# J2EE im kommerziellen Großprojekt

*Markus Völter, voelter@acm.org, www.voelter.de*
*Nicolai Josutti, Thomas Stahl*

**J2EE bietet eine Standard-Architektur für große kommerzielle Systeme. Doch wie in jedem Projekt muss man auch bei J2EE-Projekten bei der Umsetzung die besonderen Anforderungen und Randbedingungen berücksichtigen. Wir berichten hier von einem erfolgreichen Grobprojekt im Bankenumfeld, bei dem J2EE etwas anders als sonst eingesetzt wurde. Diese Besonderheiten betrachten wir als Folge der Projektgröße und der Forderung nach einer einfachen, handhabbaren, und skalierbaren Architektur. Insofern stellen wir die im folgenden vorgestellte Architektur als eine typische, praktische Umsetzung des J2EE-Ansatzes zur Diskussion.**

## Hintergrund

Bei der Umstellung der gesamten Softwarelandschaft für die Schaltersysteme eines Sparkassenverbundes auf eine neue technologische Basis zu stellen. Dabei gilt es einige interessante Randbedingungen zu berücksichtigen:

- Die Backendstruktur, bestehend aus Großrechnern, soll weiterhin bestehen bleiben und in das neue System integriert werden. Dabei bleibt deren Rolle als juristisches Bestandssystem erhalten.
- Die Bedienerarbeitsplätze sind rein Browser-basiert, also über ein Intranet angebunden. Um, wie bei Intranet-Lösungen üblich, keine Probleme mit dem Datenverkehr zu bekommen, sollen diese nur mittels HTTP angebunden werden.
- Zur Anbindung ihrer kleinen Außenstellen stellen manche Sparkassen aus Kostengründen nur 64k-Leitungen zur Verfügung. Das bedeutet, dass auf eine äußerst geringe Bandbreite zu achten ist.
- Die Anwendung muss bis zu 50.000 Client Sessions gleichzeitig behandeln können.

Als Basistechnologie wurde, wie bereits erwähnt, J2EE gewählt. Abbildung 1 zeigt das System und dessen Aufteilung auf die verschiedenen Tiers im Überblick – bis hierher eine klassische 3-Tier Architektur.

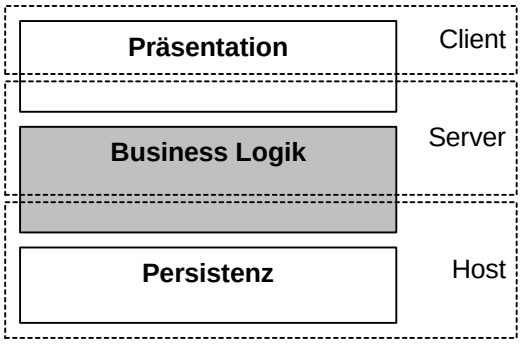


*Abbildung 1: Architektur im Überblick*

Die Persistenz wird komplett vom Host erledigt. Die Business-Logik enthält Anteile auf dem Host, sowie auch Anteile auf Server-Seite: Auf dem Host werden die Transaktionen für die einzelnen bankfachlichen Vorgänge bereitgestellt (Konto anlegen, Überweisung tätigen, ...). Auf dem Server werden die einzelnen Transaktionen zu Geschäftsprozessen zusammengestellt. Die Präsentation wird, wie bei Weboberflächen üblich, zum Teil auf dem Server und zum Teil auf dem Client-Browser abgehandelt.

Bei der Realisierung dieses Ansatzes gibt es verschiedene interessante Lösungen: Die Modellierung der Mittelschicht (also der Business-Logik), Details zur Infrastruktur (Hardware), Details zur Client-Anbindung (Bandbreitenreduziereung, etc.). Dieser Artikel betrachtet die Mittelschicht. Weitere Artikel sind in Planung.

## Die Mittelschicht

Die Realisierung der Business-Logik Schicht ist aus vielerlei Gesichtspunkten heraus interessant. Sie ist auf extreme Skalierbarkeit hin ausgelegt und will sich dabei möglichst wenig auf die Lastverteilungs- und Failover-Funktionalitäten des eingesetzten Applikationsservers verlassen.

Die Business-Logik Schicht besteht intern wiederum aus drei Schichten, der Prozessschicht, der Komponentenschicht und der Legacy-Schicht:

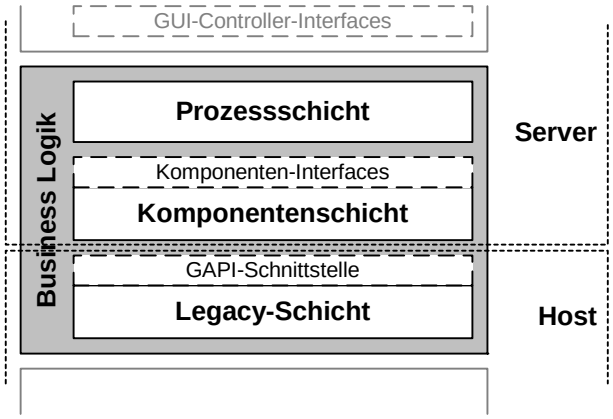


*Abbildung 2: Interne Struktur der Business-Logik Schicht*

- Die Legacy-Schicht stellt die Zugrifflogik auf das Mainframe-Backend dar. Die dafür verwendete sogenannte GAPI-Schnittstelle ist eine Java Schnittstelle, die einen einfachen und gekapselten Zugriff auf die Mainframe-Funktionalität bietet. Jeder hier angebotene Funktionsaufruf stellt eine in sich abgeschlossene Transaktion auf dem Backendsystem dar. Eine Operation wie „Überweisung tätigen" kann also nur als Ganzes und nicht in Teilen („Betrag von einem Konto abheben", „Betrag auf das andere Konto einzahlen") durchgeführt werden. Dies hat den Vorteil, dass die Transaktionen in sich abgeschlossen sind und aus Sicht des Bestandssystems in der Mittelschicht kein Transaktionskontext benötigt wird.
- Die Komponentenschicht bietet höher abstrakte Dienste an und nutzt dazu zum Teil die Legacy Schicht mittels der GAPI Schnittstelle. Sie besteht ihrerseits ebenfalls aus mehreren Schichten, um die Abhängigkeiten zwischen den einzelnen Entwicklungsteams zu reduzieren und insbesondere zyklische Abhängigkeiten zu vermeiden. Um diese Ziele auch wirklich zu erreichen ist für jede dieser Schichten genau definiert, auf welche anderen Schichten zugegriffen werden darf. Es existieren die folgenden Schichten:
    - eine Schicht mit eher technischen Basisfunktionen wie „anmelden" oder „drucken",
    - eine Schicht mit fachlichen Service-Funktionen wie „Bankleitzahl suchen" oder „Name zu einer Kundennummer liefern"
    - eine Schicht mit den eigentlichen Einzelschritten der Geschäftsprozesse („Kunde identifizieren", „Sparkonto anlegen").
- Die Prozessschicht steuert den Ablauf der jeweiligen Anwendungen und deren aktiven Geschäftsprozesse und speichert den jeweiligen Zustand der Prozesse. Sie

greift auf die Komponentenschicht zu, um Daten oder Dienste zu erhalten, und kontrolliert die GUI-Seite mittels der Controller Interfaces.

Die Prozessschicht ist also sozusagen der „Master", das Programmfluss-steuernde Element in der gesamten Architektur.

### Prozesse und Aktivitäten

Aus Sicht der Benutzer bestehen alle Anwendungen aus sogenannten *Prozessen*. Ein angemeldeter Benutzer kann gleichzeitig mehrere Prozesse bearbeiten, wobei zu einer Zeit immer nur einer aktiv sein kann. Prozesse werden mit Hilfe von UML-Aktivitätsdiagrammen in Rational Rose (oder direkt in XML) definiert. Ein Prozess besteht aus einer Folge von *Aktivitäten*, die über entsprechende Transitionen verbunden sind. Aktivitäten enthalten die eigentliche Programmlogik und greifen zum einen auf die Komponenten zu, und zum anderen kontrollieren sie das GUI mit Hilfe von entsprechenden Controller-Interfaces.

Ein wichtiges Konzept des Prozessmodells ist der sogenannte Prozesskontext. Hier werden *alle* Daten des Prozesses hinterleg. Er dient damit auch zur Datenübergabe zwischen den einzelnen Aktivitäten eines Geschäftsprozesses. Man beachte, dass dies die einzige Schnittstelle zwischen den einzelnen Aktivitäten ist. Es gibt *keinen* globalen Datenbestand (so etwas wie ein Business-Objekt-Modell). Dies hat einen essentiellen Vorteil: Es ist keine zentrale Modellierung aller Daten erforderlich. Es reicht aus, wenn zwei fachliche Teamsdie Daten im Rahmen eines Geschäftsprozesses austauschen, die Struktur dieser Daten bilateral klären. Die im Prozesskontext abgelegten Daten werden als Entitäten bezeichnet und unterliegen nur zwei Bedingungen: Sie müssen serialisierbar sein und gehen immer in den Besitz des Prozesskontextes über. Ob es sich bei Entitäten um einfache Variablen (Integer, String), einfache Datenstrukturenoder ganze Objektgeflechte handelt, bleibt den einzelnen fachlichen Teams überlassen.

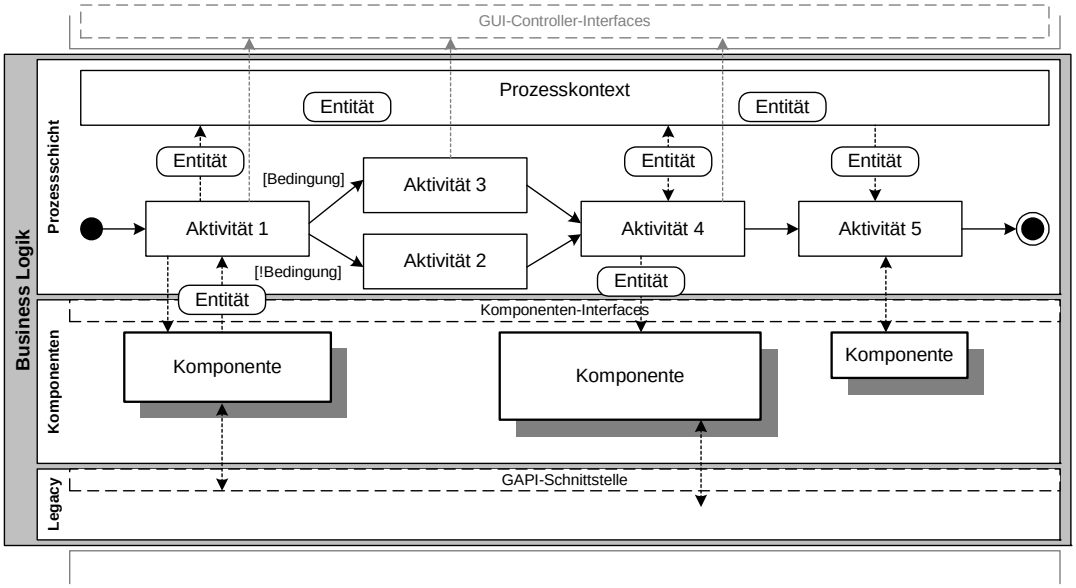


*Abbildung 3: Das Prozessmodell und Komponenten*

Im folgenden nun einige Details, wie diese Konzepte implementiert sind, und warum dies so gemacht wurde.

### Komponenten

Komponenten sind reine Diensterbringer. Sie sind daher zustandslos. Dies bedeutet, dass eine bestimmte Funktionalität innerhalb eines Methodenaufrufes erbracht werden muss – beim nächsten Aufruf weiß die Komponenten nichts mehr vom vorhergehenden. Dies hat verschiedene, weitreichende Konsequenzen. Alle Komponenteninstanzen können z.B. gepoolt werden. Da alle Instanzen einer Komponente per Definition identisch sind, kann ein Methodenaufruf von einer beliebigen Instanz gehandelt werden. Aus der Zustandslosigkeit der Komponenten ergibt sich weiterhin, dass Komponenten nie an einem bestimmten Prozess gebunden sind: alle Prozesse können auf den gemeinsamen Instanzpool zugreifen. Das reduziert die pro Prozess zu allockierenden Ressourcen und sorgt damit für Skalierbarkeit.

Auf der anderen Seite steht natürlich der Nachteil, dass die Zustandslosigkeit ein etwas anderes Programmiermodell verlangt. Insbesondere müssen also alle prozessspezifischen Daten sowie Zustand, der zwischen verschiedenen Aufrufen einer (oder mehrerer) Komponenten benötigt wird, innerhalb von Aktivitäten oder dem Prozesskontext gehalten werden. Dieser Nachteil hat sich allerdings als unkritisch erwiesen. Im Gegenteil: die klare Rollenverteilung hat dazu geführt, dass Geschäftsprozesse und Komponenten ohne große Verhandlung, wer welchen Zustand wo verwaltet, sehr schnell implementiert werden.

### Entitäten

Entitäten sind nicht nur die „Datenpakete“, die zur Kommunikation zwischen Aktivitäten verwendet werden. Jede Art von Daten, die von Komponenten geliefert werden, sind Entitäten. Auch wenn es in Bezug auf Isolation, Lebensdauer und Wiederauffindbarkeit verschiedene Arten von Entitäten geben kann (wir haben vier verschiedene Arten unterschieden), gibt es doch zahlreiche Gemeinsamkeiten, die die Semantik von Entitäten grundsätzlich klar regeln. So hat eine Entität üblicherweise eine Reihe von Attributen und einfaches Verhalten (Getter, Setter, einfache Validierer). Komplexere Funktionalität ist in Komponenten untergebracht, da dazu üblicherweise Ressourcen benötigt werden (Datenbankverbindungen, Verbindung zur Legacy Anwendung) die Entitäten nicht besitzen dürfen. Es handelt sich also weniger um klassische Objekte im Sinne von „autonomen Handlungsträgern“ als vielmehr um mehr oder weniger komplexe Datenstrukturen mit dazugehörigen Operationen.

Entitäten liegen immer in der Verantwortung der Aktivität oder Komponente, die sie angefordert hat. Sobald eine Komponente Daten in Form von Entitäten herausgibt, ist sie nicht mehr für diese Daten zuständig. Dies gilt insbesondere auch für deren Aktualität. Es ist Aufgabe der „Kunden“, dafür zu sorgen, dass die vom „Lieferanten“ gelieferten Entitäten gegebenenfalls (durch Neuanforderung bei Komponenten) aktualisiert werden.

Entitäten sind eigenständig. Eine Entität darf keine direkte, programmiersprachline Referenzen auf Ressourcen oder andere Entitäten enthalten. Wenn ein Bezug zu einer anderen Entität nötig ist (z.B. im Rahmen der Speicherung im Prozesskontext) so muss dies über einen Fremdschlüssel geschehen (was damit einer logische Referenz auf die Daten entspricht, aber keiner direkten Referenz auf eine andere Entität!)

### Isolation zwischen verschiedenen Prozessen

Entitäten sind immer Eigentum genau eines Prozesses. Sie sind komplett isoliert von den Entitäten anderer Prozesse. Dies bedeutet auch dass es vorkommen kann, dass ein Datensatz auf dem Host in mehreren Prozessen in mehreren Entitäten vorhanden ist. Die Daten sind auf Serverseite komplett unabhängig und wissen nichts voneinander.

Dies bedeutet insbesondere, dass verschiedene Instanzen (desselben oder verschiedener) Geschäftsprozesse keine gemeinsamen Daten verwenden können. Wird also in einem Geschäftsprozess die Adresse eines Kunden geändert während zeitgleich in einem anderen Geschäftsprozess ein Konto für ihn angelegt wird, wird dieses Konto mit den „alten“ Daten angelegt. Dieses Verhalten mag zunächst unschön erscheinen; doch muss man die Alternative betrachten: Hätten wir die Daten der Geschäftsprozesse gemeinsam gehalten, müssten wir nicht nur dafür sorgen, dass Änderungen anderen Geschäftsprozessen mitgeteilt werden, sondern wir hätten auch mit den daraus folgenden Inkonsistenzen umgehen müssen. Alternativ hätten wir Daten pessimistisch sperren müssen. Das hier verwendete Prinzip bedeutet optimistisches Locking. Sofern es wichtig ist, die Gültigkeit

von Daten vor schreibenden Transaktionen erneut sicherzustellen, muss dies fachlich ausprogrammiert werden.

Im Prinzip bedeutet diese Entscheidung, dass wir das Backend-System weiterhin als juristisches Bestandssystem definieren und kein neues System implementieren, dass die Abhängigkeiten von Daten unterschiedlicher Geschäftsprozesse ausmodelliert. Oder anders formuliert verhalten wir uns auch nicht anders als der Bediener, der sich bisher die Daten eines Kunden gemerkt oder aufgeschrieben hat bevor er den nächsten Schritt getätigt hat. Auch in diesem Fall konnten bisher verschiedenen Bediener den gleichen Datenbestand modifizieren.

Ein weitere Vorteil dieser Entscheidung besteht darin, dass die einzelnen Geschäftsprozesse auf verschiedene Server verteilt werden können, ohne dass zwischen diesen Servern kommuniziert werden muss. Dies Erleichtert die Skalierbarkeit des Systems essentiell.

## Konsequenzen dieses Komponentenmodells

Nachdem nun die grundlegenden Konzepte eingeführt wurden, wird im folgenden auf verschiedene Konsequenzen eingegangen. Dabei wird auch beschrieben, wie diese Dinge technologisch im Rahmen von J2EE umgesetzt wurden.

### Unterbrechbarkeit

Unterbrechbarkeit ist ein zentrales Konzept des gesamten Systems. Wie oben bereits erläutert, kann ein Benutzer zu einer Zeit mehrere Prozesse bearbeiten, allerdings kann immer nur einer aktiv sein. Die anderen sind passiv, oder *unterbrochen*. Der Zustand eines unterbrochenen Prozesses ist persistent in einer Datenbank gespeichert. Dies ist sehr einfach möglich: Der gesamte Zustand des Prozesses ist in dessen Prozesskontext abgelegt. Darin befinden sich ausschließlich serialisierbare Entitäten.

Der Zustand eines Prozesses kann also durch Speicherung seines Kontextes in einer Datenbank persistiert werden. Konkret werden in der Datenbank die folgenden Informationen abgelegt:

- Die ID des Prozesses
- Der Benutzer, dem dieser Prozess gehört
- Die ID der Aktivität, mit der der Prozess fortgesetzt werden muss
- Alle Entitäten als ein großes serialisiertes BLOB

Prinzipiell könnte der Prozess nach Abschluss jeder Aktivität unterbrochen werden – er müsste dann aber auch nach jeder Aktivität gespeichert werden. Da auch diese Speicherung Zeit benötigt, wird der Prozess nur an bestimmten, vom Prozessdesigner angegebenen „Savepoints“ gespeichert.

Wenn der Benutzer sich wieder am System anmeldet, bekommt er alle seine unterbrochenen Prozesse angezeigt und er kann auswählen, welchen er fortsetzen will.

Ein Aufsetzen auf „alte" Daten kann kritisch sein: Wollte ein Kunde ein Konto anlegen und hatte erst einen Tag später seinen Personalausweis dabei, ist dies sicherlich unkritisch. Fehlte der Personalausweis beim Kauf von Aktien spielt der Zeitpunkt sicherlich eine entscheidende Rolle. Deshalb bietet die Architektur die Möglichkeit ein Wiederaufsetzen zu erkennen. So kann man ggf. besondere fachliche Reaktionen auf ein späteres Wiederaufsetzen implementieren.

### Failover

Es kann immer passieren, dass einzelne Systemkomponenten ausfallen, insbesondere eine Instanz des Applicationservers oder gleich eine ganze Maschine. In beiden Fällen wird eine Strategie benötigt, wie der Benutzer seine Arbeit fortsetzen kann.

Wie oben bereits erläutert, sind alle „Arbeitsdaten" des Benutzers in einem oder mehreren Prozesskontexten abgelegt. Per Definition kann nur einer aktiv sein, die andern sind unterbrochen und damit persistiert – bei Ausfall eines Servers sind diese nicht betroffen. Die Daten, die im Falle eines Server-Ausfalls also betroffen sind, sind die Daten des aktuellen Prozesses sowie die Session-Daten (also die Login-Informationen und Benutzereinstellungen).

Im Allgemeinen gibt es verschiedene Ausbaustufen von Failover: Im Idealfall merkt der Benutzer nichts von einem Serverausfall. Dies wird üblicherweise durch Replikation und Hot-Standby erzielt. Der Standby-Server hat immer alle Daten des aktiven Servers und fährt mit der Arbeit fort, wenn der aktive Server stirbt – wobei diese Bedingung nicht immer einfach festzustellen ist. So schön dies für den Benutzer ist, so hat es doch signifikante Performance-Einbussen auf dem Server zur Folge, da ständig Daten repliziert werden müssen. Man beachte dass dieses Konzept sowohl auf Hardwareebene möglich ist (Clustering) als auch auf Softwareebene (Clustering der Application-Server-Instanzen).

Obwohl der von uns verwendete Weblogic Server (WLS) die Möglichkeit des Clustering bietet, haben wir uns entschlossen, auf diesen Maximalkomfort zu Gunsten der Performanz im Nicht-Fehlerfall zu verzichten. Statt dessen setzen wir eine andere Möglichkeit des Failovers ein: Wenn die Session eines Benutzers stirbt, weil seine WLS-Instanz stirbt, so startet der Benutzer einfach eine neue Session. Er muss sich dazu mit Benutzername und Passwort neu anmelden – ein Vorgang, der vielleicht 10 Sekunden dauert. Er kann dann, wie nach einem normalen Login, einen seiner unterbrochenen Prozesse wiederaufnehmen.

Die Frage ist also nun, was mit dem beim Absturz aktiven Prozess geschieht. Dabei gibt es drei Möglichkeiten:

- Wenn es ein sehr einfacher Prozess war, muss der Benutzer diesen neu starten.

- Wenn der Prozess sowieso gerade an einem fachlich modellierten Savepoint war, so kann der Prozess dort natürlich fortgesetzt werden – die Daten sind ja sowieso gespeichert.
- Wenn es sich lohnt nach einem Absturz zwischen zwei fachlichen Savepoints wieder aufzusetzen, so kann man sogenannte technische Savepoints modellieren, an denen automatisch persistiert wird und an denen der Prozess nur nach einem Absturz fortgesetzt werden kann.

Diese Lösung funktioniert komplett mit Mitteln, die wir im System sowieso realisieren mussten, kostet kaum Performanz und ist zudem auch noch unabhängig von spezifischen Features irgendwelcher Applikationsserver.

### Lastverteilung

Die Lastverteilung im Rahmen des Systems ist auch recht einfach gelöst. Sie funktioniert grundsätzlich nur auf Session-Ebene. Wir gehen dabei davon aus, dass alle Bediener und damit alle Sessions im Schnitt gleich viel Last auf einen Server bringen. Daher wird die Last beim Anmelden verteilt – bei Anmeldung wird entschieden, auf welcher Applicationsserver-Instanz die Session laufen soll. Man beachte: Nach Serverausfall, wo sich die Bediener ja neu anmelden, funktioniert dies immer noch. Die neuen Sessions werden auf die verbliebenen Server verteilt.

### Transaktionen

Auf transaktionales Verhalten konnte in der Mittelschicht dieses Systems verzichtet werden. Dies liegt zum einen an der „fallabschließenden" Kapselung der Backendsysteme. Hinzu kommt, dass es derzeit keine technische Möglichkeit einer Backendanbindung mit Transaktionssicherheit gab (war mit der vorhandenen GAPI-Schnittstelle nicht möglich). Als Konsequenz kann es passieren, dass man im Rahmen eines Geschäftsprozesses zwar ein Kundenkonto anlegen, aber den gewünschten Fonds nicht kaufen konnte. In dem Fall bleibt das Konto trotzdem angelegt. Man kann ggf. eine Transaktion aufrufen, die das angelegte Konto wieder löscht, doch ist dies aus fachlicher Sicht selten notwendig.

## To EJB or not to EJB

Der geneigte J2EE-Kenner fragt sich nun sicherlich, warum wir bisher mit keiner Silbe Enterprise Java Beans (EJBs) erwähnt haben. Wir haben diese Frage lange und ausführlich diskutiert. Dabei war entscheidend, dass wir wie überall in der Architektur zur Risikominimierung nichts ohne signifikanten Nutzen einführen. Die Frage war also, welchen Nutzen haben wir von EJBs und welche Risiken und/oder Nachteile stehen dem gegenüber. Im folgenden nun ein kurzer Abriss der entsprechenden Überlegungen.

### Entity Beans für Entitäten

Bei naiver Betrachtung bietet sich der Einsatz von Entity Beans für Entitäten an, schon aufgrund der Namensgleichheit. Allerdings ist dies ein Trugschluss. Entity Beans zeichnen sich vor allem durch zwei wichtige Eigenschaften aus: Persistenz und Threadsafety. Persistenz bedeutet, dass Änderungen an einer Entity Bean Instanz direkt in die Datenbank geschrieben werden (d.h. am Ende der entsprechenden Transaktion). Threadsafety bedeutet, dass mehrere Clients parallel auf eine Instanz zugreifen können – die Instanz serialisiert den Zugriff automatisch. Man erkauft sich diese Features aber auch mit einem recht erheblichen Laufzeitoverhead.

In unserem Falle werden all diese Features nun aber nicht benötigt: Entitäten sind nur im Rahmen des Prozesskontextes persistent, und paralleler Zugriff verschiedener Clients findet per Definition nie statt, eine Entität gehört genau einem Prozess und ist von allen anderen isoliert. Aus diesem Grund wäre die Verwendung von Entity Beans kontraproduktiv und wir haben auf deren Verwendung für Entitäten verzichtet.

### Entity Beans für Prozesskontext

An einer Stelle haben wir allerdings Entity Beans eingesetzt: Und zwar zur Repräsentation des Prozesskontextes. Jeder Kontext ist eine Entity Bean Instanz. Damit haben wir genau einen entscheidenden Anker, um mit den EJB-Mitteln den gesamten Zustand des aktuellen Geschäftsprozesses zu sichern (sei es zur fachlichen Zwischenspeicherung oder als Failover-Maßnahme). Wichtig dabei ist, dass wir - um nicht bei jedem Zugriff auf den Kontext diesern automatisch zu speichern - über einen Proxy auf die Bean zugreifen.

### Stateful Session Beans für die Session

Man könnte für die Session-Daten Stateful Session Beans einsetzen. Allerdings scheint dies auch nicht besonders sinnvoll, da die Session-Daten in der HTTP-Session des Webservers gespeichert werden und alle anderen Dinge stehen in den Prozesskontexten. Wenn also alle eigentlichen Zustandsdaten ohnehin gesichert sind, stellt sich die Frage, welchen Nutzen eine Speicherung weiterer Daten bietet. Die Konsequenz ist, dass bei einem Ausfall einer Session eine neue Session gestartet werden muss (inklusive Anmeldung) und auf die Daten zwischengespeicherter Geschäftsprozesse explizit aufgesetzt werden muss. Dieses Verhalten ist in unserem Fall unproblematisch.

### Stateless Session Beans für die Komponenten

Auf den ersten Blick erfüllen Stateless Session Beans alle Voraussetzungen für den Einsatz im Falle unserer Komponenten: stateless, gepoolt, nicht persistent. Das stimmt zwar, allerdings kommt nun noch eine weitere Sache ins Spiel: Unsere Komponenten müssen nicht remote zugreifbar sein. Und diese Remote-Fähigkeit bringt bei allen EJBs, auch bei Stateless Session Beans, einen gewissen Laufzeit-Overhead. Wir haben uns entschieden,

das Pooling selbst zu bauen, um diesen Laufzeit-Overhead nicht zu bekommen. Mit den Local-Interfaces von EJB 2.0 wäre diese Entscheidung vielleicht anders ausgefallen.

### EJBs zur Remote-Kommunikation

Wir haben EJBs noch zu einem weiteren Zweck „missbraucht", und zwar zur Anbindung entfernter Maschinen: Es gibt die Anforderung, einige Dinge per nativer C++-DLL zu berechnen bzw. auf Systemdienste zuzugreifen. Nun ist es auf dem Papier verboten, aus einer EJB heraus eine DLL anzusprechen. In der Praxis, wenn man das in einem kontrollierten Umfeld tut, funktioniert es aber durchaus – meistens. Und weil wir dazu doch nicht so sehr viel Vertrauen haben, lagern wir die Beans mit DLL-Zugriff auf einen extra Server aus, sodass bei dessen Absturz wenigstens das restliche System intakt bleibt. Die EJBs dienen hier also ausschliesslich dem Remote-Zugriff. Wir hätten dies zwar auch mit RMI direkt machen können, aber die Standard-RMI-Implementierung ist nun mal nicht sehr effektiv im Umgang mit Resourcen, oder bei gleichzeitigen Zugriffen von mehreren Clients.

## Das Programmiermodell

Einer der größten Vorteile von Standards und Referenzarchitekturen wie der J2EE liegt darin, dass sich darauf basierend ein konsistentes Programmiermodell für Entwickler definieren lässt: für typische Fälle werden Standard-Lösungen und Vorgehensweisen angeboten. Dies erleichtert den Umgang mit dem System und versteckt möglicherweise inhärente Komplexität.

Anders herum formuliert: Eines Systemarchitektur wird nur dann von den Entwicklern wirklich gelebt, wenn es ein verständliches, konsistentes und idealerweise recht einfaches Programmiermodell gibt. Ein Programmiermodell beschreibt also, wie für eine bestimmte Architektur richtig und effizient entwickelt werden kann.

Daher haben auch wir für unsere „Anwender", also die Entwickler der bankfachlichen Prozesse, Aktivitäten, Komponenten und GUIs ein solches Programmiermodell definiert. Es beschreibt sehr genau die Verantwortlichkeiten der verschiedenen Teile des System und deren „Rechte". Obwohl die vorgestellten Konzepte zunächst vielleicht recht komplex klingen, ist das Programmiermodell für die Entwickler recht einfach, sicherlich nicht komplizierter als das der direkten J2EE Programmierung (man denke nur mal an den Lifecycle von EJBs und dessen Konsequenzen für das Entwickeln *effizienter* EJBs). Die Tatsache dass das System innerhalb J2EE läuft, ist in diesem Programmiermodell praktisch nicht mehr sichtbar.

Das Programmiermodell wurde durch ein einführendes Dokument, verschiedene Beispiele und Walkthroughs, sowie einer Menge an Missionars- und Reviewarbeit etabliert, und es wird in der Praxis auch befolgt und umgesetzt.

## Zusammenfassung

Sicherlich ist klar, dass sich ein solcher Prozess-orientierter Ansatz nicht in jedem Projekt sinnvoll einsetzen lässt. Was kann man nun generell aus dem obigen Praxisbericht lernen? Nun, wir denken, verschiedene Dinge:

Erstens lohnt es sich auch bei Standardarchitekturen wie der J2EE, darüber nachzudenken, ob und wie die verschiedenen Konzepte eingesetzt werden sollten. Nicht immer ist die offensichtlichste Lösung auch die beste. So ist insbesondere der Einsatz von EJBs zu hinterfragen.

Zweitens haben auch relativ betagte Konzepte (wie z.B. API-artige, zustandslose Komponenten) auch heute noch ihre Berechtigung. Oder umgekehrt formuliert kann eine Trennung in „intelligente Datenstrukturen“ und Services nach wie vor durch die damit verbundenen klaren Regeln sehr hilfreich sein.

Drittens wird deutlich, dass Kompromisse an der richtigen Stelle (z.B. Failover und Neuanmeldung) ein System deutlich vereinfachen können, ohne den Nutzern grosse Unannehmlichkeiten zu bescheren.

Viertens sind in großen Projekten Bottlenecks jeglicher Art zu vermeiden. Das bedeutet insbesondere, dass das so oft gelehrte zentrales Objekt-Modell unter Umständen keine angemessene Lösung darstellt. Auch hier bestätigt sich die Regel, dass große Systeme nicht objektorientiert sondern modular aufgebaut sein müssen, und die Objektorientierung eher in den jeweiligen „Mikrostrukturen“ zum tragen kommt.

Daraus folgt fünftens, dass die Größe eines Projekts maßgeblichen Einfluss auf die Architektur haben kann. Auch die Anzahl und Organisation der Mitarbeiter gehört zu den beeinflussenden Randbedingungen (oder nichtfunktionalen Requirements).

Last not least wird bei all den Erfahrungen hoffentlich offensichtlich, dass es nie gut ist, nach vorgefassten Paradigmen und Modeerscheinungen zu entwerfen. Es ist völlig unerheblich, wie objektorientiert unsere Lösung ist oder ob sie XML und EJBs verwendet. Entscheidend ist, dass die Lösung für unser Problem angemessen ist. Und zwar unter Abwägung aller Kosten und Risiken. Entsprechende Diskussionen wurden im Projekt durchaus nicht selten geführt. Zwei Fragen spielten dabei immer eine entscheidende Rolle: „Warum?“ und „Wer hat konkrete praktische Erfahrung?“. Bei allen Technologien, Werkzeugen und Hilfsmitteln wurden nur die verwendet, die eine entsprechende praktische Erfahrung aufweisen und einen erkennbarem Vorteil nachweisen konnten. Ein guter Systemarchitekt ist ein sehr misstrauischer Mensch, der aber weiß, wem er wann vertrauen kann.