1

Die Java Enterprise Edition

1 Die Java Enterprise Edition

1.1 Dienste

Als „Dienst“ einer Anwendung werden Aufgaben gruppiert, die jeder fachlichen Implementierung von der Laufzeitumgebung zur Verfügung gestellt werden.

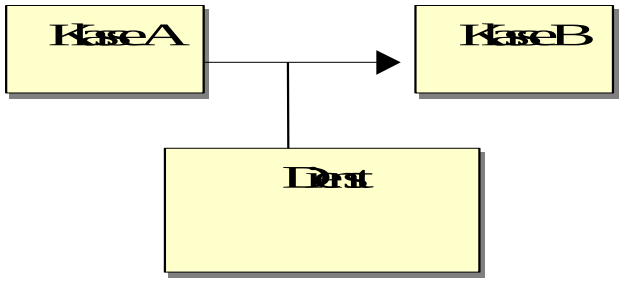
Hinweis: Dieser Ansatz ist meistens durch ein Komponentenmodell mit einem Container realisiert. Der Container bietet den installierten Komponenten die Dienste, meist formuliert als „Context“.

Es kommt nun häufig vor, dass die Dienste die eigentliche Fachlogik „umhüllen“. Typisches Beispiel hierfür sind die unten angeführten Transaktionen

- Methodenaufruf startet implizit eine Transaktion
- Die Logik wird innerhalb eines Transaktionskontextes ausgeführt
- Beim normalen Methoden-Ende wird die Transaktion implizit committed, beim Werfen einer Ausnahme zurück gesetzt

Diese Art von Diensten wird dann meistens deklarativ durch Konfigurationsdateien definiert.

In der Klassenmodellierung tauchen diese Dienste dann meist als so genannte „Assoziationsklassen“ auf:

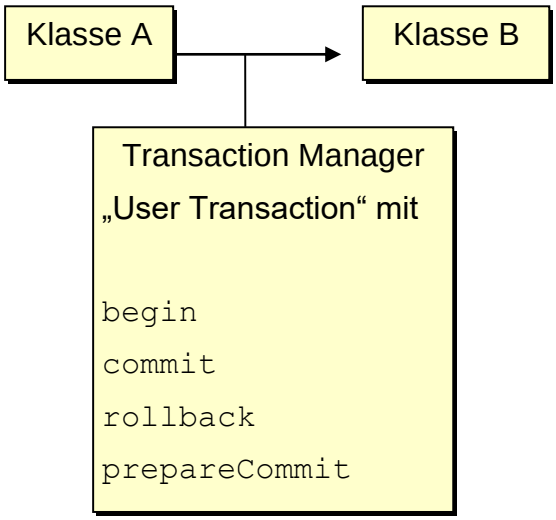


Hinweis: Aspektorientierte Programmierung: Qualitätssicherung (z.B. Prüfung auf Call-by-value), Tracing etc. sind sehr gut mit Aspekten zu realisieren. Ebenso können die deklarativen Dienste als Aspekt formuliert werden.

1.1.1 Transaktionen

Transaktionen können in den verschiedensten Schichten definiert werden:

- Client → Geschäfts-Prozess
- Innerhalb der Prozesse
- Prozess → Backend-System



Prinzipiell werden verteilte Transaktionen bis zum Backend unterstützt, „Connectors“ stellen diese Funktionalität zur Verfügung.

Hinweis: Ein Connector kann mit einem standardisierten Treiber verglichen werden.

Der entfernte Client hat innerhalb der JEE nicht notwendig Zugriff auf Transaktions-Dienste. Damit sollen lang dauernde Transaktionen, die, falls der Client eine Benutzerschnittstelle darstellt, sogar mehrere Eingaben andauern könnten, vermieden werden.

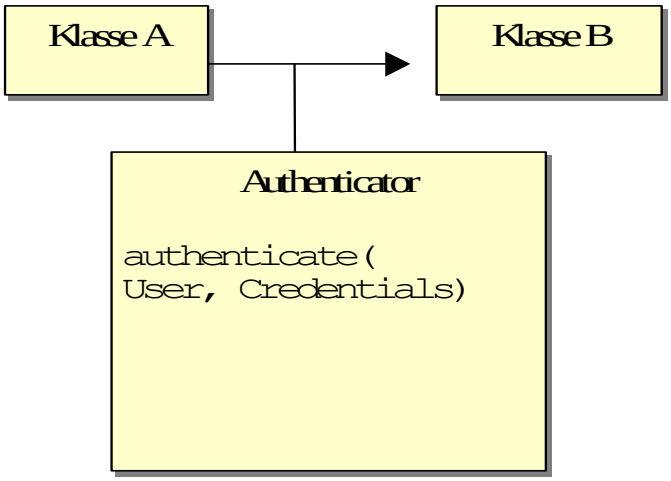
Konsequenzen:

- Eigene User-Transaction für Methoden-übergreifende Transaktionen. Dies sind oft herstellerspezifische Erweiterungen, „Client-User-Transaction“
- „Select for update“ zum aktiven Sperren von Datensätzen
- „Optimistic Locking“ z. B. mit timestamps.

1.1.2 Authentifizierung

Auch die Authentifizierung kann sich in allen Schichten befinden:

- Client → Prozess
- Innerhalb der Prozesse
- Prozess → Backend, Bestandteil des Connectors



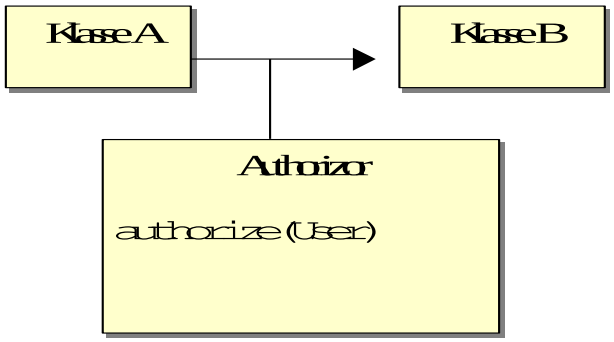
Die folgenden Kommunikationsprotokolle unterstützen die Authentifizierungsdienste:

- RMI/IIOP: JAAS (Java Authentication & Authorization Service)
- http: Browser-, Formular-, Zertifikat-basiert
- Web-Services/SOAP: noch nicht vollständig spezifiziert
- Messaging/JMS: nicht vorgesehen bzw. nur über den Message Provider

Konsequenzen:

- Ein Authorizing Framework muss von der DV zur Verfügung gestellt werden.
- Gegebenenfalls sind eigene Authentifizierungs-Protokolle (Web-Services, Messaging) mit User/Profile-Objekt nötig.

1.1.3 Autorisierung



Die JEE spezifiziert ein statisch-deklaratives Authorizing-Konzept:

- Anwender-Alias-Rollen: (admin, user, guest...) werden „realen“ Benutzern zugeordnet.
- Applikations-Server mapped Methoden/Komponenten zu den Anwender-Alias-Rollen

Dieses statische Konzept ist jedoch bei komplexen Anforderungen mehr oder weniger unbrauchbar:

- schwer wartbar
- keine dynamischen Prüfungen (z.B. nur Löschen eigener Datensätze, 4 Augen Prinzip…)

Konsequenz:

Das Design der Anwendung sollte einen eigenen Authorizor vorsehen.

1.1.4 Ausfallsicherheit

Die Skalierbarkeit wird garantiert vom Komponenten-Modell der JEE: Der Container kontrolliert den Lebenszyklus. Als Besonderheit der JEE im Vergleich zu .NET existieren hier Stateful Komponenten, die bei der Ausfallsicherheit gesondert zu betrachten sind.

Hinweis: .NET speichert Zustände sofort persistent in der Datenbank.

Stateful-Komponenten in der JEE sind:

- Stateful-Session-Bean (Daten in nicht transienten Attributen)
- Servlets (Daten in http-Session)

Es gibt nun durchaus Gründe, die gegen das Halten von Client-typischen Zuständen innerhalb des Servers sprechen:

- schlechtere Skalierbarkeit (activate/passivate; „Swap“)
- Größe des Client-Zustandes maximal 50K. Mehr garantieren die Applikationsserver-Hersteller nicht

Dies führt nun jedoch zu Problemen mit „großen“ Sessions: Der Umstieg auf ausfallsicheren Cluster führt zu massiven Performance-Problemen, da nun große Informationsmengen im Hintergrund gehalten und repliziert werden müssen.

Konsequenzen:

Das Design der Anwendung sollte einen „SessionStore“ beinhalten. der zentral als Ablage von Zuständen benutzt werden soll.

Hinweis: Die Einführung des Session Stores alleine verbessert die Performance der Anwendung natürlich erst einmal gar nicht. Es wird jedoch eine zentrale Stelle definiert, an der dann Hochspezialisierte Optimierungsmaßnahmen durchgeführt werden können.

So werden beim Session Store die Attribute einer Stateful-Session-Bean analog zur Http-Session verwaltet:

Statt:

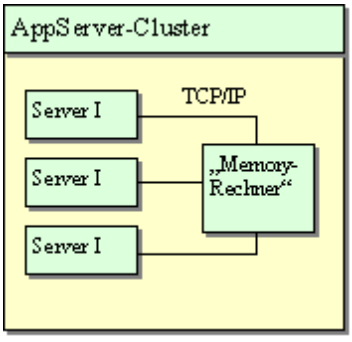
```
private String.info..;
private Data data1;
private List data2;...
```

wird der Session Store eingeführt, der intern einfach aus key-value-Paaren (also einer `Map`) besteht:

```
private SessionStore store;
```

Session-Store Implementierungs-Strategien:

- Lokale Map ≙ normales Passivieren/Aktivieren der Zustände in das lokale Filesystem oder eine Datenbank
- Map, die DB-Zugriffe macht ≙ .NET-Strategie
- Memory-Store, bei der der Speicher des Clusters benutzt wird. Eine von Herstellen recht häufig angebotene Strategie.
- Als Erweiterung der letzten Strategie können die Zustände von einem speziell und ausschließlich dafür vorgesehenen „Memory Rechner“ gehalten werden:



All diese Ideen werden von den Herstellern der Applikationsserver mehr oder weniger vollständig unterstützt. Durch die Einführung des Session Stores als Bestandteil des eigenen Anwendungsdesigns ergeben sich jedoch deutliche Vorteile bei späteren Optimierungen:

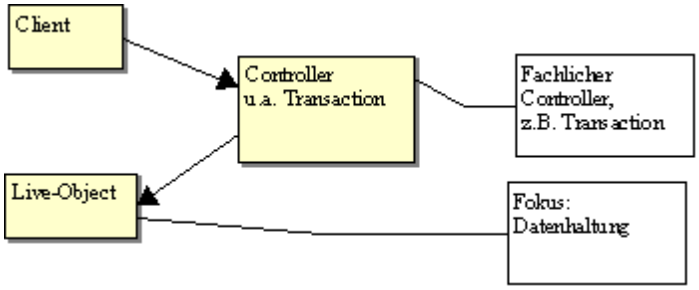
- Ein intelligenter Session Store ändert seine Strategie in Abhängigkeit von der Größe der gehaltenen Session
- Eigene Strategien können implementiert werden, ohne die Dienste des Applikationsservers anpassen zu müssen.

1.1.5 Zugriff auf persistente Daten

Der Zugriff auf persistente Daten ist ein so klares Problem der Anwendungsprogrammierung, dass eine ganze Reihe bereits definierter Modelle existieren:

- Elementares JDBC mit Result-Set + SQL
- Entity Beans (BMP, CMP)
- O/R-Mapping, z.B. mit Hibernate oder Toplink
- JDO
- …

Welche Technologie für welche Anwendung am Besten geeignet ist, kann hier leider nicht detailliert behandelt werden. Für das Design der Anwendung ist jedoch essenziell wichtig ist, einen möglichen Austausch des verwendeten Persistenz-Frameworks voraus zu sehen.

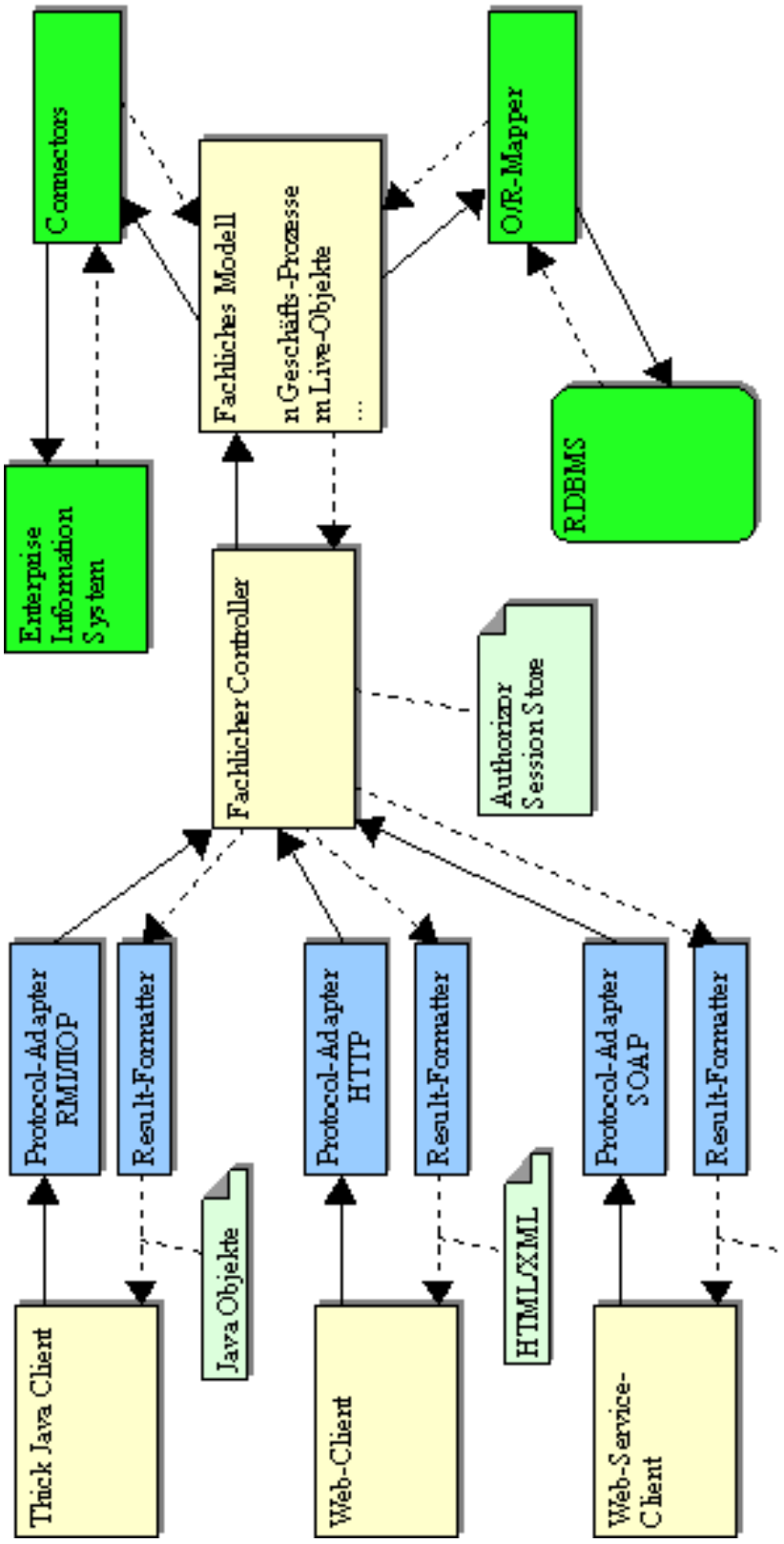


Hier soll nun jedoch noch auf eine Besonderheit bei der Verwendung von Objekten, die persistenten Datenzuständen entsprechen, hingewiesen werden. Mit einer durchaus vertretbaren Architektur-Entscheidung kann der Zugriff auf diese Daten nämlich „live“ erfolgen: Client und Persistenzschicht laufen auf einer Maschine.

Hinweis: Das bedeutet natürlich nicht, dass keine entfernten Zugriffe realisiert werden können! Der entfernte Zugriff benötigt nun jedoch eine eigene Fassade, eben das Pattern „Remote Facade“.

Hinweis: Die JEE ist sich dieser Problematik bewusst und stellt für lokale Zugriffe die „Local“-Schnittstellen zur Verfügung.

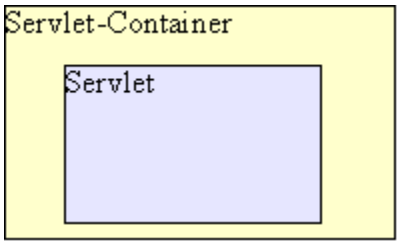
1.1.6 Gesamtbild



1.2 JEE Komponentenmodelle

Die JEE ist ein Komponenten-Modell, das für die verschiedenen Typen eigene Spezifikationen für Container definiert. Die folgenden Kapitel beschreiben die grundlegende Arbeitsweise dieser Container.

1.2.1 Der Servlet Container



Arbeitsweise:

- Lebenszyklus der Servlet-Instanz: Es genügt eine Servlet-Instanz pro Container
- Verwaltung des Zustands pro Request/Session
- Einfache Authentifizierung und Autorisierung

Servlets müssen zustandslos und threadsicher konzipiert werden, so dass alle referenzierten Klassen müssen threadsicher sein müssen.

Die Http-Session kann zum Halten des Zustands verwendet werden; verlangt die Architektur Lastverteilung oder Ausfall-Sicherheit, ist Replizierung nötig (und damit gilt die 50K-Grenze). Auch für Servlets ist somit die Verwendung des Session Stores sinnvoll.

Servlet-Komponenten brauchen häufig zusätzlich die folgenden Dienste:

- Transaktions-Steuerung
- Synchronisierung von Aufrufen bei nicht multithreading-fähigen Systemen

Nachdem Servlets häufig Dokumenten-Formate wie HTML oder XML liefern müssen, werden meistens noch zusätzlich Result-Formatter benötigt:

- Java Server Pages: = durch ein Template erzeugtes Servlet

Einfache Hilfs-Klassen: XML-Erzeugung mit JAXB/XSL oder andere Template-Sprachen wie beispielsweise Apache Velocity.

Die typischen Einsatzbereiche eines Servlets innerhalb der erweiterten MVC-Architektur sind somit:

- Protokoll-Adapter für http-Requests
- Result-Formatter (auch eine JavaServer Page ist ein Servlet!)

- Controller für Web-Anwendungen, vergleiche die Pattern „Page Controller“ bzw. „Front Controller“

1.2.2 Der Enterprise JavaBean Container

1.2.2.1 Exkurs: Verwendung von Enterprise JavaBeans

Bevor der Container für Enterprise JavaBeans besprochen werden kann, soll erst einmal die Frage „Für welche Zwecke wird eine Enterprise JavaBean benötigt“ behandelt werden. Ein grundsätzlicher Fehler einer JEE-Anwendung ist es nämlich, innerhalb einer EJB Fachlogik zu implementieren.

Hinweis: Dies ist natürlich kein syntaktischer Fehler, sondern „nur“ eine Design-Entscheidung. Enthält eine Enterprise Bean Fachlogik, so ist diese natürlich nur innerhalb des Applikationsservers lauffähig. Eine typischerweise bei großen Projekten notwendige Verteilung der Logik auf die verschiedensten heterogenen Systeme wäre damit unmöglich. Ist die Sequenz eines Datenzugriffs beispielsweise direkt in einer SessionBean implementiert, so ist es nur durch nachträgliches Ändern möglich, diese Zugriffslogik als Stored Procedure in die Datenbank oder, falls bei einer anderen Architektur nur ein Servlet Container zur Verfügung steht, in die Web Schicht zu verschieben.

Aber nicht nur die Implementierung einer EJB sollte frei von fachlichen Aufrufen sein. Auch die Signatur ist nicht die Signatur des MVC-Controllers oder -Models!

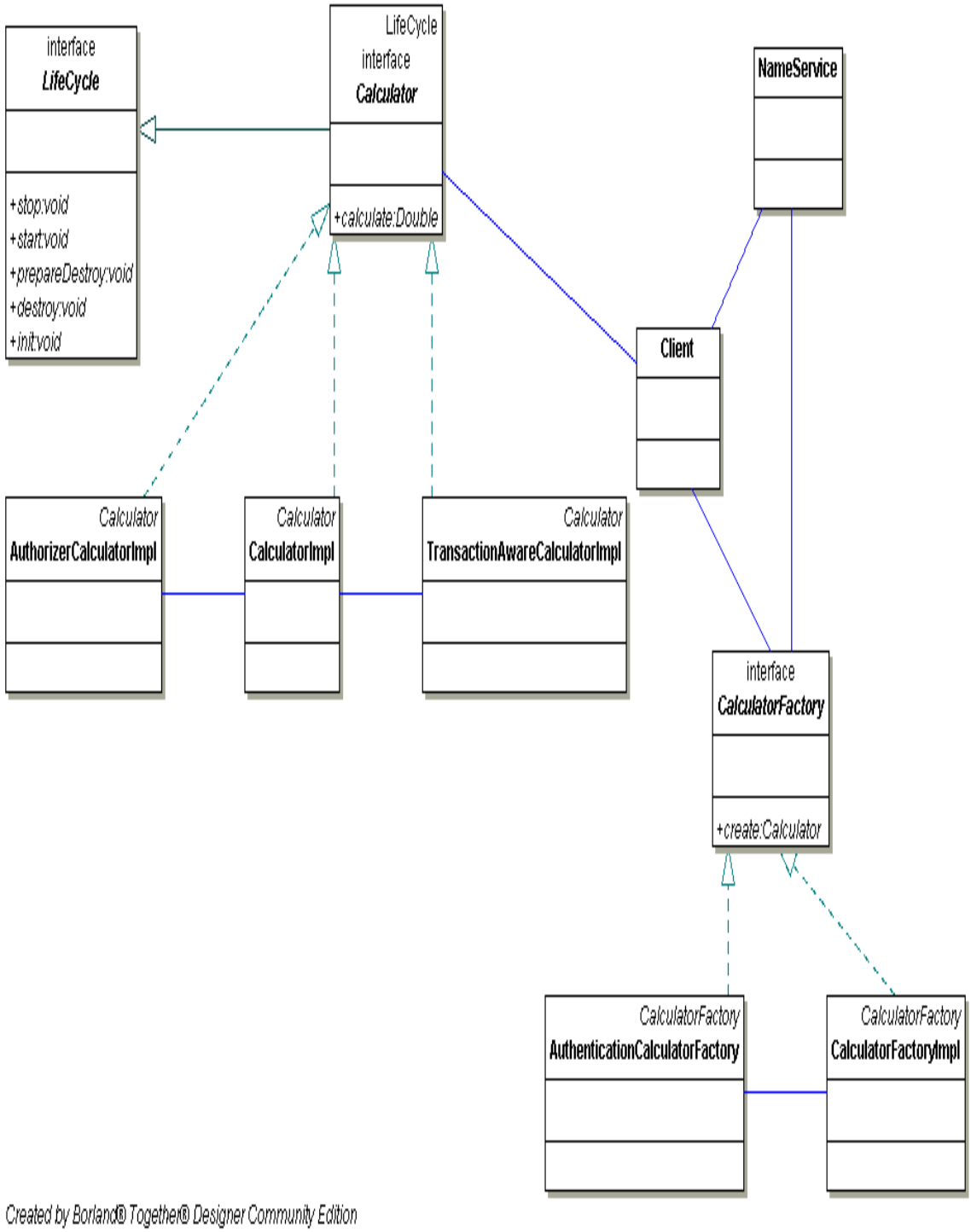
Dies lässt sich einerseits aus der Programmiersprache Java heraus begründen: Jeder Methodenaufruf einer EJB wird ja über einen Container delegiert. Für Remote Zugriffe wirft dieser Container auf Grund der Semantik eines entfernten Methodenaufrufs in Java eine bestimmte Ausnahme, die `java.rmi.RemoteException`. Dieser Ausnahmetyp muss vom Client behandelt werden, so dass dann beispielsweise eine Controller-Schnittstelle diesen doch sehr speziellen Ausnahmetyp kennen müsste. Ein klarer Verstoß gegen die Plattformunabhängigkeit!

Diese Begründung alleine ist jedoch nicht sonderlich gut, weil sie implizit eine Design-Schwäche der RMI-Aufrufe in Java impliziert. Bei .NET ist `RemoteException` und damit das Problem in dieser Form auch nicht vorhanden. Das eigentliche Argument, die fachliche MVC-Schnittstelle von der technologischen EJB-Schnittstelle komplett zu entkoppeln ist, dass bei späteren Optimierungsmaßnahmen beide Signaturen unabhängig und damit effizient optimiert werden können.

Hinweis: Soll eine bestimmte fachliche Implementierung über eine EJB angesprochen werden, wird die zugehörige technologische EJB-Schnittstelle im Standardfall erst einmal genau so aussehen, wie die fachliche Schnittstelle, jedoch ergänzt um die RemoteException. Diese Redun-

danz verschwindet aber bzw. ist nur noch teilweise gültig, sobald eben durch Optimierungsmaßnahmen Änderungen notwendig geworden sind.

Beispiel:



1.2.2.2 Der Enterprise JavaBean Container

Die EJB-Spezifikation von Sun definiert:

- Allgemeine Spezifikation für die Factory-Schnittstelle („Home“ mit create- und remove-Methoden
- Allgemeine Spezifikation für die technologische Schnittstelle („Component mit beliebigen Methoden“)
- Spezifikation für Proxy-Dienste des Containers
- Definition des Lebenszyklus für Implementierungen

Je nach technischer Architektur des Gesamtsystems sind zwei Strategien für die Enterprise JavaBeans vorhanden:

- Home/Component erlauben einen Zugriff über Remote-Aufrufe („Remote“-Schnittstellen)
- Home/Component sind nur lokal verfügbar („Local“-Schnittstellen)

Folgende Dienste eines EJB-Containers sind definiert, wobei in der rechten Spalte der Tabelle noch ein Kommentar über den zu erwartenden Mehraufwand zur Laufzeit steht[1]:

Deklarative Transaktions-Steuerung	notwendig
Propagierung des Security-Contextes (Prinzipal)	Overhead, abschaltbar
Deklarative Autorisierung	Overhead, abschaltbar
Zuordnung Client-Bean-Instanz	notwendig
Caching/Pooling der Bean-Instanzen	Overhead
Synchronisierung von „Multithreaded Aufrufen“	Overhead, Bean-Instanz ist stets „singlethreaded“

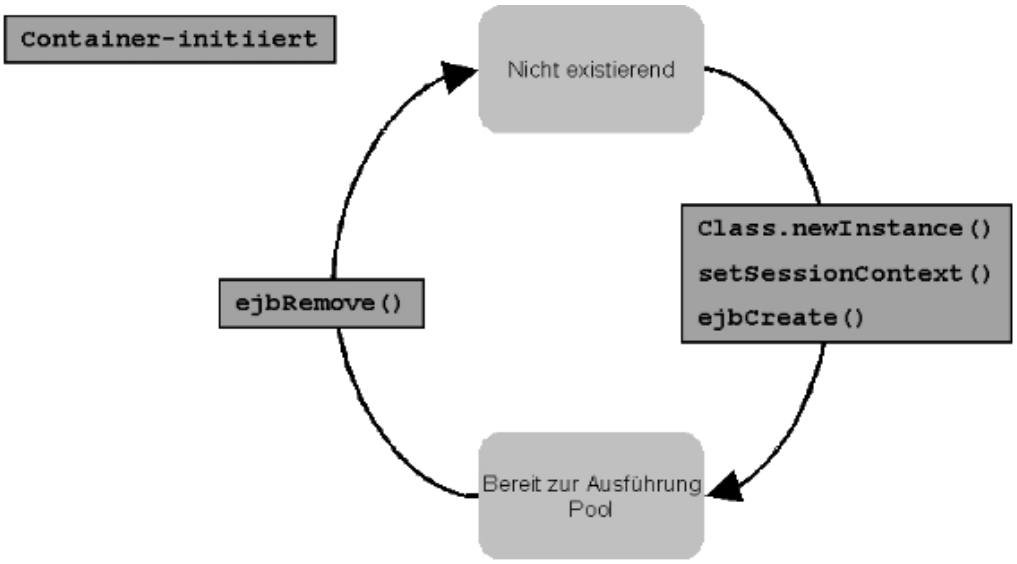
Insgesamt zeigt sich, dass bei richtiger Konfiguration der durch den EJB-Container bedingte Overhead recht gering ausfällt. Transparente Dienste wie die deklarative Transaktionssteuerung sind auch in schlankeren Frameworks notwendig.

[1] EJBs und Applikationsserver sind in letzter Zeit etwas in die Kritik gekommen, weil die Komponenten als „zu dick“ kritisiert werden. Dies ist der Ansatzpunkt für schlankere Frameworks, z.B. Spring, Keel oder auch der JBoss MicroContainer.

1.2.3 Stateless SessionBeans

1.2.3.1 Eigenschaften

- kein Client-typischer Zustand
- Ausfall-Sicherheit ist sehr gut, nicht abgefangen ist nur der Ausfall des Servers während eines Geschäfts-Prozesses. Falls die Session-Bean jedoch „idempotent“ ist (mehrfache Aufrufe haben stets die gleichen Ergebnisse ohne Nebeneffekte), kann sogar in dieser extremen Situation neu aufgesetzt werden.
- Lebenszyklus ist rein Container-initiiert (vgl. Instanz-Pool, Swapping-Proxy)



1.2.3.2 Anwendungsbeispiele

- Bestandteil des Modells, definieren allgemeine, zustandslose Geschäftslogik
- Controller sind bevorzugt zustandslos zu halten und damit auch stateless session beans
- Result-Formatter

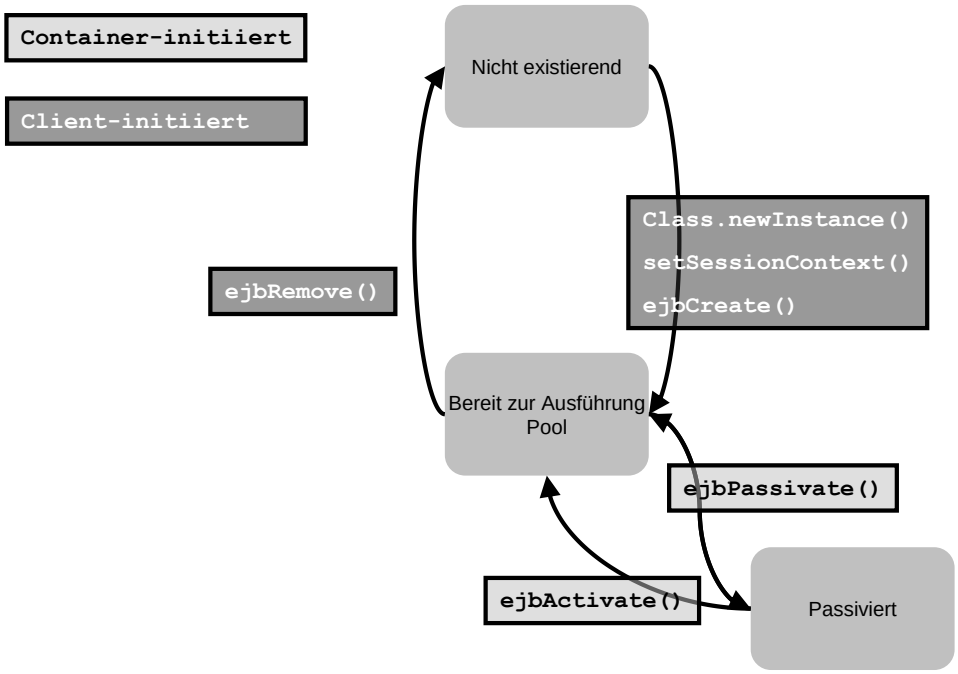
1.2.3.3 Einsatz Kriterien

- Lastverteilung,
- Ausfallsicherheit,
- Zustandslose Dienste

1.2.4 Stateful SessionBeans

1.2.4.1 Eigenschaften

- Client-typischer Zustand wird auf Server-Seite gehalten
- Ausfallsicherheit befriedigend
- Passivierung/Aktivierung des Zustands erfolgt am Ende des Methodenaufrufs und am Ende einer Transaktion. Damit Probleme bei großen Sessions
- Lebenszyklus ist sowohl Client- als auch Container-initiiert (create/remove: Client, activate/passivate/remove (bei Timeout): Container)



1.2.4.2 Anwendungsbeispiele

- Protocol-Adapter
- Controller, falls die Fachlogik ein Halten von Client-Zustand erfordert

1.2.4.3 Einsatz-Kriterien

- Server-Objekt enthält sinnvolle Zustands-Informationen (sinnvoll im Allgemeinen nur als Protocol-Adapter)
- Vorsicht bei Verbindung von Stateful-Session-Beans (u.a. wegen evtl. unterschiedlicher Timeouts)

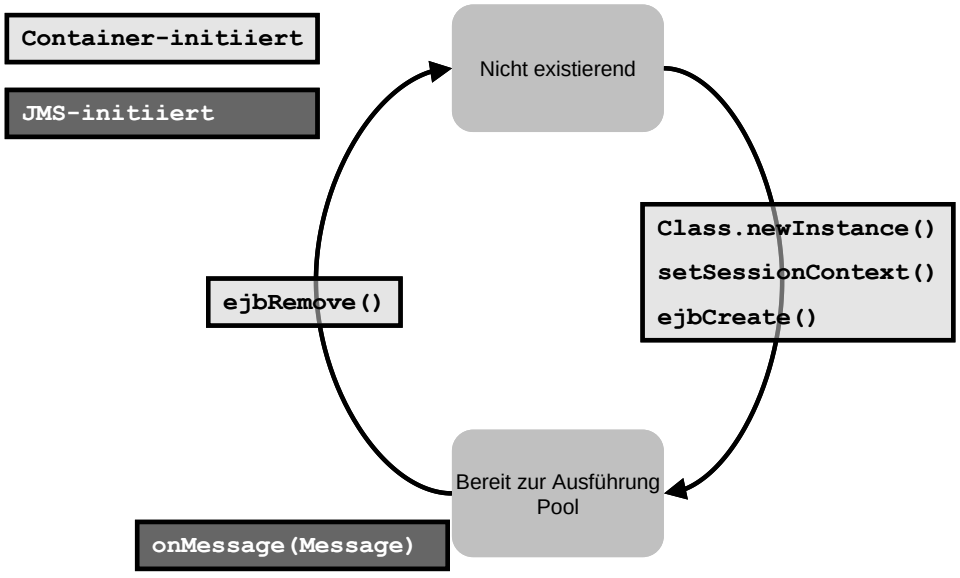
1.2.4.4 Abgrenzung Stateless- Stateful SessionBeans

Erfordert die Implementierung der Fachlogik das Halten des Zustands auf dem Server, bleiben nur Stateful SessionBeans übrig. Die Performance der beiden Bean Typen ist bei einem einzigen Applikationsserver vergleichbar. Im Cluster ist jedoch zu berücksichtigen, dass die Verwaltung von Stateful SessionBeans aufwändiger ist. Der Applikationsserver muss durch Replizieren des Conversational State die Ausfallsicherheit bzw. Lastverteilung gewährleisten. Auch hier ist jedoch zu berücksichtigen, dass diese Funktionalität notwendig ist und deshalb nicht aus Gründen der Geschwindigkeit geopfert werden kann.

1.2.5 MessageDrivenBeans

1.2.5.1 Eigenschaften

- Ist zustandslos
- Kann Transaktionen starten, kann aber keinen Transaktions-Kontext propagieren
- Ausfallsicherheit: Prinzipiell sehr gut, ein Problem ist gegebenenfalls nur die Ausfallsicherheit der Message-Destination
- Lebenszyklus wie bei den Stateless SessionBeans



1.2.5.2 Anwendungsbeispiele

- Protocol-Adapter für JMS
- Bestandteil des Modells, enthalten asynchron aufrufbare Logik

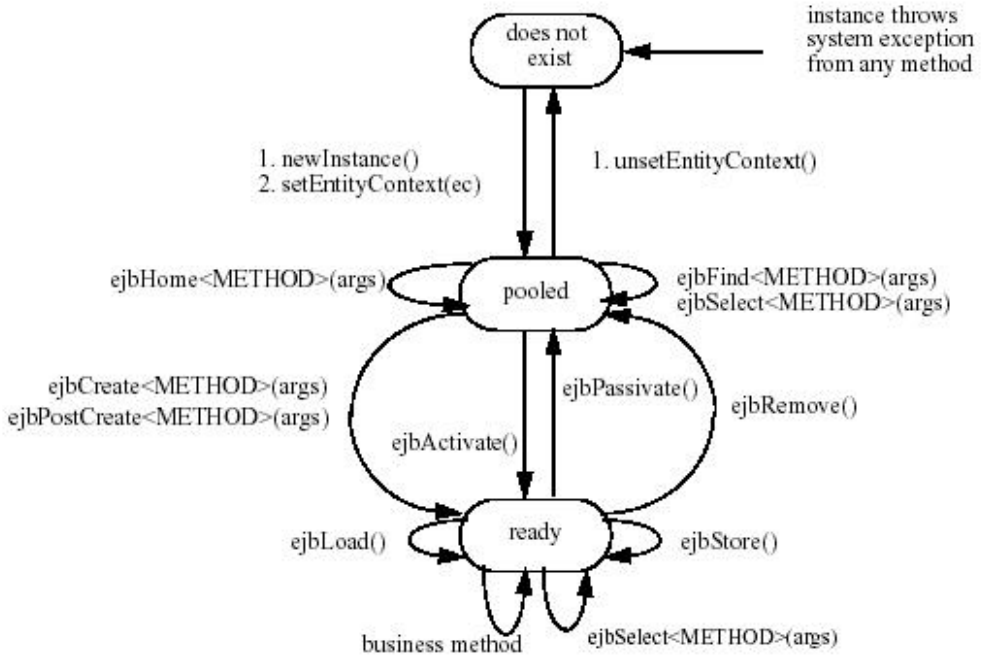
1.2.5.3 Einsatz-Kriterien

- Methoden ohne Rückgabewert
- Höchste Ausfallsicherheit durch Zwischenspeicherung der Messages durch den Message-Provider vor der Ausführung der eventuell nicht einsatzbereiten Fachkomponenten

1.2.6 EntityBeans

1.2.6.1 Eigenschaften

- Aus Sicht des Client zustandslos
- Repräsentiert einen eindeutig identifizierbaren Zustand
- Kann anhand einer ID gefunden werden
- Aufrufe unterschiedlicher Clients werden sequentiell abgearbeitet -> Zusätzlicher Container-Dienst: Synchronisierung mehrerer Clients
- Lebenszyklus



1.2.6.2 Ausfallsicherheit

- Entity beans sind persistent zu konzipieren (O/R-Mapper)
- Entity beans stellen einen Cluster-übergreifenden Cache für persistente Objekte dar. Optimal konfigurierte EntityBeans:
  - Verringern die Menge an DB-Zugriffen
  - Verringern im Mittel die Antwortzeiten durch Caching-Mechanismen
  - Führen zu erhöhter Speicherlast des Applikationsservers
  - Insbesondere bei komplexen Entity-Bäumen können einzelne Zugriffe sehr lange dauern
- Abfrage zur DB ist SQL-lastig, andere Persistenz-Mechanismen werden nicht unbedingt direkt vom Container unterstützt.

1.2.6.3 Anwendungsbeispiele

- Read-Only-Datenbestände
- „read-mostly“ - entity beans (e-Business-Pattern)
- nicht-persistenter gemeinsam nutzbarer Memory-Cache

1.2.6.4 Einsatz-Kriterien

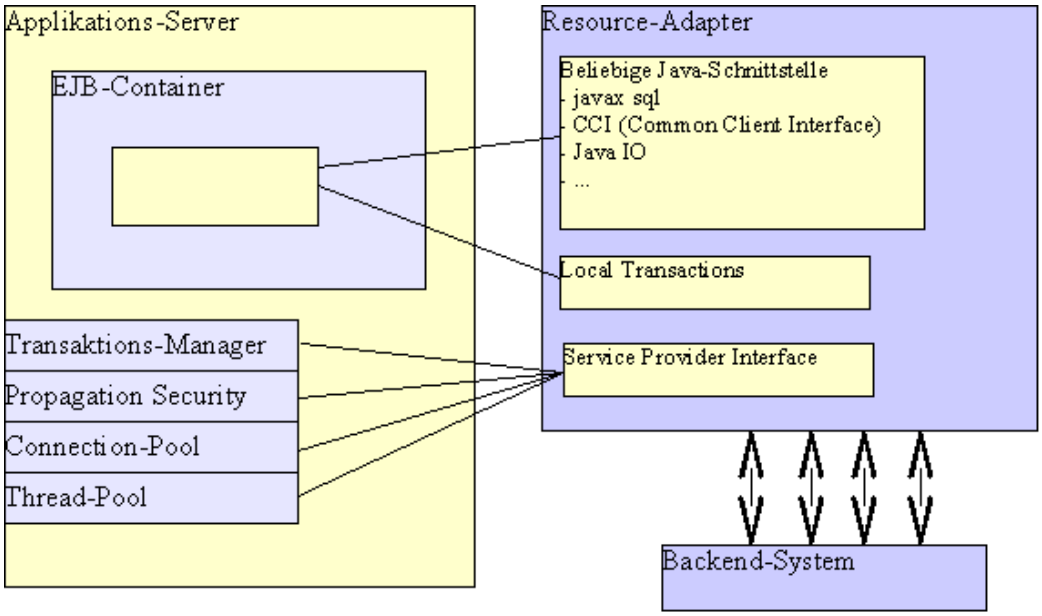
- Ein Cluster-übergreifender Cache für persistente Objekte ist sinnvoll

1.2.7 Die Java Connector Architecture

Ziel der Java Connector Architecture (JCA) ist die Integration eines vorhandenen Backend-Systems. Häufig ist dieses System transaktionsfähig und verlangt Authentifizierung. Bei der Anbindung des Backend-Systems sollen die zentralen Dienste des Applikationsservers (Resource-Management durch Connection Pooling, Transaktionsmanagement, Multithreaded Ausführung von Anweisungen…) natürlich benutzt werden. Deshalb muss ein Connector bzw. der Resource Adapter insgesamt drei Schnittstellen bedienen:

- Eine proprietäre Schnittstelle zum Backend-System
- Eine Schnittstelle zu den Diensten des Applikationsservers (das Service Provider Interface, SPI)
- Eine Schnittstelle für den Zugriff von Komponenten. Dies ist eine beliebige Schnittstelle, die bereits Bestandteil der Java-Bibliothek sein kann (z.B., eine `javax.sql.DataSource`), eine Selbstdefinierte Schnittstelle oder das sehr allgemein gehaltene Common Client Interface (aus `javax.resource.cci`).

Auf Grund dieser Spezifikation ist ein Connector von allen Applikationsservern benutzbar.



1.3 Das Basis-Design mit JEE-Komponenten