive, mehrfach notwendige Erzeugung derselben fachlichen Instanzen massive Last auf den Serverprozess bringen würde.

Die Alternative kann hier nur ein zustandsbehafteter Server sein. Hier kommt uns in der Java-Welt das Konzept der Stateful Session Beans entgegen, weil diese den Zustand vorhalten können und dank vorhandenem Passivierungskonzept auch in hoch belasteten Serverumgebungen zum Einsatz kommen können. Ein kleiner Blick auf andere Technologien zeigt, wie ausgereift das Java-Konzept ist: Im Microsoft COM+-Server (ehemals MTS-Server) können zwar auch Zustände verwendet werden, es existiert jedoch standardmäßig kein sessionbezogenes Passivierungskonzept. Dies führt zur weit verbreiteten Empfehlung, COM+ nur zustandslos zu verwenden. Folgt man dieser Empfehlung, ist es unter den oben genannten Bedingungen nicht empfehlenswert, im COM+-Server Geschäftslogik zu implementieren.

© Copyright Software & Support Verlag

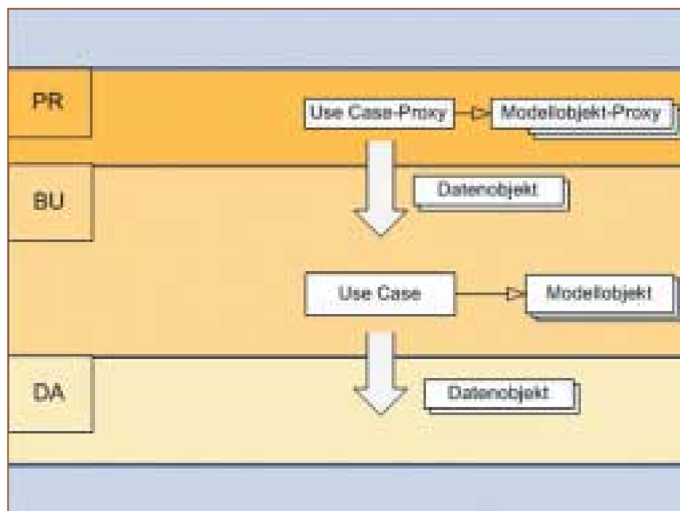

Abb.3: Synchronisation der Schichten

von den technischen Aufgaben, die für die Verteilung notwendig sind. Nur dadurch kann er sich in der logischen Schichtung bewegen und mischt Fachlichkeit nicht mit technischer Komplexität. Der Entwickler bestimmt jedoch mit einfachen Mitteln, welche Methoden lokal oder auf den Applikations-Server ausgeführt werden sollen. Ist kein Applikations-Server vorhanden, weil eine andere Architektur-Installationen zum Einsatz kommt (oder der Entwickler die Anwendung im Entwicklungsmodus nicht verteilt betreibt), so wird ohne Codeänderung der Anwendungscode lokal ausgeführt.

Bei der Verwendung eines Servers sorgt das Framework dafür, dass die logische Aufteilung in UseCase-Manager, UseCase und Modellinstanzen auf dem Server und auf dem Client physisch instanziert werden. Dabei wird berücksichtigt, dass nur solche Instanzen in den Client gelangen, die da auch wirklich verwendet werden (Abb. 4).

Im Anwendungsablauf kommt man nun zu dem Punkt, wo fachliche Methoden von einer Oberflächenimplementierung aufgerufen werden. Die Proxy-Modellobjekte, die sich im Client-Prozess befinden, werden beim Aufruf verwendet. Diese sollen die Methoden an das kooperierende Server-Objekt delegieren. Um dies ohne viel Programmieraufwand transparent zu erreichen, wird für jedes Modell und für jeden UseCase ein Interface eingeführt, welches die fachlichen Methoden beinhaltet, die auf dem Server ausgeführt werden können. Technische Methoden (z.B. das Speichern eines Anwendungsfalls oder das Laden von Modellobjekten) sind bereits im Framework vorhanden. Das Framework übernimmt die korrekte Delegation aller fachlichen Methoden und sorgt für eine korrekte Synchronisation des Anwendungszustands (z.B. muss in einem Auftragserfassungssystem die Bestandsreservierung und Preisrechnung immer auf den aktuell in der Oberfläche eingegebenen Positionsmengen arbeiten).

Als Beispiel verwenden wir die fachliche Klasse *Auftrag*, die eine Methode *preisrechnung* besitzt, die auf dem Applikationsserver ausgeführt werden soll. Die Methode wird in dem Interface *AuftragBU* deklariert und zunächst von der Modellklasse selbst implementiert. Diese Implementierung wird verwendet, wenn die Auftragsinstanz keinen Serverpart besitzt. Dies könnte z.B. ein Rich-Client oder Webserver sein.

Wird hingegen ein Serverpart verwendet, so existiert eine zweite Instanz der Klasse *Auftrag*, an die die Ausführung der Methode delegiert wird. Die Delegation wird durch die Bildung eines dynamischen Proxies automatisiert (*java.lang.reflect.Proxy.newProxyInstance*(...)). Die Ausführung der Methode wird dabei an eine Instanz von *java.lang.reflect.InvocationHandler* delegiert (Abb. 5).

Für die Delegation von Methoden kommt uns die Modellierung der Benutzertransaktion als *UseCaseManager* nun sehr entgegen, weil wir uns somit immer in

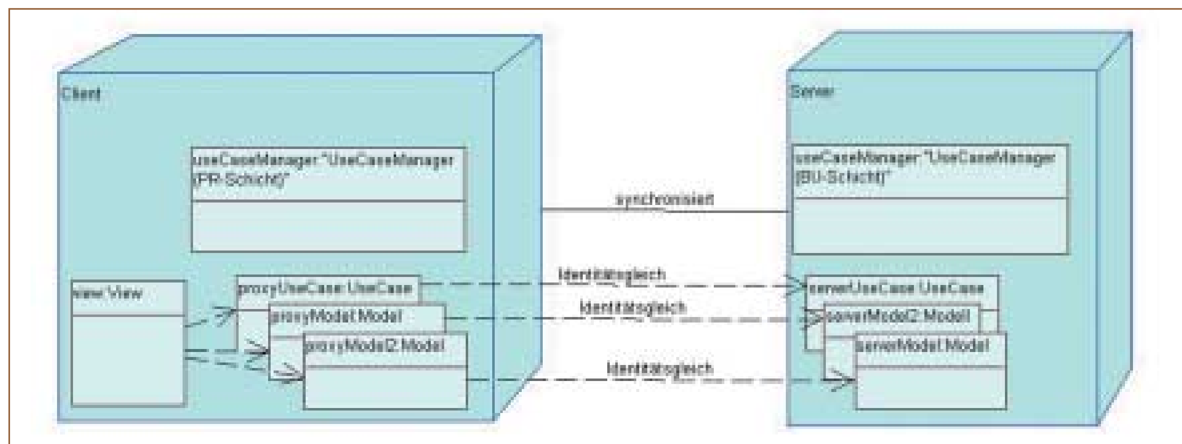

Abb.4: Verteilung auf Client- und Serverprozess

© Copyright Software & Support Verlag

Abb.5: Klassenstruktur eines verteilten Systems

einem klar abgegrenzten Scope befinden, innerhalb dessen Modell- und UseCase-Instanzen in Beziehung gebracht werden müssen.

Bei der Delegation gibt es zwei Varianten. Die eine verwendet EJB Session Beans, um die Methodenausführung an einen J2EE-Applikations-Server zu delegieren. Dies ist die Produktivumgebung bei der Verwendung von Thin-Clients. Die Zweite Variante dient dem Test einer lokal entwickelten Anwendung in Bezug auf ihre Verteilbarkeit. Hier wird die Verwendung eines Applikations-Servers in Bezug auf häufig auftretende Probleme in einer einfach zu kontrollierenden und debugbaren Umgebung simuliert. Dabei können die meisten Fehlerursachen wie z.B. die Verwendung nicht serialisierbarer Objekte als Parameter von Methoden leicht gefunden und behoben werden. Für andere technische Serverumgebungen können hier nun leicht weitere Ausprägungen realisiert werden. Denkbar ist eine direkte TCP/IP, RMI, HTTP oder SOAP- bzw. Web Service-Schnittstelle. In alle Fällen bliebe der Anwendungscode unverändert.

Entwicklungsschritte einer verteilten Anwendung

Die Entwicklung einer verteilten Anwendung wird stufenweise vorgenommen. Zuerst wird in einer einfachen, nicht verteilten Umgebung entwickelt. Hier werden alle fachlichen Probleme gelöst und die Anwendung komplett, inkl. der Oberfläche, getestet. Diese Phase nimmt üblicherweise die meiste Entwicklungszeit in Anspruch. In einem zweiten Schritt wird die Verteilung geplant, das Modell-Interface mit den Verteilungsmethoden ausgestattet und die Methodenaufrufe implementiert (hier wird anstatt des direkten Aufrufs der Modellmethode das Modellproxy angesprochen und die Methode *bu().preisrechnung* aufgerufen). Der nächste Schritt ist der Verteilungstest, der durch Konfiguration der Anwendung aktiviert wird. Erst wenn dieser Test erfolgreich verlaufen ist, passiert die Verteilung auf einen Applikationsserver. Die notwendigen EJBs liegen bereits im Framework vor, sodass hier kein zusätzlicher Code entsteht. Nach dem Deployment wird die EJB-Verwendung wiederum durch einfache Anwendungskonfiguration aktiviert.

Bei der Verteilung der Datenzugriffsschicht wird ein ähnliches Konzept verfolgt. Eine Verteilung findet jedoch meistens nicht statt, wenn bereits die Geschäftslogik auf einem Server liegt. Aus diesem Grunde wird dieser Verteilungsaspekt hier nicht näher betrachtet.

Fazit

Die Risiken einer J2EE-basierten Entwicklung können durch eine geeignete Anwendungsarchitektur vermieden werden. Auch das ausgereifte J2EE-Framework bietet großen Raum für den Einsatz weiterer Frameworks, die die Entwicklungsproduktivität erheblich steigern können. Das c.a.r.u.s. Java Enterprise Framework liefert hier als Beispiel eine ausgereifte Implementierung adäquater Konzepte. Unserer Erfahrungen nach lassen sich darüber hinaus durch den Einsatz von Frameworks die Geschwindigkeit und die Qualität der Softwareentwicklung und damit verbunden auch die Motivation der Entwickler erheblich steigern.

Béla Bodnár ist verantwortlich für die Softwaretechnik bei der c.a.r.u.s. Information Technology AG. Ein Schwerpunkt ist die Architektur komplexer Anwendungssysteme und die Entwicklung der c.a.r.u.s. Frameworks.

Links & Literatur

[1] c.a.r.u.s. Java Enterprise Framework, www.cjef.de/, www.carus-it.com/

[2] Design Patterns, Erich Gamma, R. Helm, R. Johnson, J. Vlissides, Addison-Wesley, 1997

[3] J2EE-Pattern SessionFacade, java.sun.com/blueprints/corej2eepatterns/Patterns/SessionFacade.html

[4] J2EE-Pattern Data Access Object, java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html

[5] Java Data Objects, java.sun.com/products/jdo/

© Copyright Software & Support Verlag