5

Typische Design-Beispiele aus dem JEE Umfeld

5 Typische Design-Beispiele aus dem JEE Umfeld

5.1 Verteilte Anwendungen

Um verschiedene Architekturen im Rahmen einer JEE-Anwendung abbilden zu können müssen zumindest bestimmte Services auch über ein Netzwerkprotokoll angeboten werden können. Genau genommen bedingte ja die Definition des Begriff "Service" genaue diese Verteilbarkeit. Rein "lokale Services" wären durch die Verwendung von Value Objects unnötig kompliziert.

Der JEE-konforme Applikationsserver stellt über die an Hand der Spezifikation geforderten Standard Inbound Connectors (http/https, RMI/IIOP, JMS, SOAP) bereits ein für die meisten Fälle genügendes Repertoire zur Verfügung. Werden weitere Protokolle benötigt so muss ein entsprechender Inbound Connector eventuell programmiert und zusätzlich installiert werden.

5.1.1 RMI/IIOP

5.1.1.1 Allgemeines

Stateless und Stateful SessionBeans sind von vorn herein für den Zugriff via RMI ausgelegt. Es soll hier aber nochmals explizit darauf hingewiesen werden, dass dies nicht das eigentliche Kriterium für den EJB-Einsatz ist. Dies sind die Dienste Transaktionssteuerung und Security. Eine Stateless SessionBean, die eine Service-Implementierung um diese Dienste erweitert wird in der Regel stets eine lokale Schnittstelle aufweise, die Remote-Schnittstellen sind optional!

Benötigt beispielsweise das Servlet einer Web-Anwendung einen über Session Facade realisierten Zugriff auf eine Business Object und die Web-Anwendung läuft direkt im Applikationsserver produzieren RMI-Aufrufe nur absolut unnötigen Overhead[1].

[1] Diese Overhead wird häufig durch Optimierungsmaßnahmen der Applikationsserver-Implementierungen kompensiert. Architektonisch sauber ist dies aber nicht.

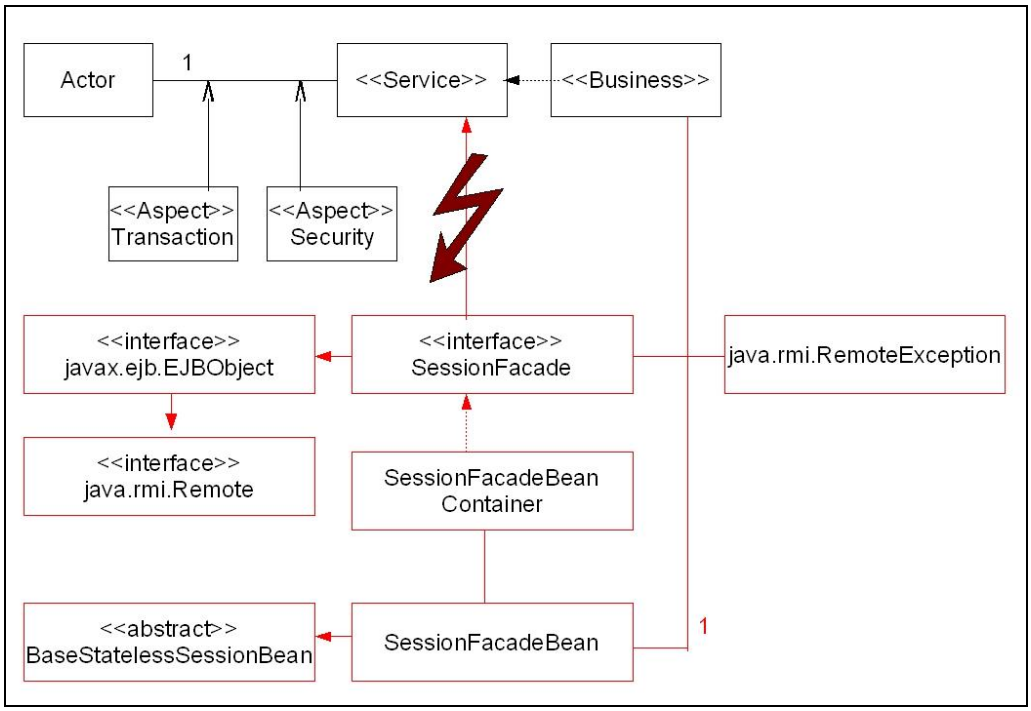
5.1.1.2 Session Facade und Business Delegate

5.1.1.2.1 Notwendigkeit

Soll der Zugriff auf ein Business Object via RMI ermöglicht werden, so werden Schnittstellen benötigt, die RMI-konform gehalten sind. Dies bedeutet:

- Erweitern der `java.rmi.Remote`-Schnittstelle
- Alle Methoden werfen die `java.rmi.RemoteException`

Diese auf Grund der Java-RMI-Spezifikation notwendigen Bedingungen führen auf den ersten Blick leider zu einer Unschönheit im Design: Die vom Client zu benutzenden Schnittstellen können nicht mehr die Service-Schnittstelle erweitern, da eine abgeleitete Schnittstelle nicht beginnen kann, Methoden der Super-Schnittstelle zusätzliche Exception-Typen werfen zu lassen. Dies ist ja nur für `RuntimeException`s zulässig und die `RemoteException` ist keine. Ein Vererbungshierarchie dieser Form ist somit unzulässig[2]:



[2] Um diese zu ermöglichen müsste die Service-Schnittstelle ja selbst bereits alle Methode mit der `RemoteException` ausgestattet haben. Dies ist aber definitiv keine Plattform-unabhängige Schnittstelle mehr!

Diese Unmöglichkeit hat im Rahmen der JEE-Programmierung zu einigem Aufruhr geführt, noch dazu der direkte Konkurrent, Microsofts .NET diese Problem nicht kennt (dort gibt es eh nur `RuntimeExceptions`). Häufig resultierte dies zu dem Vorwurf, dass das JEE Programmiermodell zu kompliziert sei.

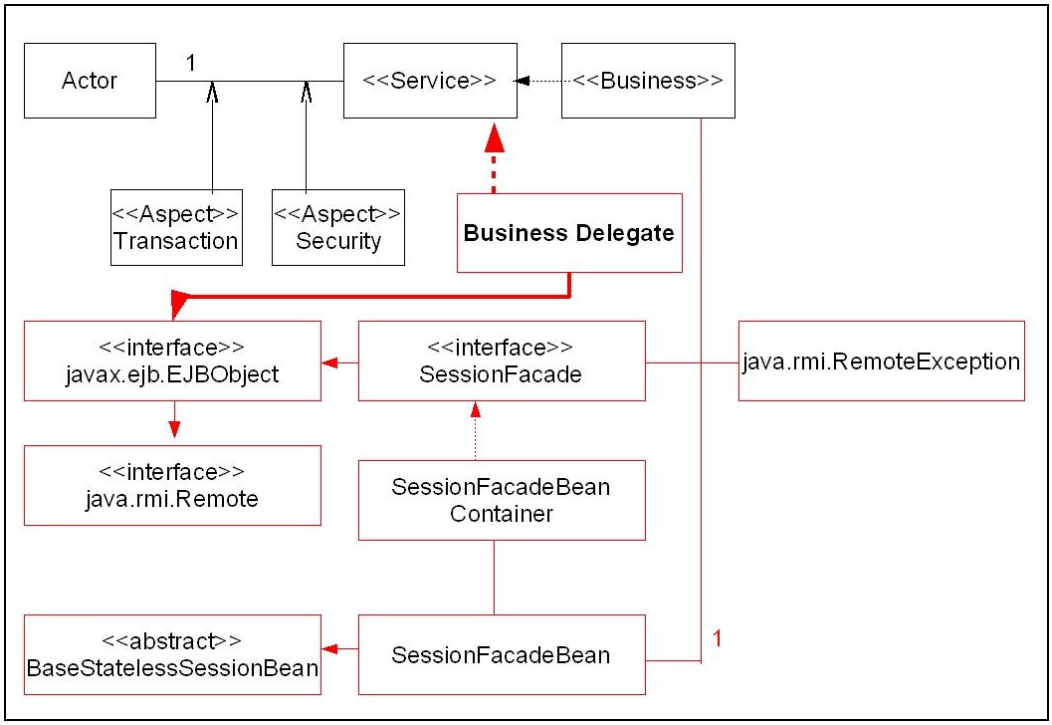
In der Praxis tritt dieses Problem jedoch nur sehr entschärft auf, da ein einfaches Design Pattern eine Lösung liefert. Und das ist der Business Delegate.

5.1.1.2.2 Der Business Delegate

Diese neu einzuführende Klasse tritt einfach als Adapter zwischen den `RemoteException`s werfenden EJB-Schnittstellen und der Plattform-unabhängigen Service-Schnittstelle auf. Allerdings sind seine Aufgaben noch deutlich klarer präzisierbar als eine einfache Adaptions-Logik:

- Der Business Delegate braucht eine Referenz auf die EJB und benutzt dazu den Service Locator.
- Er muss sich um die Behandlung der `RemoteException` kümmern. Die Standard-Implementierung eines Business Delegates wird daraus wohl eine `RuntimeException` machen. Damit verhält sich die JEE-Anwendung genau so wie eine .NET oder auch eine Spring-Anwendung. Das saubere am Business Delegate ist nun jedoch, dass auch komplexere Strategien problemlos konzipiert und eingesetzt werden können:
  - Aufruf eines Backup-Systems
  - Erzeugen einer Exception, die im Plattform-unabhängigen Modell bereits existiert.
  - Retry-Mechanismen
  - ...

Der Business Delegate wird auf folgende Art im Klassendiagramm integriert:



Der Business Delegate implementiert die Service-Schnittstelle und delegiert an die zugehörige Session Facade.

5.1.1.3 Paketstruktur

Die Einführung der zusätzlichen Schicht für die Verteilung über RMI erfordert eine Erweiterung der Paketstruktur:

- `<domain>.<service>`
- `<domain>.<service>.value`
- `<domain>.<service>.test`
- `<domain>.<service>.business`
- `<domain>.<service>.aspects`
- `<domain>.<service>.ejb`
- **`<domain>.<service>.network.ejb`**
  - Enthält die generierten Schnittstellen für den RMI-Zugriff sowie den Business Delegate

5.1.1.4 EJB 3

```
Die Idee des generischen Business Delegates wurde auch bei der
Konzeption der EJB 3 mit eingebracht. Diese Klasse sowie die
Session Facade wurden in das JEE Framework integriert, das
heißt: der JNDI-Lookup liefert hier direkt eine Implementierung
der Service-Schnittstelle!
```

5.1.1.5 Remote Facade, spezielle Business Delegates und Transfer Objects

Die eben eingeführten Hilfsklassen bzw. die EJB 3-Spezifikation erleichterten die Einführung einer Remoting-Schicht für RMI gewaltig. Im Endeffekt genügt in beiden Fällen die Einführung einer neuen Klasse, der Bean-Implementierung. Diese wird entweder mit XDoclet- oder EJB 3-Annotations ausgestattet. Die eigentliche Implementierung ist das Setzen der Dependencies und ein triviales Delegieren an das eigentliche Business Object[3].

Es stellt sich nun jedoch recht schnell die Frage, ob diese Trivial-Implementierung in allen Fällen genügt. Anders formuliert lautet die Frage: Wie gut ist die Service-Schnittstelle an die Besonderheiten des RMI-Protokolls in allen Anwendungssituationen angepasst?

Diese Frage mag nun etwas verblüffend erscheinen, da Service-Schnittstellen ja explizit die Forderung der Verteilbarkeit erfüllen müssen und damit sollte die Frage "eigentlich" überflüssig erscheinen.

Eine Betrachtung selbst der einfachsten Service-Schnittstelle ermöglicht nun aber bereits die Konstruktion von Szenarien, in denen beispielsweise die Signatur

```
public String next()
```

"verbessert" werden kann.

Das erste Szenarium fokussiert auf eine extrem schlechte Netzwerkverbindung, in der jedes eingesparte zu übertragende Byte eine Verbesserung bedingt. Ist bekannt, dass der erzeugte Schlüssel aus ASCII-Zeichen besteht, so ist eine Übertragung von Unicode-Zeichen, in denen das höherwertige Byte stets Null ist, überflüssig:

```
public byte[] next()
```

[3] Prinzipiell könnte bei EJB 3 das Business Object selber bereits die Annotations für die Session Facade enthalten. Diese sind jedoch (leider) teilweise im javax.ejb-Paket gelandet, so dass damit die Forderung der Plattform-unabhängigen Implementierung verletzt wird. Bei einer sauberen Modellierung verlangt also auch EJB 3 die Einführung einer eigenen Klasse.

Falls der Schlüssel intern durch ein Zahl repräsentiert wird wäre die nächste Signatur noch "besser" geeignet:

```
public int next()
```

Das zweite Szenarium ist ein anderer Service, der für seine internen Sequenzen eine gewisse Menge an Schlüsseln anfordern muss[4]. So lange der Aufruf des `KeyGenerator`-Services lokal erfolgt ist die folgende Schleife vollkommen unkritisch:

```
for (int i =0; i < 100; i++){
 String key = keygenerator.next();
//...
}
```

Wird nun aber auf Grund einer Architektur-Änderung der Remote-Zugriff notwendig benötigt die obige Schleife ausschließlich auf Grund der Netzwerkverbindung etwa 100 Millisekunden zusätzlich[5]. Nun ist eine Signatur der Form

```
public String[] next(int blockSize);
```

wesentlich performanter zu benutzen.

Alle diese neuen Signaturen sind ganz spezielle Optimierungsmaßnahmen, die natürlich nur in ganz speziellen Situationen hilfreich oder möglich sind. Deshalb ist eine Ergänzung der Service-Schnittstelle um diese Methoden nicht immer richtig bzw. erzeugt unerwünschte Nebeneffekte:

- Alle Business Objekte müssen die neuen Signaturen unterstützen, selbst wenn sie nur in unkritischen Bereichen der Anwendung eingesetzt werden.
- Der Programmierer des Service-Benutzers sieht innerhalb der Service-Schnittstelle Methoden, von denen er nicht unbedingt wissen kann, welche für ihn nun die "richtige" ist. Dokumentationsaufwand und Fehleranfälligkeit des Services steigen damit ungemein.

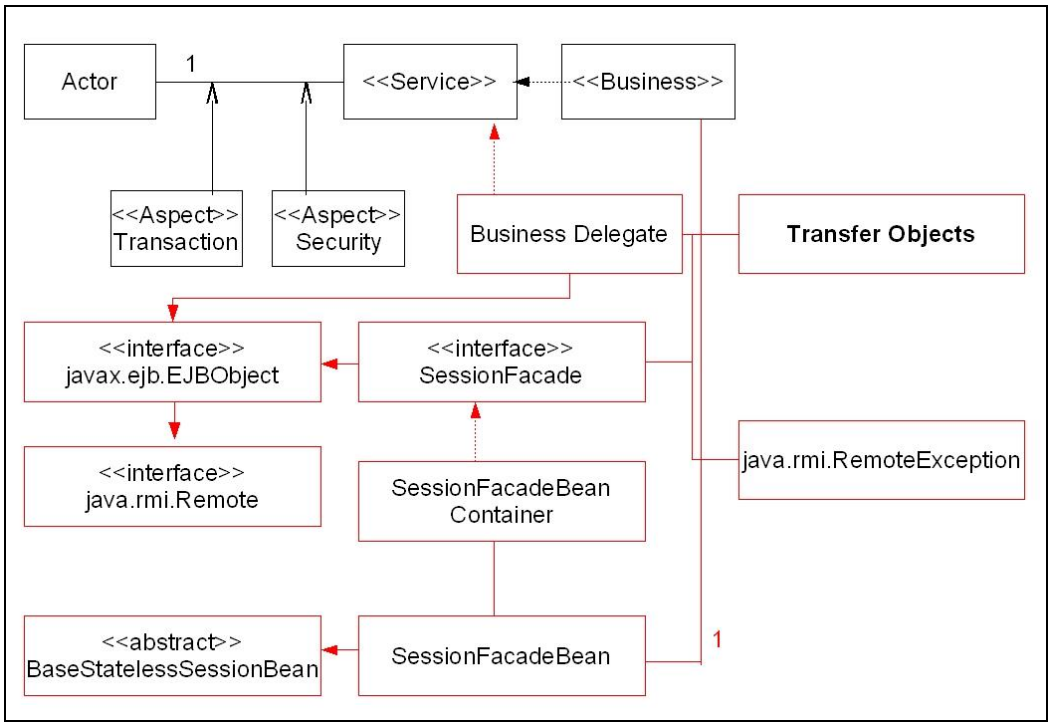
Solche Anpassungen können aber völlig problemlos in das vorhandene Design integriert werden ohne die Service-Schnittstellen ändern zu müssen. Es spricht nichts dagegen, diese Signaturen, die ja aus-

[4] Beispielsweise das Einfügen eines komplexen Objektbaums in eine Datenbank mit einem neuen Master-Eintrag und vielen Detail-Einträgen.

[5] Ein RMI-Methodenaufruf benötigt bei typischen Firmen-internen Netzwerken etwa eine Millisekunde.

schließlich in bestimmten Netzwerk-Architekturen sinnvoll sind, dadurch zu realisieren, dass nur die Session Facade diese Methoden anbietet[6]!

Damit ist die Facaden-Logik nicht mehr trivial, sondern sie passt die vom Service gelieferten Value Objekte um in Objekte, die "besser" von einer Schicht der Anwendung in die nächste übertragen werden können: Die (Data) Transfer Objects.



Im folgenden Beispiel wird die Übertragung eines ganzen Blocks von Schlüsseln realisiert. Das Transfer Object ist in diesem Falle ein String-Array.

[6] Diese Argumentation ist natürlich nicht immer stichhaltig bzw. vernünftig. Selbstverständlich gibt es auch Situationen, in denen der Client ganz gezielt bestimmte neue Signaturen benutzen muss. Dann ist es an der Zeit entweder den Service durch weitere Signaturen zu ergänzen oder aber einen neuen Service zu konzipieren, der den ursprünglichen erweitert.

5.1.1.6 Erweiterungen

Die folgenden Erweiterungen sind für komplexere Anwendungen zu empfehlen.

5.1.1.6.1 Local und Remote Session Facade

Nachdem konzeptuell jeder Client, der eine besondere Anpassung der Service-Signatur benötigt, eine eigene spezielle Facade-Signatur benötigt wird die Facade-Implementierung bei größerer Anzahl von Clients naturgemäß immer mächtiger und unübersichtlicher. Dann ist die logisch bereits vorhandene Trennung der Aufgabenverteilung in zwei verschiedene SessionBeans zu erwägen. Die **Local Session Facade** enthält die der Bean zugeordnete fachliche Logik:

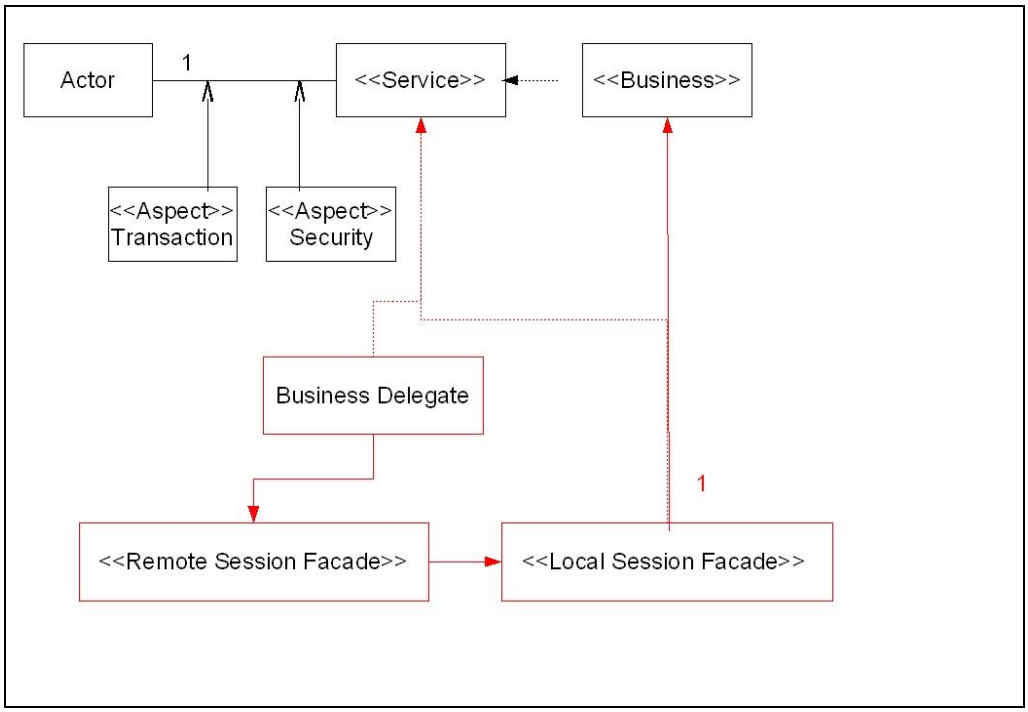
- Transaktionssteuerung
- Security
- Analysieren und Setzen der Dependencies des Business Objects

Eine oder mehrere Remote Session Facaden passen die Signaturen an bzw. erzeugen aus den von der Business Implementierung gelieferten Value Objects die angepassten Transfer Objects.

Um keine Redundanzen entstehen zu lassen wird die Remote Session Facade an die Local Session Facade delegieren.

Um die Klassendiagramme etwas übersichtlicher gestalten zu können werden im Folgenden drei neue UML-Stereotype eingeführt:

- `<<Local SessionFacade>>` ist eine Stateless SessionBean mit lokalen Schnittstellen.
- `<<Remote SessionFacade>>` ist eine Stateless SessionBean mit ausschließlich Remote-Schnittstellen.
- `<<Business Delegate>>` ist der spezielle Business Delegate.



5.1.1.6.2 Value Objects und Transfer Objects

Der Katalog der JEE Patterns trifft keine Unterscheidung zwischen Value und Transfer Objects. Früher wurde ausschließlich Value Object, nun Transfer Object benutzt. Der Unterschied zwischen diesen beiden Pattern tritt auch erst bei den eingeführten Optimierungsmaßnahmen auf, vorher gilt tatsächlich Value Object = Transfer Object.

Sobald jedoch eine Remote Session Facade spezielle Objekttypen liefert ergibt sich eine Trennung:

- Value Objects sind Bestandteil der Service-Definition. Sie definieren eine spezielle Client-Sicht auf einen Datenbestand. Value Objects sind auf Grund dieser Definition nicht notwendig serialisierbar.
- Transfer Objects repräsentieren einen Datenbestand, der zu einem Client über ein Netzwerk transferiert werden muss. Transfer Objekte sind damit in irgendeiner Form in einem serialisierbaren Datenstrom umzuwandeln.

Ein Beispiel für diesen Unterschied liefert der `BooksService`. Die speziellen Client-Sichten `BookValue` und `BookListValue` treten in der Signatur der Service-Schnittstelle auf. Unter anderem gibt es die Methode

```
BookValue updateBook(BookValue bookDetailValue) throws Book-
Exception;
```

Diese Methode sendet geänderte Buch-Informationen zum Server, der dann in einer realen Implementierung die geänderten Werte in einer Datenbank abspeichern wird. Beginnt man aber, diese Sequenzen umzusetzen wird schnell ein Problem erkennbar: Woher "weiß" der Server denn, welche Werte und Attribute überhaupt geändert wurden? Der Client-Anwendung sind diese Informationen bekannt, aber wie kann diese Zusatz-Information sauber transferiert werden[7]? Es ist hier sinnvoll, nur die geänderten Werte zu übertragen. Für flache Value Objekte genügt hierfür eine `Map`.

Damit der Business Delegate erkennen kann, welche Attribute geändert wurden, wird eine Subklasse von `BookValue` geschrieben, die sich Zustandsänderungen merken kann[8].Der Business Delegate sendet nur die `Map` mit den "Dirty Values": Die spezielle Remote Session Facade empfängt und schreibt ausschließlich die geänderten Werte:

5.1.1.6.3 Value Object Assembler und Data Transfer Object Factory

Diese beiden Pattern sind ebenfalls Bestandteil des JEE Katalogs, wenn auch in jeweils unterschiedlichen Versionen. Der ursprüngliche Value Object Assembler diente dazu, aus Server-seitigen Objekten Objekte zu erzeugen, die zum Client gesendet werden können. Diese Rolle übernimmt nun die Data Transfer Object Factory.

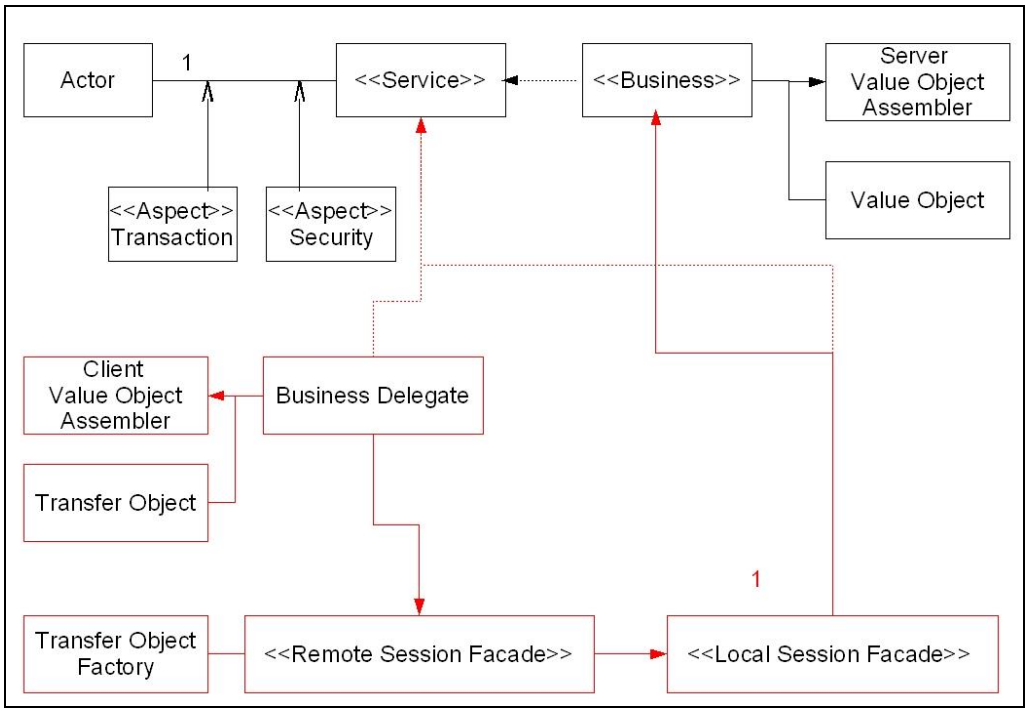
Die ursprüngliche Aufgabe ist nun dem Business Object zugeordnet. Dieses kennt und benutzt in komplexeren Anwendungen mehrere Services, um das benötigte Value Object zu erzeugen bzw. eben "aufzusammeln". Auch hier enthält die Buchanwendung ein Beispiel: Das `available`-Attribut des `BookValue` wird vom `StockService` bereitgestellt.

Der Value Object Assembler wird nun jedoch auch auf Client-Seite benötigt. Hier muss der Business Delegate aus dem Transfer Object die Value Objects erzeugen.

---

[7] Natürlich könnte das gesendete `BookValue` nur die Attribute belegt bekommen, die geändert wurden. Alle anderen stehen auf symbolischen Werten.

[8] Auf Grund dieser Logik ist dies im strengen Sinn kein Value Object mehr.



Diese Entwurfsmuster werden in einfachen Szenarien nicht durch eigenen Klassen realisiert. Sie genügen dem typische Refactoring-Vorgang Sequenz -> private Methode -> Hilfsklasse.

5.1.1.6.4 Value und Transfer Documents

Die Entwurfsmuster Value Object bzw. Transfer Object fixieren auf eine Objektorientierte Modellierung der Schnittstellen mit typisierten Objekten. Dies ist nicht immer richtig: Im Rahmen einer Dokumentenorientierten Programmierung wird statt eines Typs ein Dokument verwendet, das einem, falls erwünscht validierbarem Schema genügt. Die Signatur lautet dann:

```
public Document execute (Document document);
```

`Document` kann, muss aber nicht ein XML-Dokument sein. Auch eine Zeichenkette, ein `javax.servlet.HttpServletRequest` oder eine `javax.jms.Message` sind geeignete Dokument-Typen.

In solchen Fällen spricht man dann natürlich sinnvoller von Value Document bzw. Transfer Document.

5.1.1.6.5 Value List Handler

Eine weitere typische Optimierungsmaßnahme ist die Einführung einer aus der Datenbank-Programmierung bekannten Fetch Size für RMI-Methoden. Bei einer Datenbank-Abfrage mit potenziell vielen Treffern enthält das gelieferte `ResultSet` nicht bereits die komplette Ergeb-

nismenge, sondern einen Teilausschnitt sowie einen Cursor auf die innerhalb der Datenbank gehaltenen Treffermenge.

So ein Verhalten kann auch bei Service-Schnittstellen sinnvoll sein. Dann spricht man vom Design Pattern Value List Handler. Dem `ResultSet` entspricht hierbei der Rückgabetyp `java.util.Collection` oder eine ihrer Sub-Schnittstellen.

Ist die Einführung eines Value List Handlers bereits bei der Konzeption der Service-Schnittstelle klar, so könnten die kritischen Suchmethoden auf folgende Art und Weise geschrieben werden:

```
public int prepareSearch ();
public Collection next(int fetchSize);
public void discard();
```

`prepareSearch` stößt auf Server-Seite die Suche an und liefert die Anzahl der Treffer, `next` liefert dann jeweils den nächsten Block von Ergebnissen. Dieser Ansatz ist jedoch kritisch: Der eben definierte Service ist nicht mehr Zustandslos, da die Ergebnismenge auf dem Server irgendwie gehalten werden muss. Dies erfolgt entweder dadurch, dass der Server pro Client-Anfrage die gesamte Treffermenge in seinen Hauptspeicher einlädt oder aber das zugehörige `ResultSet` bleibt zwischen den einzelnen Aufrufen geöffnet. So eine Service-Schnittstelle kann deshalb in dieser Form nicht Bestandteil eines Platform Independent Models sein.

Die nächste Service-Schnittstelle ist Zustandslos:

```
public int count ();
public Collection next(int start, int fetchSize);
public void discard();
```

Allerdings ist die Implementierung der `next`-Methode nur extrem aufwändig zu realisieren. Mit hoher Wahrscheinlichkeit wird diese “Optimierung” dazu führen, dass die Datenbank die Abfrage mit vielen Treffern nun sogar mehrfach komplett ausführen muss.

Die Ansätze auf der Ebene der Service-Schnittstelle sind somit alle nicht sonderlich befriedigend. Verlagert man das Problem in das Platform Specific Model für die JEE so ist die erste Alternative durch die Einführung einer Stateful SessionBean durchaus machbar. Nachdem der Service aber weiterhin seine Zustandslose Formulierung bekommen kann wird in diesem Beispiel der Value List Handler wiederum durch das Zusammenspiel Business Delegate – Session Facade realisiert.

Die zugehörige Remote Session Facade[9] ist nun zustandsbehaftet. Nachdem die Service-Schnittstelle dem Client keine Iterationslogik anbieten kann liefert der Business Delegate eine spezielle Collection-Implementierung. Diese hält sich die geladenen Daten in einer internen Liste und weiß, wie Daten nachgeladen werden können.

Der Business Delegate erzeugt nun bei der entsprechenden Suche eine Instanz dieser Collection und setzt sich als Attribut, um der Collection das Nachladen zu ermöglichen.

Der Client bemerkt von der Existenz des Value List Handlers nichts: Er bekommt in der ursprünglichen Form eine vollständig gefüllte Collection, danach eine Collection mit Nachlade-Logik.

Die Optimierung mit Hilfe des Value List Handlers funktioniert nur mit einigen Nebenbedingungen:

- Der Server kann Client-behafteten Zustand im Hauptspeicher halten. Dies ist in der JEE bei Stateful SessionBeans und Servlets der Fall.
- Das eigentliche Laden der großen Datenmengen von der Datenbank ist nicht das eigentliche Problem, sondern die Übertragung zu einem entfernten Client. Liegt das Performance-Problem in der Datenbank so bringt die Einführung eines Value List Handlers kaum Vorteile.
- Die Sequenzen des Clients können bereits mit der Nicht-vollständig gefüllten Collection arbeiten. Dies ist besonders klar bei Clients mit einer Benutzerschnittstelle (GUI-Anwendung oder Web Frontend). Soll beispielsweise eine Swing-Tabelle mit den Ergebnissen befüllt werden, so genügt dem Tabellenmodell die Information „Anzahl der Treffer“ sowie die ersten in der Tabelle dargestellten Daten. Völlig ungeeignet ist der Value List Handler für eine Verarbeitung, die auf der gesamten Treffermenge arbeiten muss, z.B. Reporting. Hier führt die Einführung dieses Patterns nur zu einer Erhöhung der Anzahl der Remote Methodenaufrufe und ist damit Performance-mindernd.

5.1.2 Die Message Facade

5.1.2.1 Vorteile des JMS-Protokolls

Der Zugriff auf die Local Session Facade über eine vorgeschaltete Remote Session Facade ist nicht immer notwendig. Alternativ oder ergänzend kann statt des RMI-Protokolls auch JMS verwendet werden. Die Entscheidung darüber ist Bestandteil der Anwendungs-Architektur. Kriterien dafür lauten:

- Ausfallsicherheit und Nachvollziehbarkeit: Hier ist ein JMS-System auf Grund des Message Puffers der JMS-Destinations häufig die geeignete Wahl.

[9] Für lokale Zugriffe ist der Value List Handler unsinnig.

- Flexibilität: Die Session Facaden werden über typisierte entfernte Referenzen angesprochen. Ein flexibles Ergänzen durch weitere Funktionalitäten verlangt relativ viel Programmieraufwand[10].
- JMS-Provider bieten häufig APIs für verschiedene Programmiersprachen. Deshalb kann Messaging auch für die Interoperabilität eingesetzt werden.
- Asynchrone Aufrufe oder Rückgaben via Callback-Aufrufen sind integraler Bestandteil von JMS.

5.1.2.2 Layer Supertype und Basisklasse für den Business Delegate

Die bisher vorhandenen Service-Definitionen sind Schnittstellen-orientiert ausgelegt worden. Um die Umstellung des Netzwerkprotokolls auf JMS zu ermöglichen müssen spezielle Klassen erzeugt werden:

1. Auf Client-Seite muss ein spezieller Business Delegate alle Methodenaufrufe in eine Message umwandeln und zum Server senden. Auf einer speziellen Destination wartet der Business Delegate dann auf die Antwort-Message des Servers und extrahiert daraus das Ergebnis. Dies ist entweder das normale Resultat oder eine Exception.
2. Auf Server-seite muss eine `javax.jms.MessageListener` oder eine `javax.ejb.MessageDrivenBean` die empfangene Message analysieren, den Aufruf der Session Facade durchführen und das Ergebnis zum Client zurücksenden.

Diese Aufgaben sind relativ komplex, können jedoch in geeignete Hilfsklassen ausgelagert werden.

5.1.2.3 Der Service to Worker

5.1.2.3.1 Einführung

Die oben am Beispiel des Schlüsselgenerators demonstrierte Vorgehensweise ist für komplexere Service-Schnittstellen immer noch aufwändig zu programmieren. Der Programmierer muss sich ein geeignetes Protokoll ausdenken und innerhalb der Server-Implementierung müssen die Messages entsprechend analysiert werden.

Andererseits lässt sich die Aufgabenstellung recht allgemein formulieren:

- Über das Netzwerk wird ein Aufruf empfangen. Darin stehen Informationen, die eindeutig eine Service-Methode kennzeichnen. Zusätzlich sind eventuell notwendige Parameter enthalten.

[10] In einem späteren Abschnitt wird diese Flexibilität jedoch auch für Session Facaden eingeführt.

- Die Server-Implementierung analysiert den Request und ruft die Service-Methode auf.
- Das Ergebnis des Aufrufs kann verschiedene Zustände aufweisen ("OK", fachliche Ausnahme, Laufzeitfehler) und wird vom Server in irgendeiner Form als Antwort über das Netz gesendet.

Diese Sequenzen lassen sich sehr schön mit dem Design Pattern „Command“ umsetzen. Allerdings sind eben noch einige Erweiterungen, insbesondere die Berücksichtigung der Netzwerk-Kommunikation notwendig. Deshalb wird dieses Pattern in diesem Zusammenhang als „Service to Worker“ bezeichnet[11].

5.1.2.3.2 Das Command-Framework

Das Command-Framework besteht aus drei Hilfsklassen:

- Ein `Result` hat einen internen Zustand und hält ein Ergebnis-Objekt. Ergebnis-Objekte müssen über ein Netzwerk transferiert werden können:
- Ein `Command` kann ausgeführt werden. Die abstrakte Basisklasse wandelt die Ergebnisse in `Result`-Objekte um.
- Eine `CommandMap` hält eine `Map` von `Command`-Implementierungen und stellt diese zur Verfügung.

5.1.2.4 Message Facade und Klassendiagramm

Die Message Facade übernimmt im Klassendiagramm eine der Remote Session Facade entsprechenden Rolle. Sie wird in der Regel an die Local Session Facade delegieren, um die darin konfigurierten Dienste nicht redundant konfigurieren zu müssen. Prinzipiell ist es aber auch möglich, dass die Message Facade selber die Transaktionssteuerung übernimmt, da auch eine MessageDrivenBean den Zugriff auf den Transaktionsmanager des Applikationsservers hat. Dies entspricht einer Architektur, in der die Verteilung ausschließlich über JMS realisiert werden soll.

[11] Service to Worker ist aus der Historie her eigentlich ein JEE Pattern der Präsentationsschicht und wird häufig im Zusammenhang mit Web-Anwendungen benutzt. Dieses Entwurfsmuster lässt sich aber vollkommen problemlos für die oben geschilderte allgemeine Problemstellung nutzen.

5.1.2.5 Dokumenten-orientierte Services

Das Command-Pattern erlaubt es auf sehr einfache **Weise**, Services mit neuer Funktionalität zu erweitern:

- Die Einführung einer neuen Methode erfordert keine Änderung der Signatur der Message Facade. Dies wäre ja auch unmöglich, da ausschließlich eine `onMessage`-Methode vorhanden ist. Es muss nur in der `CommandMap` ein neues Kommando registriert werden. Dies ist aber bei geeigneter Programmierung sogar zur Laufzeit möglich.
- Ein Kommando kann von einer ganzen Kette verarbeitender Instanzen behandelt werden, von denen jede einen bestimmten Teil der Fachlogik modular getrennt ausführt. Diese Idee ist das im Zusammenhang mit Command eingeführte Entwurfsmuster „Chain of Command“. Ebenso wie das Hinzufügen neuer `Command`-Instanzen können somit auch ganze Verarbeitungsschritte dynamisch zur Laufzeit hinzugefügt oder entfernt werden.
- Ein `Command`-Objekt kann sehr einfach an komplett andere Services weitergeleitet werden. Es ist an dieser Stelle nur ein Umkopieren/Ergänzen der Parameter-Liste notwendig[12].

Alle diese Argumente sind sogar unabhängig von JMS und lassen sich zu dem Begriff „Dokumenten-orientierte Schnittstelle“ verallgemeinern. Kennzeichen einer Dokumenten-orientierten Schnittstelle ist, dass statt typisierter Methoden-Signaturen nur eine einzige Signatur angeboten wird:

```
public ResultDocument execute(ParameterDocument)
```

`ResultDocument` und `ParameterDocument` haben keinen festen Typ und damit keine feste Struktur sondern enthalten flexibel Informationen. Eine Server-seitige Verarbeitung nimmt sich aus dem `ParameterDocument` die Informationen, die für seine Verarbeitung relevant sind heraus und ignoriert den Rest. Für eine Weiterverarbeitung in einer „Chain of Command“ kann die Verarbeitung jederzeit weitere Informationen ergänzen, die dann in einer Folgeverarbeitung benötigt werden.

Wird als Dokument-Typ XML verwendet, so kann durch die Vorgabe eines XML-Schemas zumindest zur Laufzeit eine Validierung erfolgen. Dem Java-Typ, das der Compiler prüft, entspricht hier dann das XML-Schema, das der Parser validiert.

[12] Wäre in der Basisklasse statt einer Liste eine Map verwendet worden könnte diese ohne Kopie durch weitere Parameter ergänzt werden, was für dieses Szenarium sicherlich vorteilhaft ist.

5.1.2.6 Dokumenten-orientierte Remote Facade

Selbstverständlich kann eine Dokumenten-orientierte Fassade auch mit einer Session Bean realisiert werden. Dazu wird einfach die obige Signatur

```
public ResultDocument execute(ParameterDocument)
```

übernommen. Dies kann bereits im Platform Independent Model geschehen oder aber in der Fassaden-Signatur des Platform Specific Models.

Im ersteren Fall ist die Service-Schnittstelle selbst bereits Dokumenten-orientiert modelliert. Im zweiten Fall sind die Dokumente als Transfer Documents aufzufassen. Der zweite Fall kann eine zentrale Session Facade alle ankommenden Anfragen zentral verwalten. Dies entspricht dem Design Pattern EJB Command[13]. Die sich daraus ergebenden Vorteile liegen auf der Hand:

- Konsistentes Programmiermodell für Message Facade und Remote Facade
- Zentraler Einstiegspunkt für das dynamische Einhängen weiterer Dienste
- Dynamische Erweiterungen/Änderungen von Services ohne erneutes Erzeugen der Fassaden. Es genügt wie bei der Message Facade die Verwaltung von `Command`s in der `CommandMap`.

5.1.3 Die Http Facade

Eine Verteilung der Anwendung ist auch über das http-Protokoll möglich[14]. Hier dient der `javax.servlet.http.HttpServletRequest` als Transfer Document auf und wird von einem Servlet entgegen genommen.

Durch die Einführung des Service-to-Worker ist eine Realisierung sehr einfach: Die http Facade analysiert den Request und delegiert genau so wie die Message Facade an diesen weiter. Der Business Delegate baut einen geeigneten URL-Request auf und liest aus dem `InputStream` das Resultat.

[13] Siehe http://www.theserverside.com

[14] Dies bedeutet natürlich nicht, dass es sich hierbei um eine Web Anwendung handelt. Es werden ausschließlich die Service-Aufrufe über http getunnelt, der Client bleibt beispielsweise eine Swing-Anwendung.

5.1.4 Die SOAP Facade

Service-Schnittstellen sind auf Grund ihrer Definition in der Regel bereits kompatibel zur Web Service Spezifikation. Das Ansprechen eines Services über das SOAP-Protokoll erfordert deshalb nur sehr wenig Aufwand. Die im Rahmen der Web Services-Spezifikation enthaltenen Annotations ermöglichen es, beispielsweise eine vorhandene Session Bean sofort zu einem Web Service zu machen.

Der Zugriff auf die Facade erfolgt dann wie Üblich über einen Business Delegate. Dieser holt sich in diesem Fall die Referenz auf den Web Service über RPC. Selbstverständlich kann die SOAPFacade auch die Aufgaben einer echten Remote Facade übernehmen, also beispielsweise spezielle Transfer Objekte erzeugen (hier wahrscheinlich XML-Dokumente), Signaturen ändern etc. Insbesondere in Interoperablen Architekturen, in denen beispielsweise ein Perl- oder .NET-Client den Web Service aufrufen will ist es häufig notwendig, aus komplexen Objektbäumen einfachere Dokumente zu erzeugen.

5.2 Zugriff auf Entities

5.2.1 Lokale Services und Entities

Die Definition eines Services enthält explizit die Verteilbarkeit der Implementierung in verschiedenen Architekturen. Eine zentrale Forderung daraus war die Einführung der Call-by-Value-Semantik bzw. des Design Patterns Value Object.

Wird unter Beibehaltung der restlichen Anforderungen auf die Verteilbarkeit der Services verzichtet, so wird im Folgenden der Begriff "Lokaler Service" bzw. das UML-Stereotyp `<<Local Service>>` verwendet.

Für lokale Services ist die Einführung von Value Objekten überflüssig, in den meisten Fällen sogar falsch:

- Zusätzlicher Aufwand durch Umkopieren von Daten.
- Komplexeres Anwendungsdesign durch die Verwendung von Datenkopien statt echter Referenzen.
- Verschlechterung der Ausführungsgeschwindigkeit der Anwendung, da Value Objekte zur Erfüllung der Call-by-Value-Semantik auch bei lokalen Aufrufen stets kopiert werden müssen[15].

Lokale Services liefern damit als Ergebnis und bekommen als Parameter Referenzen.

Um den Unterschied zwischen lokalen und verteilbaren Services klar zu machen hier die Service-Implementierung eines Warenkorbs (`Basket`)

Verteilbarer Service[16]:

```
Map<Item> getBasket(String id)
void updateBasket(String id, Map<Item> basket)
```

[15] Einige Applikationsserver optimieren dies dadurch, dass auf diese Kopie verzichtet wird. Die widerspricht jedoch der strengen Definition von Value Objekten und kann zu Fehlern in der Ausführung führen, wenn beispielsweise die Server-Implementierung eine Änderung des Zustandes eines Parameters durchführt. Der Aufrufer der Server-Methode darf in der Call-by-Value-Semantik in seinem lokalen Objekt diese Änderung nicht bemerken, da er ja erwartet, dass eine Kopie an den Server gesendet wurde. Wird dieser jedoch lokal angesprochen und der Applikationsserver führt die oben angesprochene Optimierung durch, wird die Änderung auf den Aufrufer durchschlagen. Dieses Problem kann beispielsweise bei Parametern vom Typ `Collection` oder `Map` auftreten und verlangt entsprechende Programmiervorgaben.

[16] Man beachte die Zustands-lose Formulierung durch Verwendung einer Id.

Lokaler Service:

```
Map<Item> getBasket(String id)
```

Beim lokalen Service ist ein Update auf den `Basket` nicht unbedingt notwendig, da der Aufrufer über die Referenz das "echte" Objekt verändern wird.

Die Einführung eines lokalen Services ist im Rahmen einer Objektorientierten Programmierung nichts Neues: Jedes API, das über eine Trennung von Schnittstelle und Implementierung verfügt, ist ein lokaler Service.

Eine besondere Form der lokalen Services, die eine besondere Betrachtung erfordern. sind diejenigen, die den Zugriff auf persistente Daten bereitstellen. Die Objekt-Repräsentation dieser Daten ist eine besondere Referenz, eine "Entity". Der zugehörige lokale Service wird "EntityAccess" genannt. Die zugehörigen UML-Stereotype lauten

- `<<EntityAccess>>`
- `<<Entity>>`

Der Zustand einer Entity muss folgenden Forderungen genügen:

- Er muss konsistent mit der persistenten Datenhaltung, in der Regel einer Datenbank, sein.
- Die Konsistenz des Objektbaums muss auch über die verschiedenen Clients hinweg erhalten werden. So muss beispielsweise in einem Server-Cluster eine Änderung eines solchen Objekts in einem Cluster-Knoten neben der Datenbank auch allen anderen Knoten mitgeteilt werden.
- Falls Änderungen atomar in einer Transaktion erfolgen sollen muss der Objektbaum gesperrt werden können.

Die Umsetzung dieser Anforderungen erfolgt in den meisten Fällen durch Datenbank-Mittel. Andere Varianten wie ein Cluster-übergreifender Cache mit Transaktionsunterstützung sind zwar ebenso möglich, funktionieren in der Regel aber nur in bestimmten Laufzeitumgebungen, die dann meistens auch noch den Einsatz ganz bestimmter Produkte erfordern. So bieten JEE-Applikationsserver für die EntityBeans ein Caching-System an, das allerdings nur eine Cluster funktioniert, der aus einem Server-Produkt besteht[17]. Ein JEE-unabhängiges Produkt mit Transaktionsunterstützung ist der JBoss-TreeCache. Der Einsatz dieser Caches wird in heterogenen Umgebungen, in denen Datenänderung am Cache vorbei direkt in die Datenbank erfolgen können,

[17] Der Weblogic-Cache funktioniert nur mit anderen Weblogic-Instanzen und nicht interoperabel mit Websphere oder JBoss.

jedoch deutlich erschwert wenn nicht sogar unmöglich gemacht: Der Cache muss sich entweder vor jedem Zugriff mit der Datenbank synchronisieren, was dann aber das Caching insgesamt fragwürdig macht, oder aber die Datenbank müsste Änderungen aktiv durch einen Trigger-Mechanismus an den Cache weiterleiten. Dann muss aber am Ende jeder Datenbanktransaktion eine Netzwerk-Methodenaufruf erfolgen, was den Transaktionsdurchsatz der Datenbank deutlich verschlechtern wird.

Deshalb erfolgt, wie erwähnt, in den allermeisten Fällen die Umsetzung des Entity-Verhaltens über zentrale Datenbank-Mechanismen:

- Das Caching von Ergebnis-Mengen erfolgt in der Datenbank. Nur bei sich selten oder optimal: gar nicht ändernden Daten ist ein Zwischenspeichern innerhalb des Servers sinnvoll.
- Das Sperren der Daten erfolgt über Datenbank-Transaktionen.

Die Implementierung einer Entity kann mit den verschiedensten Strategien erfolgen:

- Die Entity benutzt intern ein aktualisierbares `ResultSet`. Jede Änderung des Objektzustandes wird in das `ResultSet` eingetragen[18]. Der Aufrufer der Entity sieht an dessen Signatur keinerlei Besonderheiten, die auf das Persistenz-Verhalten hindeuten.
- Die Entity referenziert eine Hilfsklasse, die die Anbindung an die Datenbank durchführt. Die Entity hat damit selber eine `save`-Methode, die aufgerufen werden muss.
- Ein Persistenz-Manager ist in der Lage, Entity-Objekte in die Datenbank zu speichern. Die Entity muss somit als Parameter einer `save`-Methode des Managers übergeben werden.

Die Frage ist nun: Welche Idee der Umsetzung ist denn die "richtige" oder "beste"? Die Antwort ist leider nicht eindeutig und so darf es nicht überraschen, dass verschiedene Framework-Ansätze existieren, die jeweils eine bestimmte Umsetzungsstrategie favorisieren:

- Die EntityBeans der EJB 2.1 sind Live-Objekte. Dieses Verhalten wird aber dadurch erkauft, dass eine Bean-Instanz intern an den EJB-Container des Applikationsservers gekoppelt ist. Eine Verwendung dieser Instanz außerhalb des Applikationsservers ist somit nicht möglich.

[18] Natürlich sind auch andere Techniken möglich.

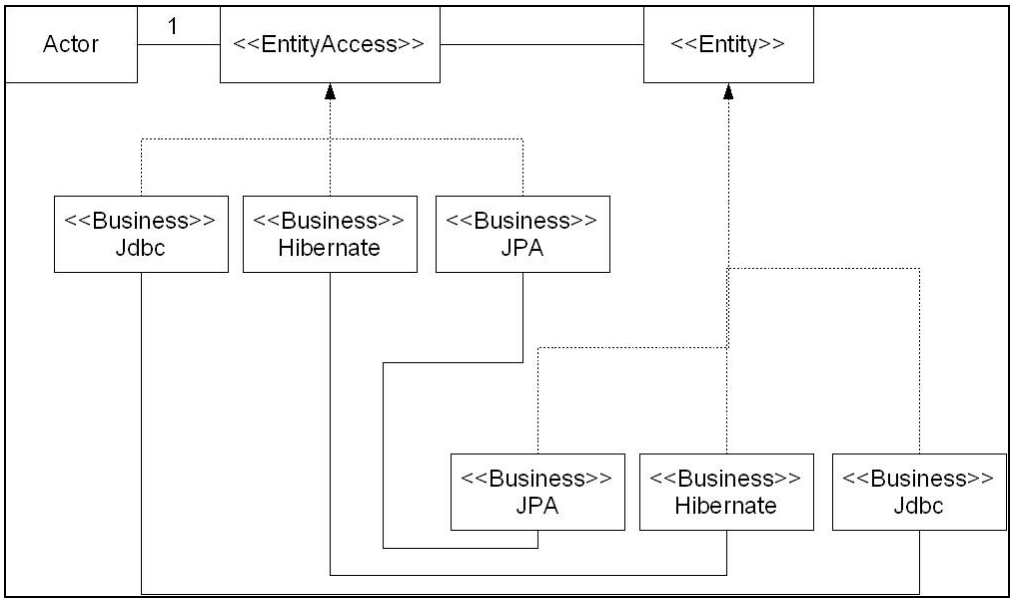
- Apache Torque verlangt neben der Entity-Implementierung auch eine Peer-Klasse[19]. Auch hier ist die Implementierung durch die Peer-Klasse darauf angewiesen, in einer Laufzeitumgebung zu existieren, die Datenbankverbindungen zur Verfügung stellt. Ein Transfer der Entity auf einen entfernten Client ist nicht möglich.
- Die letzte Variante ist momentan zu bevorzugen. Deshalb existieren hier auch eine ganze Reihe von Frameworks, die durch verschiedene Technologien die Kapselung des Persistenz-Verhaltens in einen Manager vornehmen, wobei die eigentlichen Entity-Objekte davon nichts wissen:
  - Java Data Objects: Diese Spezifikation erweitert die Entity-Klassen durch einen zusätzlichen Prozess, das Bytecode Enhancement, um das Persistenz-Verhalten. Soll die Entity außerhalb einer persistenten Umgebung benutzt werden, werden hier einfach die nicht-erweiterten Klassen verwendet. Auf Grund von Mängel in der Spezifikation und einer nicht eindeutig nachvollziehbaren Vermarktungsstrategie von Sun ist diese Spezifikation aktuell in einem nicht zukunftssicheren Zustand.
  - TopLink, Hibernate und Apache iBatis sind drei mehr oder weniger äquivalente Persistenz-Frameworks. Dabei wird die Entity als POJO-Klasse implementiert und durch eine Mapping-Datei ergänzt. Der Persistence Manager kennt diese Dateien und "weiß" somit, wie das POJO abgespeichert werden kann. Hibernate und iBatis sind Open Source Implementierungen.
  - Das Java Persistence API (JPA) bzw. Hibernate mit den Hibernate Annotations erweitern die POJO-Klasse durch spezielle Annotations, in denen das Persistenz-Verhalten definiert wird. Das JPA ist Bestandteil der JEE 5 und ist auch als Ersatz für die EntityBeans vorgesehen.

Durch diese verschiedenen Möglichkeiten der **Implementierung** einer Entity folgt eine grundsätzliche Änderung innerhalb eines Platform Independent Models: Im Gegensatz zu Value Objects sind Entities immer durch eine Schnittstelle zu definieren[20]!

---

[19] Genauer gesagt: Der Torque-Codegenerator erzeugt aus dem Datenmodell automatisch Entity- und Peer-Klassen.

[20] Auch bei Value Objects "schadet" eine Definition über eine Schnittstelle nicht bzw. ist bei komplexeren Anwendungen sogar zu bevorzugen. Bei der Besprechung des Value List Handlers wurde die Fetch-Übertragung für Collections dadurch realisiert, dass eine eigene Value List Handler-Collection-Implementierung eingeführt wurde. Soll jedoch auch ein dynamisches Nachladen von einzelnen Attributen eines Value Objects erfolgen, so erfordert dies die Implementierung der Nachladelogik im Value Object. Ebenso können fachliche Erweiterungen (Validierung, Halten von Dirty-Flags, Undo...) momentan entweder im Value Object selber oder aber einer Subklasse davon abgelegt werden.



Es existieren Business-Implementierungen für die zu unterstützenden Technologien sowohl für die Entity Access-Schnittstelle als auch für die Entity selber.

5.2.2 Transaktionsskripte

Eine Art des Datenzugriffs, die mit der vorherigen Diskussion eigentlich nichts zu tun hat, aber im Zusammenhang mit Datenzugriffen erwähnt werden soll ist das Pattern "Transaktionsskript". Hier erfolgt zwar ebenfalls ein Datenzugriff, aber die Parameter und insbesondere die Rückgabewerte sind nur bedingt Bestandteil eines Objektmodells, sondern bestehen entweder nur aus einfachen Datentypen oder Arrays.

In diesem Modell ist die Verarbeitungslogik praktisch ausschließlich innerhalb der Datenbank angesiedelt. Dazu setzt die Java-Komponente SQL-Statements ab bzw. ruft eine Stored Procedure auf. Aus der Abarbeitung dieses SQL-Skripts rührt der Begriff "Transaktionsskript" her.

Die Aufgabe einer SessionBean oder eines Servlets beschränkt sich hierbei auf:

- Holen der `Connection`.
- Eventuell Kontrolle der Transaktionen, wobei diese Aufgabe häufig ebenfalls von der Datenbank übernommen werden kann.
- Eventuell Fehlerbehandlung.
- Eventuell Interpretation der Datenbank-Antwort und Rückgabe der aufbereiteten Ergebnisse.

5.2.3 Fast Lane Reader

Ein dem Transaktionsskript sehr ähnliches Pattern ist der Fast Lane Reader. Der Unterschied ist, dass die Ergebnisse des SQL-Statements sehr wohl Objektbäume sind. Diese sind aber eher flach und nicht direkt zum Zurückschreiben der Daten gedacht. Ein anderer Name dieses Patterns ist deshalb "JDBC for Reading".

Eine typische Warenhaus-Anwendung wird als Ergebnis einer allgemeinen Produktsuche wahrscheinlich eine Menge an Treffern liefern, deren Darstellung in tabellarischer Form erfolgen wird. Die Serverseitige Umwandlung des Datenmodells, das natürlich ebenfalls in Form einer Tabelle vorliegt, in einen Objektbaum und die anschließende Client-seitige Transformation wieder zurück in eine Tabelle erfordert unnötige Objekterzeugungen und Datenkopien. In einem solchen Fall genügt als Treffer ein flaches Wertobjekt. Im Beispiel der Buchanwendung ist dies beim Holen der Buchliste der Fall.

5.2.4 EntityBeans der JEE 1.4

Die EntityBean-Spezifikation in der Version JEE 1.4 behandelt diese Komponenten als normale Enterprise JavaBeans. Wie bei den Stateless SessionBeans existieren die Home- und die Komponentenschnittstellen sowohl für eine lokale als auch für eine Remote-Variante. Die Home-Schnittstelle deklariert dabei Methoden, die zum Finden und Anlegen neuer persistenter Objekte notwendig sind. Weiterhin existieren die so genannten Home-Methoden, die Sinnvollerweise Operationen auf dem gesamten Datenbestand definieren. Die Komponentenschnittstelle enthält die Methoden, die ein einzelnes persistentes Objekt in seinem Zustand ändern können.

Diese Konzeption weist jedoch einige eklatante Mängel auf:

- Finder-Methoden können deklarativ keine dynamischen Abfrage enthalten. Sie bekommen eine feste Parameterliste, die dann eins zu eins als Parameter in ein `PreparedStatement` übernommen werden. Dynamische Abfragen oder das Hinzufügen von Order-Anweisungen etc. sind nicht möglich.
- Die Komponenten-Schnittstellen für den entfernten Zugriff müssen aus Performance-Gründen wie bei der Einführung der Service-Schnittstelle einen grob-granularen Zugriff aufweisen. Oder anders formuliert: die Remote-Schnittstelle einer EntityBean muss Value Objekte liefern. Dies ist prinzipiell natürlich problemlos möglich, jedoch in komplexeren Anwendungen nicht sinnvoll: Eine EntityBean entspricht laut Konzeption einem Datensatz, ein Value Object einer Client-Sicht auf den Datenbestand. Für verschiedene Anwendungstypen müsste folglich die EntityBean verschiedene Value Object-Typen generieren. Bei einer Änderung oder Ergänzung eines Use Cases müsste die Klasse angepasst werden, die die Datenhaltung übernimmt. Eine andere Lösung ist auch nicht immer geeignet: Die Enti-

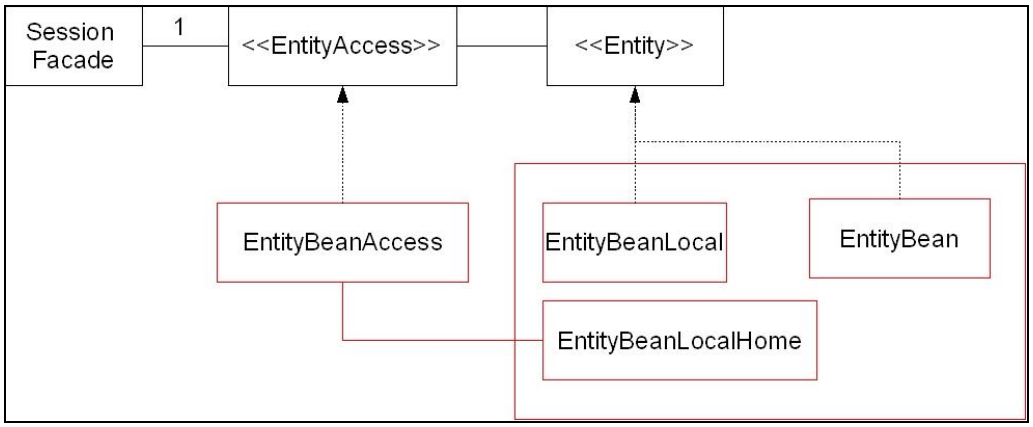
tyBean erzeugt immer ein Value Object, das dem gesamten Datenzustand entspricht. Dieses Objekt wird in der Literatur als "Data Object" bezeichnet. Die Einführung eines Data Objects erleichtert die Rollenteilung: Die EntityBean repräsentiert einen Datensatz und erzeugt das Data Object. Eine Verarbeitungslogik, beispielsweise innerhalb eines Servlets oder einer vorgeschalteten SessionBean erzeugt dann aus dem Data Object das spezielle Value Object. Dieser Ansatz krankt nun aber daran, dass für das simple Darstellen eines Datensatzes für eine Client eine vollkommen überflüssige Objekterzeugung erfolgt: Aus der EntityBean wird das (oftmals überflüssige) DataObject, daraus das Value Object.

- Die EntityBean-Instanz ist nur innerhalb des Containers gültig und kann keinesfalls direkt als Data Object oder Value Object verwendet werden. Somit ist auch in "einfachen" Anwendungen, in denen der Client wirklich ein Data Object benötigt, ein eigentlich überflüssiges Umkopieren der Daten notwendig.
- EntityBeans definieren den Datenzugriff über eigene Home-Schnittstellen. Damit wird ein echt polymorpher Zugriff praktisch unmöglich gemacht.

Um diese Probleme zu umgehen werden folgende Design-Vorgaben gemacht:

1. Der Zugriff auf eine EntityBean ist ausschließlich über die lokalen Schnittstellen zulässig.
2. Die Transaktionssteuerung übernimmt eine SessionFacade.
3. Soll der Datenzugriff verteilbar sein, so wird eine eigene Remote Session Facade dafür eingesetzt.
4. Die Komponenten-Schnittstelle enthält einen fein-granularen Zugriff auf die einzelnen persistenten Attribute der EntityBean.
5. Transaktionsskripte bzw. der Fast Lane Reader ergänzen die Entity-Bean bzw. den EntityAccess.
6. Die lokale Komponenten-Schnittstelle erweitert die Entity-Schnittstelle. Dieser Trick ermöglicht es, zumindest in lokalen Anwendungen die EntityBean unter Verzicht auf ein Data Object selber verwenden zu können[21].

[21] Ehrlicher gesagt: Die Einführung der Entity-Schnittstelle erfolgte insbesondere deshalb, um diesen Trick anwenden zu können. Andere Persistenz-Framework wie das unten angesprochene Java Persistence API oder aber O/R-Mapper wie Hibernate oder Apache iBatis arbeiten jedoch auch mit Proxy-Klassen, die beim Vorhandensein von Schnittstellen elegant erzeugt werden können.



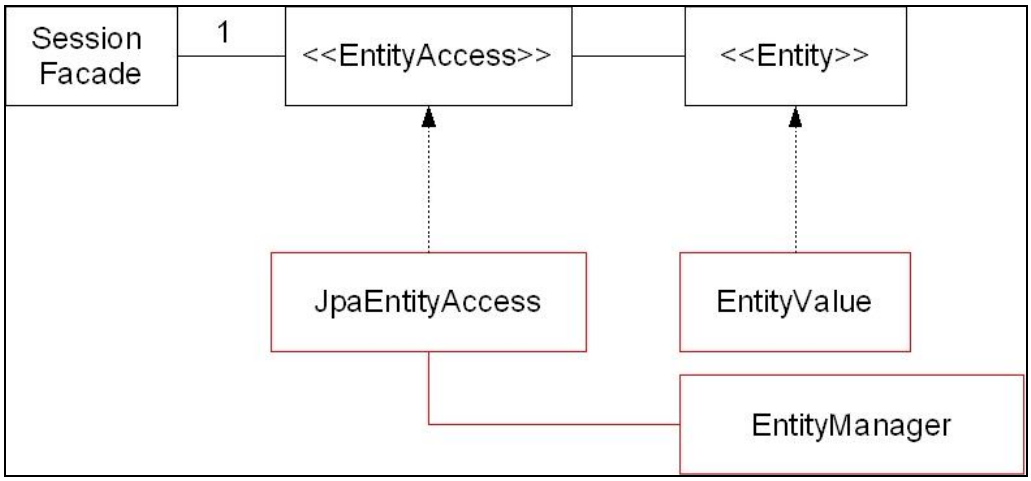
5.2.5 Java Persistence API

Das mit der JEE 5 eingeführte Java Persistence API (JPA) löst die EntityBeans ab:

- Wie bei den SessionBeans und den MessageDrivenBeans völliger Verzicht auf die Implementierung spezieller Schnittstellen für Container-Callbacks. JPA-Entities sind POJOs.
- Verzicht insbesondere auf die Verwendung der Home-Schnittstellen. Ihre Aufgaben übernimmt der generische `javax.persistence.EntityManager`. Damit wird die Programmierung deutlich einfacher und polymorphe Abfragen werden möglich.
- Annotations übernehmen die Definition der persistenten Meta-Attribute[22].

Insbesondere der POJO-Charakter der Entities ist interessant, da diese in einer Standard Java-Laufzeitumgebung ohne JPA-Erweiterung ganz normal verwendet werde können. Dies tritt insbesondere bei Client-Server-Anwendungen positiv in Erscheinung, da die Entity ohne Programmieraufwand direkt als Value Object zum Client gesendet werden kann: Durch die Serialisierung verliert das Objekt seinen Entity-Charakter und wird zum Value Object. Umgekehrt kann der `EntityManager` auch Value Objects wieder als Entitäten zur Verfügung stellen. Im Modell werden diese Klassen im Folgenden als "EntityValue" bezeichnet:

[22] Damit sind JPA-Entities eigentlich keine POJOs mehr.



EntityValue implementiert die Entity-Schnittstelle. Im Unterschied zur EntityBean-Variante können Instanzen dieser Klasse jedoch ohne Problem direkt zu einem entfernten Client gesendet werden.