



Qualität ist das beste Rezept

Komponenten mit J2EE-Patterns

Gerd Beneken und Manfred Schamper

Enterprise-Beans sind server-seitige Komponenten. Sie enthalten nur fachlichen Code, während die technischen Eigenschaften wie Transaktionssteuerung oder Persistenz im Deployment-Deskriptor festgelegt werden. Das funktioniert nur für einfache Systeme. Erfahrungen aus mehreren Projekten zeigen, dass bei komplexeren Anforderungen größere Komponenten als Enterprise-Beans erforderlich sind. Auch die Trennung von Anwendungscode und Technik ist nur über eine saubere Systemarchitektur möglich. Sie ist nicht mehr deklarativ über den Deployment-Deskriptor zu erreichen. Über die Kombination von J2EE-Patterns können grobgranulare Anwendungskomponenten implementiert werden, außerdem lässt sich technischer Code von fachlichem Code sauber isolieren.

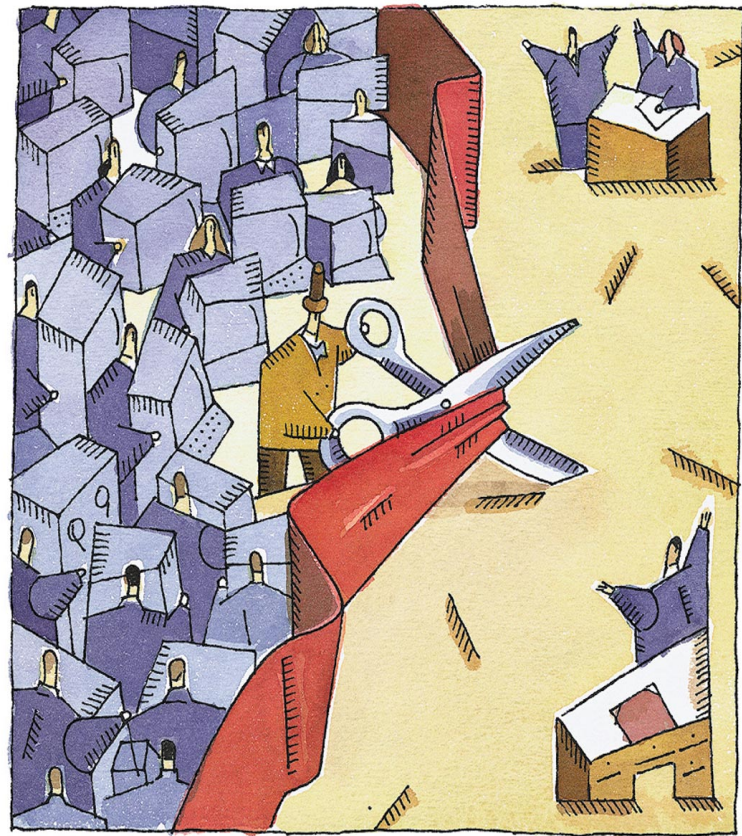
Flexibilität, Kostenreduzierung und verringerte Time-to-Market durch Einsatz wiederverwendbarer Komponenten ist in anderen Ingenieursdisziplinen schon lange Realität. Im Software-Engineering verspricht die Komponententechnik ähnliche Vorteile. Viele Unternehmen setzen daher auf Enterprise-JavaBeans (EJB) als Entwicklungsmodell für den Server und erstellen damit geschäftskritische Anwendungen. Software-Architekten erwarten sich von Enterprise-JavaBeans drei Vorteile:

- ▼ Das Entwicklungsteam kann sich vollständig auf das fachliche Problem konzentrieren. Es programmiert die fachliche Logik in Enterprise-Beans. Die technischen Aspekte wie Transaktionen, Persistenz oder Sicherheit werden deklariert.
- ▼ Der Anwendungskern am Server wird über das Komponentenmodell von EJB strukturiert. Eine Basisarchitektur ist durch EJB bereits vorgegeben, beispielsweise die Trennung von Geschäftsvorfällen in Session-Beans und Geschäftsobjekten in Entity-Beans.
- ▼ Eine leistungsfähige Laufzeitumgebung, der Applikationsserver, sorgt dafür, dass die Anwendung viele Clients und eine große Transaktionslast verkraftet. Um die Probleme der Lastverteilung und des Ressourcen-Managements sollte sich der Architekt kaum Gedanken machen müssen.

Praktische Erfahrungen in vielen unserer Projekte haben gezeigt, dass diese Vorteile nur für einfache Systeme direkt nutzbar sind. Komplexe betriebliche Informationssysteme erfordern nach wie vor eine solide Anwendungsarchitektur. Die derzeit verfügbaren Entwurfsmuster [Alu01] sind dabei eine Hilfe.

Komponentenbegriff

Um von einer Architektur auf der Basis von Komponenten zu sprechen, muss der Begriff Komponente genauer definiert werden. Eine Komponente erfüllt die folgenden vier Eigenschaften:



- ▼ Komponenten implementieren wohldefinierte Schnittstellen,
- ▼ haben explizite Abhängigkeiten,
- ▼ können unabhängig entwickelt, installiert, betrieben und konfiguriert sowie
- ▼ mit anderen Komponenten kombiniert werden.

Diese Eigenschaften sind notwendig, um große Systeme zu strukturieren und damit eine kostengünstigere Wartung zu erreichen. Wichtig ist in diesem Zusammenhang, die Abhängigkeiten der Komponente von ihrer Umgebung und von anderen Komponenten explizit zu machen und möglichst gering zu halten. Technische oder fachliche Änderungen sollen nur die Komponente betreffen, nur sie sollte neu ausgeliefert werden. EJB erfüllt diese Forderungen zum Teil [Gro01].

Granularität

Zentrales Problem beim Entwurf einer Komponente ist die richtige Wahl ihrer Größe. Eine Komponente soll groß genug sein, um eine fachlich sinnvoll nutzbare Einheit darzustellen, aber nicht zu groß. Komponenten wie „Kunde“ oder „Konto“ sind zu klein um einer umfangreichen Anwendung Struktur zu verleihen. Wegen der großen Zahl solcher kleiner Komponenten geht die Übersicht verloren. Auf der anderen Seite dürfen Komponenten nicht zu grobkörnig werden. Übergroße Komponenten haben leicht die altbekannten Probleme der Software-Monolithen.

Empfohlen wird hier die Orientierung an identifizierbaren fachlichen Komponenten wie „Kundenverwaltung“ und „Rechnungsstellung“. Komponenten werden somit nahezu automatisch zu Bestandteilen der Systemarchitektur. Wie in [Gri98] beschrieben ist es wichtig, diese von den Bestandteilen der Implementierung – den Klassen bzw. Objekten – zu unterscheiden. Jedoch führt die EJB-Spezifikation Entwickler schnell auf die schiefe Bahn, da sie Komponenten mit Klassen gleichsetzt bzw. maximal auf die Ebene von Aggregaten hebt. Das führt zur Verwechslung solcher feingranularer Implementierungseinheiten mit Komponenten und resultiert in einer unübersichtlichen feingliedrigeren Systemstruktur.

Kopplung

Schmale Schnittstellen sind die Voraussetzung, um die Kopplung zwischen interagierenden Komponenten zu vermindern. Zugriffe auf Komponenten erfolgen über wenige Methoden, die im inneren komplexe Arbeitsabläufe verbergen können. Ein Nutzer der Komponente arbeitet auf Ebene dieser Dienste und muss nicht auf einzelne Geschäftsobjekte zugreifen und sich nicht um deren Beziehungen untereinander kümmern. Die nach außen hin geringe Zahl der Zugriffspunkte auf die Komponente stellt eine übersichtliche Zusammenfassung der gebotenen Dienste dar und bündelt externe Abhängigkeiten.

Die EJB-Spezifikation 1.1 sieht für jede Enterprise-Bean eine Remote-Schnittstelle vor. Jede Enterprise-Bean wird damit nicht nur von anderen Enterprise-Beans derselben Komponente, sondern auch potentiell vom Client aus angesprochen. Die Schnittstelle zwischen Client und Server wird damit sehr groß. Erst mit EJB 2.0 [EJB2.0] wird dieses Problem über lokale Schnittstellen gemildert. Mit lokalen Schnittstellen kann der Architekt zwischen internen und externen Schnittstellen unterscheiden.

Trennung von Zuständigkeiten

Ein wichtiger Grundsatz in der Softwareentwicklung ist seit langer Zeit die Trennung von Zuständigkeiten, *Separation of Concerns*, [Dij76]. Fachliche und technische Rahmenbedingungen ändern sich mit unterschiedlicher Geschwindigkeit (beispielsweise gibt es fast jedes Jahr eine neue EJB-Spezifikation). Die Trennung von anwendungs- und technikbestimmtem Code [SD00] ist daher von zentraler Bedeutung, um Änderungsaufwände bei der Wartung zu begrenzen.

Schwächen in der Abbildung von Entity-Beans auf die Datenbank über Container Managed Persistence (CMP) führen dazu, dass die Enterprise-Beans, die in der Theorie nur fachlichen Code enthalten sollten, mit JDBC und darin enthaltenem SQL verschmutzt werden. In der Praxis werden Entwickler bei der Implementierung von Bean Managed Persistence (BMP) durch load- und store-Methoden im Entity-Bean-Interface sogar dazu verleitet, technischen Code mit Anwendungscode zu mischen, was die spätere Wartung deutlich erschwert.

Verwendung von Mustern

Die oben beschriebenen Probleme können durch eine Kombination von J2EE-Mustern [Alu01] gelöst werden. Diese erleichtern es, Anwendungskomponenten nach fachlichen Vorgaben aus mehreren Enterprise-Beans und anderen Java-Klassen zusammenzufügen und mit einer schmalen Schnittstelle zu versehen. Zusätzlich können anwendungs- und technikbestimmter Code sauber getrennt werden.

Die drei wichtigsten Entwurfsmuster zum Schneiden von Komponenten sind *Business Proxy* am Client und *Session Facade* sowie *Data Access Object* (DAO) am Server (s. Abb. 1). Zusätzliche Muster wie Value List Handler lösen Detailprobleme.

Am Client entkoppelt der *Business Proxy* (eine Variante des Proxy-Musters aus [Gam94]) die Präsentation und die Dialogsteuerung von der konkreten Middleware. Der Client kann so programmiert werden, als ob es EJB nicht gäbe. Der *Business Proxy* verfügt über eine fachliche Schnittstelle. Die Dialoge werden gegen diese Schnittstelle programmiert. Im *Business Proxy* werden beispielsweise die `java.rmi.RemoteExceptions` in fachlich sinnvolle Ausnahmen umgesetzt. Die Konvertierung von Transportstrukturen sowie eine Unterstützung der Ausfallsicherheit durch transparentes Wechseln des Servers ist ebenfalls möglich.

Für den Test- oder Demobetrieb von Anwendungs-Clients bringt ein Proxy den Vorteil, dass dahinter Dummy-Implementierungen des Anwendungskerns einsetzbar sind. Die Entwicklung wird flexibler und es müssen nicht unbedingt performanzhungrige Applikationsserver installiert werden.

Die *Session Facade* bündelt Enterprise-Beans und Java-Klassen zu grobgranularen Anwendungskomponenten. Nur über die *Session Facade* wird der Anwendungskern angesprochen. Damit ist sie die Schnittstelle der Komponente, sie definiert die zur Verfügung gestellten Dienste. Da die Zahl der Methodenaufrufe auf die Geschäftsobjekte gewöhnlich hoch ist, führt die server-seitige Zusammenfassung dieser Aufrufe in der Fassade zu geringerer Auslastung der Netzwerkinfrastruktur.

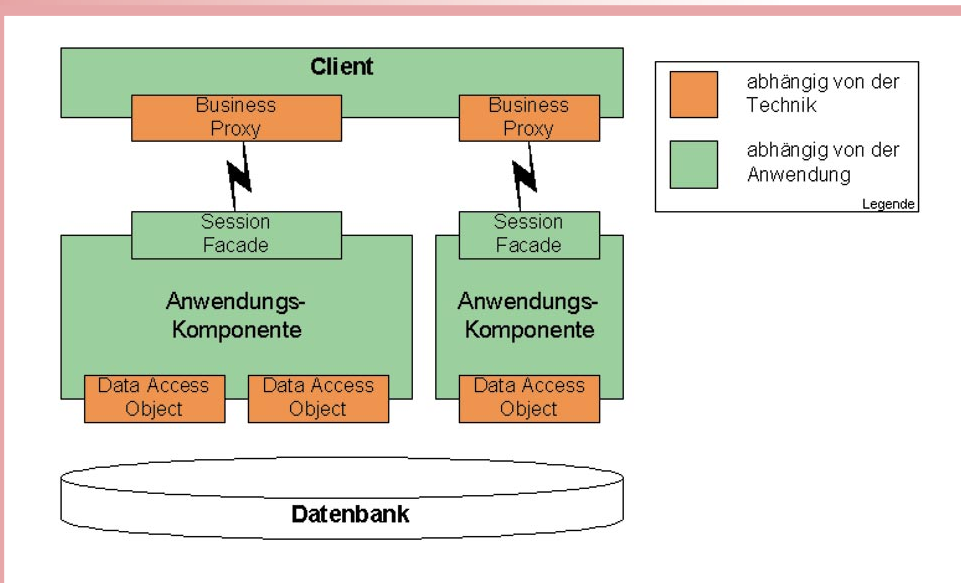


Abb. 1: Die drei wichtigsten Entwurfsmuster zum Schneiden von Komponenten sind *Business Proxy* am Client und *Session Facade* sowie *Data Access Object* (DAO) am Server

Das *Data Access Object* (DAO) kapselt den Datenbank- oder Nachbarsystemzugriff. Die DAOs bilden die einzige Stelle im System, wo JDBC oder SQL zu finden sind. Das DAO wird von mehreren anderen J2EE-Mustern verwendet (z. B. Fast Lane Reader und Value List Handler) und kann ebenso gut in herkömmlichen Java-Anwendungen Einsatz finden. Entity-Beans erledigen ihren Datenhaushalt ebenso über DAOs. Mischformen mit CMP und DAO sind denkbar.

Im Sinne der Komponentendefinition sind DAOs ein Teil der Definition externer Abhängigkeiten einer Komponente. Durch verschiedene DAO-Implementierungen werden Anwendungs-komponenten portabel.

Das Innenleben einer durch die Muster isolierten Komponente kann separat betrachtet werden. Diese Innensicht ist für den Benutzer der Komponente unerheblich. Er verwendet nur die durch die Session Facade zur Verfügung gestellten Dienste, auf die er über den Proxy technikunabhängig zugreift (s. Abb. 2).

Fazit

Die Komponenten- und Schnittstellen-Gedanken der EJB-Spezifikation genügen alleine nicht, um große Systeme zu strukturieren. Eine Systemarchitektur auf höherer Ebene ist notwendig. Die Muster, die [Alu01] und andere Quellen vorschlagen, erlauben eine Strukturierung in grobe fachliche Komponenten. Mit Hilfe dieser Muster werden fachliche Komponenten geschnitten und der notwendige technische Code vom Anwendungscode entkoppelt.

Literatur

[Alu01] D. Alur, J. Crupi, D. Malks, Core J2EE Patterns: Best Practices and Design Strategies, Prentice Hall, 2001

[Dij76] E. W. Dijkstra, A discipline of programming, Prentice Hall, 1976

[EJB2.0] EJB-Spezifikation 2.0, Proposed Final Draft 2, Sun Microsystems, <http://java.sun.com/products/ejb/docs.html>

[Gam94] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object Oriented Software, Addison-Wesley, 1994

[Gri98] F. Griffel, Componentware – Konzepte und Techniken eines Softwareparadigmas, dpunkt, 1998

[Gro01] S. Große, EJB und Geschäftsanwendungen – Mythos und Realität, in: JavaSpektrum 2/01

[Mar02] F. Marinescu, <http://www.theserverside.com/resources/patterns-review.jsp>

[SD00] J. Siedersleben, E. Denert, Wie baut man Informationssysteme? Überle-

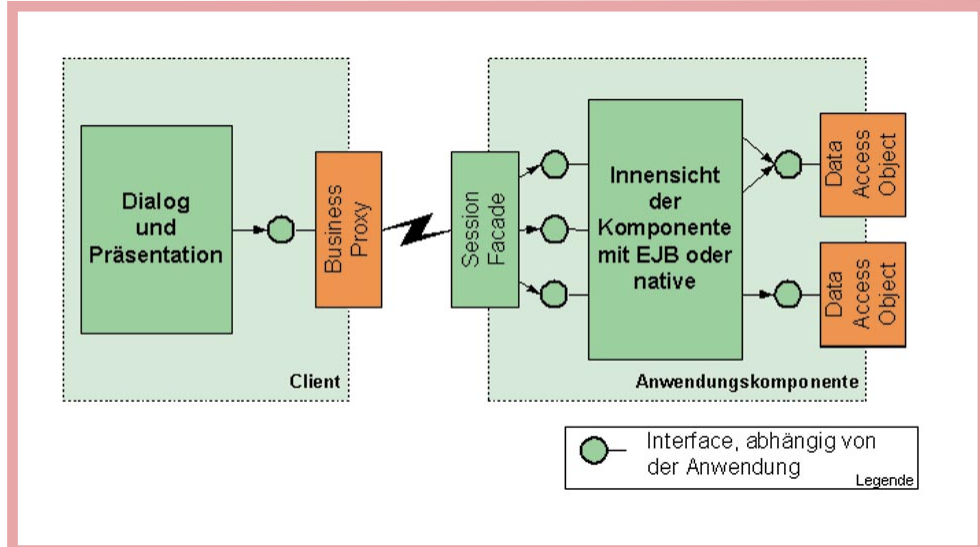


Abb. 2: Innen- und Außensicht (Schnittstelle) einer Komponente

gungen zur Standardarchitektur, Informatik Spektrum 23.4, S. 247-257, 2000

[Sun02] SunBlue prints, <http://java.sun.com/blueprints/patterns/j2eepatterns/index.html>

[ZB00] J. Zimmermann, G. Beneken, Verteilte Komponenten und Datenbank-anbindung: Mehrstufige Architekturen mit SQLJ und EJB 2.0, Addison-Wesley, 2000



Gerd Beneken ist Technischer Berater bei der sd&m AG. Er arbeitet derzeit für sd&m Research mit den Themenschwerpunkten Persistenz und Enterprise-Java-Beans. Er verfügt über langjährige Erfahrungen in der Entwicklung und im Design großer verteilter objektorientierter Systeme. Herr Beneken ist außerdem Lehrbeauftragter der FH Rosenheim und Mitautor des bei Addison-Wesley erschienenen Buchs „Verteilte Komponenten und Datenbank-anbindung: Mehrstufige Architekturen mit SQLJ und EJB 2.0“. E-Mail: gerd.beneken@sdm-research.de.

Manfred Schamper studiert an der LMU München Informatik mit den Schwerpunkten Software-Engineering und Datenbanken. Zur Zeit arbeitet er bei der sd&m Resarch GmbH an seiner Diplomarbeit. Er verfügt über mehrjährige Erfahrungen in der Entwicklung und im Design von Backend-Systemen großer Internet-Anwendungen bei der argo_tec GmbH, der NCI GmbH u.a. E-Mail: manfred.schamper@sdm-research.de.

▼ **Weitere Informationsquellen**

<http://www.sdm-research.de>