Fundsachenmanagement mit EJBs
Erfahrungen und Patterns

Peter Hauck

TLC GmbH
Kölner Str.26
60327 Frankfurt
peter.hauck@tlc.de

Kurzfassung. Dieser Beitrag beschreibt die EJB-Architektur einer Intranet-Anwendung auf der Basis von J2EE. Im Projekt „Verloren und Gefunden“ hat die TLC GmbH für die Deutsche Bahn AG ein Fundsachenmanagement-System erstellt. Dabei mußten Fragen beantwortet werden, die sich jeder EJB-Architektur stellen: Verwendung von Entity- und SessionBeans, Ort der Geschäftslogik, Fehlerbehandlung, Kommunikation mit dem Client, Transaktionen und Sperren. Bei der Vorstellung der gewählten Lösung wird auch auf einige J2EE-Patterns von SUN eingegangen.

Die Anwendung „Verloren und Gefunden“

Vor der Einführung der Anwendung erfolgte die Verwaltung der Fundsachen bei der Bahn manuell. Das neue, dv-gestützte Verfahren reduziert zum einen die Verwaltungskosten, zum anderen wird die Rückgabequote von Fundsachen deutlich verbessert. Die Anwendung wird zunächst nur im Intranet genutzt. Eine Ausweitung auf das Internet ist eine Option für die Zukunft.
Verlorene Gegenstände fallen bei über 200 Fundstellen an. Nach einem festgelegten Zeitraum werden diese Fundsachen an ein zentrales Fundbüro weitergeleitet und dort nach einem weiteren definierten Zeitraum versteigert, bzw. vernichtet, falls sie nicht vorher dem Verlierer zugeordnet und zurückgegeben werden konnten. Verlustmeldungen der Kunden werden u.a. in einem Callcenter entgegengenommen. Das System ist für die Erfassung von etwa 200.000 Fundsachen und bis zu 600.000 Verlustmeldungen pro Jahr ausgelegt.

Fachobjekte

Die wichtigsten Entitäten sind Fundsachen und Verlustmeldungen. Für beide gibt es eine Reihe von beschreibenden Attributen. Anhand dieser Attribute können Fundsachen mit Verlustmeldungen abgeglichen werden.
Weitere Entitäten sind Organisationseinheiten und Benutzer (mit allem was zu einer Rechteverwaltung notwendig ist). Einschließlich der Verknüpfungstabellen (für N:N – Beziehungen) und der Tabellen für Logging ergeben sich an die 30 Tabellen.

Fachliche Logik

Neben der Erfassung von Fundsachen und Verlustmeldungen sowie der Bereitstellung von Statistiken und Belegen sind die Nachforschung nach verlorenen Gegenständen und die Verwaltung des Lebenszyklus der Fundsachen die Kernaufgaben des Systems.

Nachforschung. Zwei Arten der Nachforschung sind zu unterscheiden. Bei der *automatischen Nachforschung* werden täglich alle offenen Verlustmeldungen mit den noch nicht zugeordneten Fundsachen abgeglichen und bei genauer Übereinstimmung wird eine Fundsache der entsprechenden Verlustmeldung zugeordnet. Bei der *manuellen Nachforschung* wird dem Bearbeiter zu einer bestimmten Verlustmeldung eine Liste von möglicherweise passenden Fundsachen präsentiert. Die Gegenstände in der Liste sind nach dem Grad der Übereinstimmung sortiert.
Der Abgleich von Fundsachen mit Verlustmeldungen erfolgt nach Kriterien wie Verlustdatum, Fundort, Farbe, Hersteller, Warengruppe, etc. Manche Kriterien müssen exakt übereinstimmen, andere spielen nur für die Festlegung des Übereinstimmungsgrades eine Rolle. Die Kriterien und ihre Wertigkeit sind zur Laufzeit änderbare Parameter.
Lebenszyklus der Fundsachen. Eine Fundsache durchläuft im System wie im echten Leben einen komplexen Lebenszyklus. Ein Statusübergang wird automatisch nach bestimmten Fristen oder durch Benutzeraktionen ausgelöst. Eine Fundsache wird bei der Fundstelle abgegeben, für den Transport bereitgestellt, ins zentrale Fundbüro transportiert, dort im Lager eingeordnet, zur Versteigerung vorbereitet, eventuell mit anderen Fundsachen gebündelt und schließlich versteigert oder vernichtet. Eine erfolgreiche Nachforschung unterbricht diesen Ablauf. Der Lebenszyklus endet dann mit der Rückgabe an den Verlierer.

Die Architektur

Die Gesamtarchitektur von „Verloren und Gefunden“ (siehe Abb.1) basiert auf der J2EE-Architektur. Als Frontend wurde ein HTML-Client gewählt. Die zahlreichen

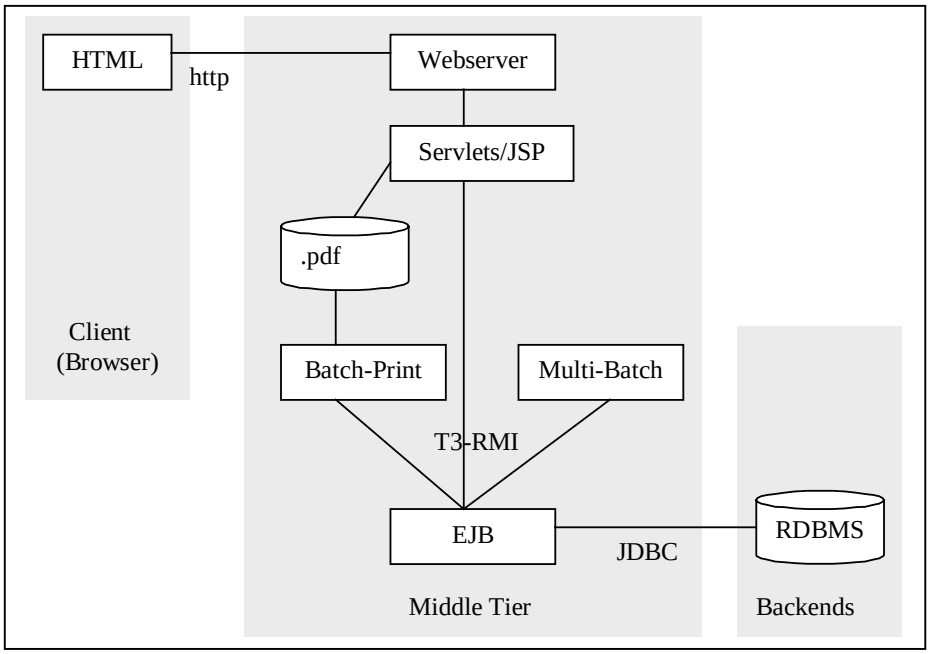


Abb.1 Gesamtarchitektur „Verloren und Gefunden“

Drucksachen werden im Online- wie auch im Batchbetrieb auf der Middle-Tier als PDF-Files generiert. Der sogenannte „Multi-Batch“ faßt eine Reihe von Batchprozessen zusammen. Diese sind aus EJB-Sicht ganz normale Java-Clients. Auf die zentrale, relationale Datenbank greifen ausschließlich die EJBs zu.

Enterprise JavaBeans

Als wir im Frühjahr 2000 die Architektur von „Verloren und Gefunden“ festlegten, waren die J2EE-Patterns von SUN noch nicht publiziert.[1] Umso spannender war es, jetzt zu untersuchen, ob wir einige dieser Patterns in unserer Anwendung wiederfinden würden. Wo dies der Fall war, wird nach der Beschreibung des gewählten Lösungsansatzes auf das entsprechende Pattern eingegangen.

Ort der fachlichen Logik

EJB-Entwickler müssen sich entscheiden, wo sie welche fachliche Logik implementieren. EntityBeans, SessionBeans und nicht zuletzt ganz normale Java-Klassen oder Java-Beans bieten sich an.
In „Verloren und Gefunden“ ergaben sich die Antworten fast von selbst. Der naheliegende Ort für die Verwaltung des Lebenszyklus einer Fundsache ist eine Fundsache-EntityBean. Hier stehen alle notwendigen Informationen zur Verfügung, die hier umzusetzende Logik ist kontextunabhängig. Der Anstoß für einen Statuswechsel kommt zwar von außen, die Umsetzung erfolgt aber in der EntityBean.
Die Nachforschungslogik wurde weitestgehend durch zur Laufzeit erzeugtes SQL realisiert. Damit befindet sich völlig regelwidrig und im Gegensatz zu den meisten Lehrbüchern ein relevanter Teil der Geschäftslogik in der Persistenzschicht.[2] Eine reine Java-Implementierung für die Nachforschung wäre wegen der Instantiierung sehr großer Objektmengen zu langsam geworden und wurde daher nicht in Betracht gezogen.

SessionBeans

SessionBeans werden als Fassade genutzt. Sie bieten die fachlichen Funktionen als Services an. Die SessionBeans nutzen EntityBeans, greifen aber auch selbst über *Data Access Objects (DAOs)* auf die Datenbank zu. Alle Updates gehen über EntityBeans. Es gibt mehrere SessionBeans. Kriterien zum Schneiden von SessionBeans sind

- fachlicher Kontext
- Client-Typ (in unserem Fall z.B. Online und Batch)
- Security-Anforderungen.

Alle SessionBeans konnten zustandslos gehalten werden. Die SessionBeans delegieren an EntityBeans und implementieren die Logik, die mehrere EntityBean-Instanzen betrifft. Zum einen sind das Methoden, deren Logik mehrere Geschäftsobjekte verschiedenen Typs benötigt. Zum anderen sind es aber auch Mengenoperationen, die sich auf einen Typ beziehen. In einer EJB2.0 Umgebung könnte man die letztgenannten Methoden auch im HomeInterface der betreffenden EntityBean definieren.

---

[1] Das Programmiermodell [Blueprint] war bereits veröffentlicht und diente uns teilweise als Leitfaden.
[2] Immerhin konnte auf den Einsatz von Stored Procedures verzichtet werden.

Session Facade Pattern

Das Fassadenmuster (siehe Abb.2) kann fast schon als klassisch bezeichnet werden.

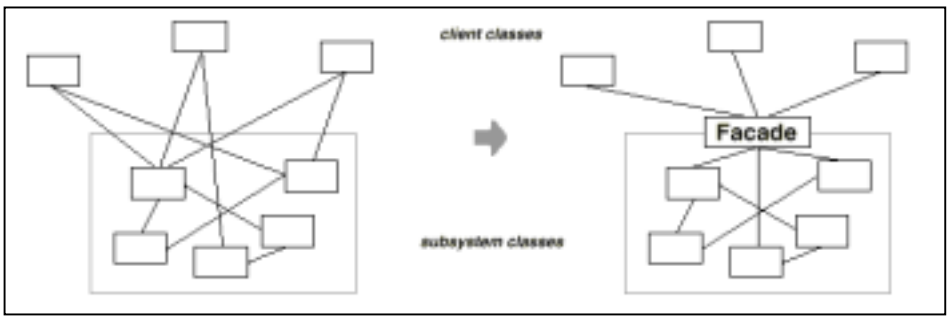


Abb.2 Klassendiagramm Fassadenmuster (aus [Gamma], s.189)

In [J2EEPattern] begegnet uns das Fassadenmuster als *Session Facade Pattern*.[3] Dieses adressiert drei Probleme:

- Enge Kopplung mit direkter Abhängigkeit der Clients von den Fachobjekten.
- Zu viele Methodenaufrufe zwischen Client und Server, was zu Netzwerk-Performance-Problemen führen kann.
- Fehlen einer einheitlichen Client-Zugriffs-Strategie und damit Exponierung der Geschäftsobjekte für möglichen Mißbrauch.

Die Lösung besteht in der Benutzung von SessionBeans als Fassade zur Kapselung der Komplexität der Interaktionen zwischen Geschäftsobjekten, die an einem Geschäftsprozeß beteiligt sind (siehe Abb.3). Die *Session Facade* steuert die Geschäftsobjekte und stellt eine einheitliche, grobgranulare Service-Zugriffsschicht für die Clients bereit.

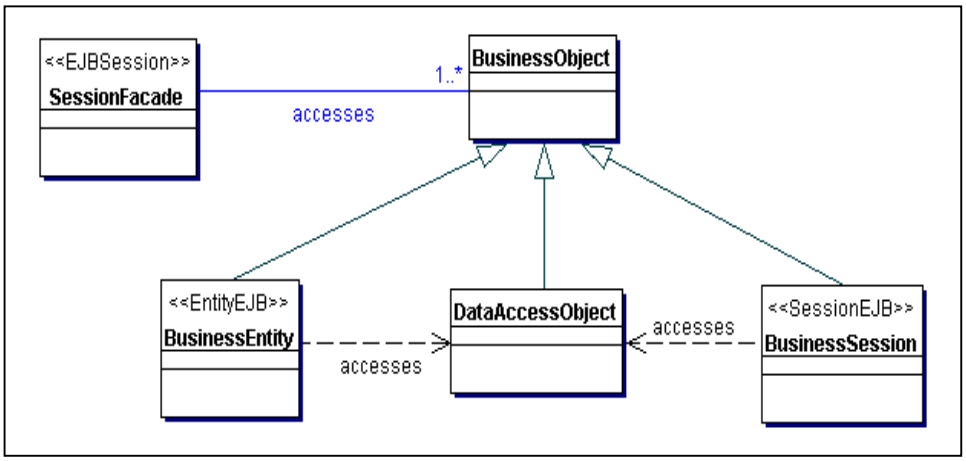


Abb.3 Klassendiagramm *Session Facade Pattern* (aus [J2EEPattern])

[3] In [BluePattern] wird eine *Session Entity Facade* beschrieben. Dieses Pattern ist als API für EntityBeans gedacht, welches auf Client- oder Serverseite implementiert sein kann.

Geschäftsobjekte können als EntityBean, als SessionBean, als *Data Access Object* oder als ganz normale Java-Klasse implementiert sein. Die Nutzung von Session-Beans zur Implementierung einer Schicht an der Containergrenze ermöglicht auch zentralisiertes Sicherheits- und Transaktionsmanagement. Ein weiterer Pluspunkt ist die Vereinheitlichung des ExceptionHandlings.
Dieses Pattern findet sich in unserer Anwendung praktisch in der beschriebenen Form wieder. Nach unseren Erfahrungen kann dieses Pattern uneingeschränkt empfohlen werden. Weiter zu diskutieren ist der Spezialisierungsgrad der Methoden. Sollen nur einige wenige sehr allgemeine Methoden mit Aktions- oder Befehlsobjekte als Parameter definiert werden, oder soll es für jede Geschäftsfunktion einen eigenen Service mit detaillierter Parameterliste geben? Diese zwei Pole stecken hier die Bandbreite der möglichen Lösungen ab.
Das Pattern adressiert ausschließlich die Serverseite. Das Pendant auf der Clientseite ist das *Business Delegate Pattern* [J2EEPattern]. Zusammengenommen bilden *Session Facade* und *Business Delegate* eine gedehnte Fassade im Sinne von Fowler.[4]

EntityBeans

EntityBeans repräsentieren die persistenten Geschäftsobjekte. In „Verloren und Gefunden“ entspricht meistens eine Entität einer Datenbank-Tabelle. Nicht jede Datenbanktabelle wird jedoch als EntityBean dargestellt (z.B. Log-Tabellen). Assoziationen zwischen Geschäftsobjekten werden in den EntityBeans implementiert. EntityBeans übernehmen auch Cache-Funktion. Eine fachliche Vererbung existiert in „Verloren und Gefunden“ nicht. EntityBeans stellen nur in Ausnahmefällen Getter und Setter auf einzelne Attribute zur Verfügung, für die Weitergabe an Clients werden generisch Transportobjekte erzeugt. In EntityBeans hat auch ein Teil der fachliche Logik ihren Platz. Neben instanzbezogene Prüfungen sind das beispielsweise die Abbildung des Lebenszyklus oder Statuswechselaktionen.
ContainerManaged Persistence(CMP) wird verwendet, wo einfache, statische Finder[5] genügen und eine Bean sich auf genau eine Tabelle abbilden läßt. Für die Abbildung der Stammdaten sind EntityBeans mit CMP bestens geeignet. Durch die Verlagerung von technischer Funktionalität in Superklassen, können sie sehr einfach sein und keine einzige Methode implementieren. CMP ermöglicht in diesen Fällen ein standardisiertes Mapping von Bean-Attributen auf Tabellenspalten.
BeanManaged Persistence (BMP) kommt demgegenüber zum Einsatz, wo eine Bean auf mehrere Tabellen zugreift und/oder dynamische Finder-Methoden erforderlich sind. Fundsache und Verlustmeldung sind Beispiele für solche EntityBeans. BMP-EntityBeans greifen nicht selbst auf die Datenbank zu, sondern delegieren diese Funktionalität an sogenannte *Data Access Objects*.

[4] Fowler beschreibt das „Dehnen von Fassaden in Client/Server-Umgebungen“ ([Fowler], s.270).
[5] Dynamischen Finder-Methoden wären dagegen solche, deren Selektionskriterien erst zur Laufzeit aufgrund von Parametern in Tabellen oder aufgrund von Benutzereingaben festgelegt werden können.

Data Access Objects (DAOs)

Alle Datenbankzugriffe erfolgen von eigenen Java-Klassen aus, die die SQL-Zugriffe kapseln. In EJBs sind also kein JDBC-Aufrufe zu finden. Die Implementierung ist hier wenig generisch. Die *DAO*-Klassen sind auf ihre spezifischen Anforderungen zugeschnitten.

DAO Pattern[6]

Das *DAO* Pattern zielt auf die Entkopplung von Geschäfts- und Persistenzlogik. Die Persistenz soll in einer eigenen Schicht von Datenzugriffsobjekten implementiert werden. Jedes Geschäftsobjekt nutzt ein passendes Datenobjekt (siehe Abb.4).

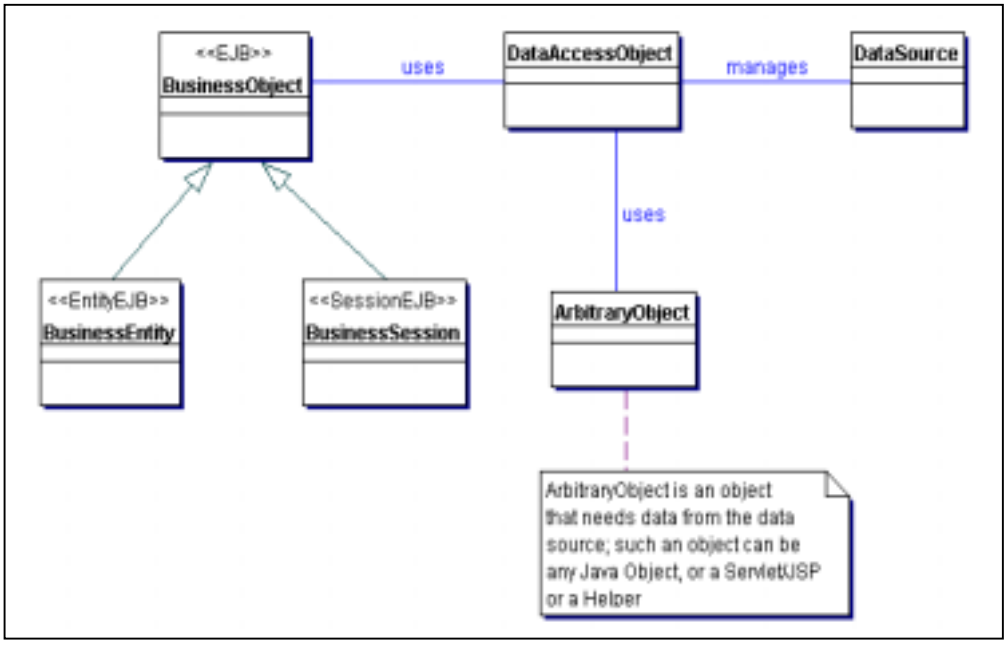


Abb.4 Klassendiagramm *Data Access Object Pattern* (aus [J2EEPattern])

Was hier vorgeschlagen wird, ist ein einfacher Delegationsmechanismus im Sinne der „separation of concerns“. Die Vorteile diese Musters sind Transparenz, leichtere Migration auf einen andere Datenspeicher oder von BMP nach CMP und geringere Komplexität der Fachobjekte.

Im Projekt „Verloren und Gefunden“ haben wir die Persistenzlogik ebenfalls in einer eigenen Schicht zusammengefaßt. Unsere Erfahrung war jedoch, daß man sich selbst bei einem recht kleinen Projekt sehr schnell mitten in der Entwicklung eines Persistenz-Frameworks findet. Und da ist doch zu überlegen, ob in jedem Fall eine vertikal strukturierte Schicht angemessen ist, oder ob es nicht wesentlich effektiver sein

[6] Dieses Pattern wird nach [J2EEPattern] beschrieben. In [BluePattern] wird dasselbe Pattern in anderer Form und mit Verweisen zur „Pet Store“-Musteranwendung dargestellt.

würde, eine dünne horizontal strukturierte Schicht[7] zu entwickeln, die auch in anderen, vergleichbaren Projekten wiederverwendet werden könnte.

Value Objects[8]

Die EJB-Clients haben übers Netz keinen Zugriff auf einzelne Attribute von Geschäftsobjekten. Mit dem Client werden Informationen über *Value Objects* ausgetauscht. Diese beinhalten Gruppen von Attributen, in der Regel eines Geschäftsobjektes. Eine Projektion auf einen Ausschnitt von Attributen ist möglich, ein Feature, das besonders für Listen relevant ist. *Value Objects* werden in unserer Anwendung von den EntityBeans, bzw. einer von diesen genutzten Factory, mittels Reflection erzeugt und über SessionBeans entweder einzeln oder als Array[9] zur Verfügung gestellt. Für Update-Operationen gibt der Client ebenfalls *Value Objects* zurück.
Für die *Value Objects* wurde ein Interface und eine Defaultimplementierung erstellt. Intern wird eine HashMap mit Attributnamen und –werten gehalten. Ein Attributwert kann auch eine Liste von Object-Ids sein.[10] Dazu kommen weitere Informationen wie Objekt-Id und Zeitpunkt der letzten Änderung. Wo die Defaultimplementierung nicht ausreichte, wurden weitere fachobjekt-bezogene Value Objekt Klassen ergänzt. Zu den *Value Objects* findet sich in [J2EEPattern] ein gleichlautendes Muster.

Value Object Pattern[11]

Um die Anzahl von zeitaufwendigen Zugriffen über das Netzwerk zu reduzieren, sollten Zugriffe auf einzelne Attribute über Getter- und Setter-Methoden vermieden werden. Statt einzelner Attribute sollen *Value Objects* in einem Methodenaufruf übertragen werden (siehe Abb.5). In einem *Value Object* werden die Attributwerte beispielsweise einer EntityBean zusammengefaßt. Grundsätzlich ist ein *Value Object* eine beliebige, serialisierbare Java-Klasse. In der einfachsten Form sind *Value Objects* nicht veränderbar und haben keine Setter-Methoden. Werden *Value Objects* dagegen auch für Updates genutzt, so spricht man von *Mutable Value Objects*. Für den Fall, daß Clients nicht immer die gleichen Attribute eines Business Objects sehen wollen, sieht das Pattern die Bereitstellung von mehreren *Value Objects* pro Business Object vor. Das sind dann *Multiple Value Objects.*

---

[7] Eine aus Säulen aufgebaute Zugriffsschicht realisiert die Zugriffsfunktionen je Fachobjekt. Eine Teilschichten-Architektur implementiert diese generisch (siehe [Hahn], s.190).

[8] In der Diskussion sind *Value Objects* auch als „Datencontainer“, „Detail Objects“, „Objects By Value(OBV)“ oder „Datenklo“ unterwegs..

[9] In „Verloren und Gefunden“ sind die Listgrößen beschränkt. Bei zu großen Ergebnismengen wird der Anwender um eine weitere Einschränkung seiner Suchanfrage gebeten. Etwas in der Art des *Value List Handler Patterns* [J2EEPattern] wurde daher nicht implementiert.

[10] Für unsere Belange genügte diese primitive Form der Abbildung von Assoziationen. Für die Erzeugung von Graphen aus *Value Objects* hält SUN das *Value Object Assembler Pattern* bereit [J2EEPattern].

[11] Das *Value Object Pattern* in [BluePattern] wirkt wenig ausgereift. Es bezieht sich allein auf „read-only data“. Ein *Value Object* wird eher als clientseitiger Proxy verstanden denn als Transportobjekt. Auch eine klare Abgrenzung zu „Dependent Objects“ [EJB 2.0] ist noch nicht vollzogen.

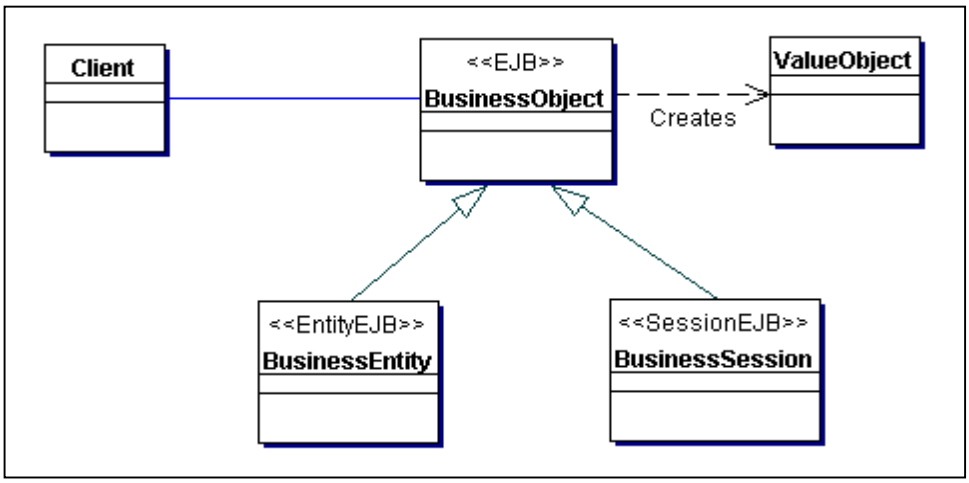


Abb.5 Klassendiagramm der einfachen Form des *Value Object Pattern* (aus [J2EEPattern])

Als Implementierung der *Value Objects* hat man sich eine oder mehrere konkrete Klassen je Business Object vorzustellen.[12] Um Code-Duplikation zu vermeiden wird von SUN die *Entity Inherits Value Object Strategy* (siehe Abb.6) vorgeschlagen.

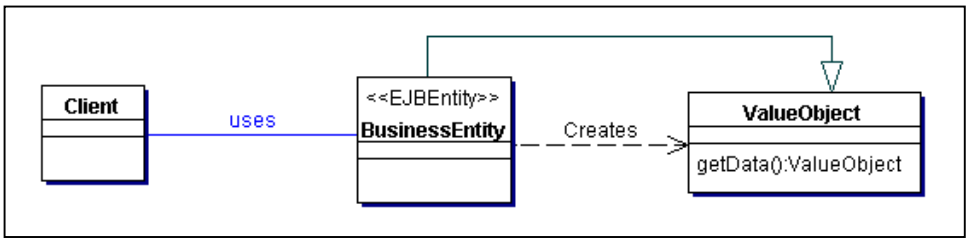


Abb.6 Klassendiagramm *Entity Inherits Value Object Strategy* (aus [J2EEPattern])

Diese Vererbungshierarchie erspart zwar einiges an Kodieraufwand, der Preis ist aber eine extrem enge Kopplung von BusinessEntity und *Value Object*. Für ContainerManaged EntityBeans auf Basis von [EJB 2.0] ist diese Strategie nicht mehr durchführbar. [13]

Bei der Diskussion der *Value Objects* kommt der Aspekt der Transaktionen zu kurz. Beispielsweise möchte man bei einer Namensänderung Vor- und Nachname in einer Transaktion ändern. Wo werden hier die Transaktionsgrenzen gesetzt? Bei feingranularen Zugriffen auf einzelne Attribute setzt der Client diese Grenzen. Es gibt aber viele gute Gründe (Sicherheitsbelange, Sperrdauer), genau das vermeiden zu wollen. Mit *Value Objects* wird es wesentlich einfacher, die Transaktionsgrenzen ausschließlich auf Serverseite zu bestimmen.

---

[12] In den Begriffen von Beschorner sind das „passive Datencontainer“, bei denen pro Fachobjekt mindestens eine Container-Klasse deklariert wird ([Beschorner], s.20).

[13] EntityBeans mit ContainerManaged Persistence werden mit [EJB 2.0] als abstrakte Klassen definiert. Die persistenten Attribute werden in diesen über Setter und Getter und nicht mehr über Instanzvariablen festgelegt.

Das *Value Object Pattern* trägt zur Begriffsbildung bei, die Implementierungsvorschläge wirken noch nicht sehr ausgereift, hier ist [Beschorner] weiter.[14] In „Verloren und Gefunden“ haben wir *Mutable, Multiple Value Objects* verwendet, die in der Implementierung den „aktiven Datencontainern“ im Sinne von Beschorner sehr nahe kommen.[15] Es sollte sich lohnen, an der Optimierung der generischen *Value Objects* zu arbeiten und weitere Implementierungsstrategien zu entwickeln.

Fehlerbehandlung

[EJB 1.1] unterscheidet zwischen „System Level“ und „Application Level“ Exceptions. Mit ersteren teilt eine Komponente mit „Mit mir ist etwas nicht in Ordnung, zumindest im Moment bin ich nicht brauchbar“, mit letzteren will sie dem Client sagen „Mit mir ist alles in Ordnung, bloß so wie Du dir das vorstellst, läuft es nicht. Sieh zu, daß Du mir alles so zur Verfügung stellst, wie ich es erwarte, dann bin ich Dir auch zu Diensten.“ Entsprechend haben wir zwei Exception-Klassen definiert. Beide können weitere Exceptions schachteln. Das Konzept der „Nested Exceptions“ findet sich auch in Java-Exception-Klassen wie *RemoteException* und *InvocationTargetException.* Da die Verschachtelung nicht einheitlich implementiert ist, haben wir eine Helper-Klasse zum Auspacken der Exceptions geschrieben.[16]
Unsere Exception-Klassen halten außerdem eine Fehlernummer[17] und weitere Informationen zur aufgetretenen Ausnahmesituation. In den Fassadenmethoden werden die Exceptions gefiltert, um sicher zu stellen, daß Clients auch nur die definierten Fehler erhalten.
Auf eine Anpassung des Container-Loggings haben wir verzichtet. Da in unserem Projekt EJB-Clients ausschließlich auf der Middle-Tier zu Hause sind, haben wir diesen das Loggen überlassen. Da wir alle Exceptions schachteln, können wir auch auf dieser Ebene noch ausführliche Informationen zu aufgetretenen Fehlern protokollieren.[18]

Transaktionen und Sperren

Wie fast immer, ist es auch in „Verloren und Gefunden“ erforderlich, die technischen Sperren möglichst kurz zu halten. Daher wurden keine benutzerabhängigen Transaktionen zugelassen. Auch Massenverarbeitung innerhalb einer Transaktion mußte vermieden werden, sie wird in vielen kurzen technischen Transaktionen durchgeführt. Die fachlichen Anforderungen ermöglichen problemloses Wiederaufsetzen, falls ein Prozeß nach erfolgreicher Verarbeitung einer Teilmenge abbricht.

---

[14] [Beschorner] untersucht unterschiedliche Implementierungsstrategien auch unter Performance-Aspekten.
[15] „Aktiv“ sind solche Container, die spezialisierte, nicht-fachliche Methoden haben.
[16] Eine derartige Klasse könnte das Konzepts von [Knabe] im J2EE-Umfeld ergänzen.
[17] Die Alternative zu Fehlernummern sind spezifische Exception-Klassen. Diese bieten mehrere Vorteile: Der Client erkennt an der Methodensignatur, sofern diese auch die konkreten Exception-Klassen wiedergibt, welche Exceptions zu erwarten sind. Catch-Statements können differenziert eingesetzt werden. In den Konstruktoren der Exception-Klassen kann deutlich gemacht werden, welche spezifischen Zusatzinformationen bei einem bestimmten Exceptiontyp erforderlich sind.
[18] Im Unterschied zu [Knabe] reichen wir die komplette Fehlerinformationen nicht an den Endbenutzer durch.

In „Verloren und Gefunden“ wird ein optimistisches Sperrverfahren über Zeitstempel und in einigen fachlich motivierten Fällen auch über Attributabgleich realisiert.
Die Transaktionsprogrammierung erfolgt rein deklarativ über ContainerManaged Transactions. Die Idee, alle EntityBean-Methoden mit dem Transaktionsattribut "supports" zu versehen, und die Transaktionsteuerung vollständig in die SessionBeans zu verlagern, konnte nicht realisiert werden, da der von uns eingesetzte EJB-Container in der damaligen Version dieses Transaktionsattribut noch nicht unterstützt hat.
Daher mußte im Einzelfall untersucht werden, welche transaktionalen Abhängigkeiten zwischen den SessionBean- und EntityBean-Methoden bestehen. In der Regel sind bei allen EntityBean-Methoden außer den reinen Leseoperationen Transaktionen erforderlich.

Erfahrungen

Anwendungen der beschriebenen Art lassen sich mit J2EE problemlos erstellen. Manches erscheint bei Lektüre der Spezifikation einfacher als im wirklichen Leben. Wir haben zum Beispiel das Transaktionsthema zunächst vernachlässigt, der Container sollte ja die Transaktionen managen. Um die Deploymentdeskriptoren kümmert man sich gerne so nebenbei und oft erst, wenn in Tests Probleme auftreten, etwa Deadlocks bei Nebenläufigkeitstests. Auch deklarative Programmierung ist eben Programmierung und erfordert Planung und Systematik.
Nicht jedes angebotene Feature muß genutzt werden. Der Einsatz von ContainerManaged Persistence (auf dem Level der EJB-Spezifikation 1.1) hat keine Vorteile, wenn man sowieso BeanManaged Persistence nutzt und dafür einige generische Klassen schreibt. Zum direkten Ärgernis wurde in unserem Projekt die Verwendung von ContainerManaged Persistence bei der Umstellung des Containers von EJB 1.0 auf EJB 1.1. Noch mehr Freude dürfte bei der Umstellung auf EJB 2.0 aufkommen. Der Wert von Standards relativiert sich doch, wenn sie überholt sind, sobald die ersten Implementierungen vorliegen.
Natürlich läßt die EJB-Spezifikation auch noch Wünsche offen. Im Projekt „Verloren und Gefunden“ konnten feingranulare EntityBeans verwendet werden. Das dürfte nicht der Regelfall sein. Hilfreich wäre eine Differenzierung zwischen grob- und feingranularen Komponenten und zwischen Komponenten, die nur im Container leben und nicht zum Client exponiert werden, und solchen, die an den Containergrenzen exponiert werden. Die EJB-Spezifikation bringt hier mit den „Dependent Objects“ einen Ansatz, der sich in der Praxis noch bewähren muß.
Zur Persistenz gibt es derzeit nur vorläufige Lösungen. Im J2EE-Umfeld ist hier sehr viel in Bewegung, mit jeder neuen Version der EJB-Spezifikation kommen gravierende Änderungen. Nebenbei wird noch die JDO-Spezifikation[JDO] erstellt. Die OR-Mapping-Tools hinken der jeweils aktuellen Spezifikation nach. OO-Datenbanken versuchen sich als persistenter Middle-Tier-Object-Cache neu zu positionieren. Dem Anwendungsentwickler wird es hier wirklich leicht gemacht, dauerhaften Lösungen aus dem Weg zu gehen.
ExceptionHandling, generische Transportcontainer, Logging und Typkonvertierung sind Bereiche, die nicht in jedem Projekt neu implementiert werden müssen. Hier könnten vorgefertigte Komponenten oder Frameworks verwendet werden. Hilfreich sind auch Referenzarchitekturen und Muster wie die J2EE-Patterns.

Literatur

[Beschorner] Klaus Beschorner, Wolfgang Rosenstiel; Effiziente Datenübertragung in EJB-Systemen; in: [NetObj2000] S.14-29

[BluePattern] Sun Microsystems; Design Patterns; 2001; Extra zu [Blueprint] Http://java.sun.com/j2ee/blueprints/design_patterns/index.html

[Blueprint] Sun Microsystems; Designing Enterprise Applications with the Java 2 Platform, Enterprise Edition, Version 1.0; Final Release; 22.3.2000; Http://www.javasoft.com/j2ee/blueprints/

[EJB 1.0] Vlada Matena, Mark Hapner; Sun Microsystems; Enterprise JavaBeans Specification, Version 1.0; 21.3.1998

[EJB 1.1] Vlada Matena, Mark Hapner; Sun Microsystems; Enterprise JavaBeans Specification, v1.1; Final Release; 17.12.99

[EJB 2.0] Linda G. DeMichiel, L. Ümit Yalçinalp, Sanjeev Krishnan; Sun Microsystems; Enterprise JavaBeans TM Specification, Version 2.0; Proposed Final Draft; 23.10.2000

[Fowler] Martin Fowler; Analysemuster; 1999

[Gamma] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides; Entwurfsmuster; 1996

[Hahn] Wolfgang Hahn, Gerhard Albers; Datenbank-Zugriffsschichten; in: Peter Brössler, Johannes Siedersleben; Softwaretechnik; 2000; S.185-206

[J2EEPattern] Sun Microsystems; Sun Java Center J2EE Patterns, J2EE Patterns Catalog; Version 1.0 Beta; 2001; http://developer.java.sun.com/ developer/ Restricted/patterns/J2EEPatternsAtAGlance.html

[JDO] Sun Microsystems; Java DataObjects, JSR000012, Version 0.8; PublicReviewDraft; 2000; http://java.sun.com/aboutJava/communityprocess/ Review/jsr012/JDO_0_8.pdf

[Knabe] Christoph Knabe; Ein Framework zur Ausnahmebehandlung in mehrschichtigen Softwaresystemen; in: [NetObj2000] S.238-248

[NetObj2000] Net.ObjectDays-Forum (Hrsg.); NetObjectDays2000, Tagungsband; 2000