3

JEE und die Business Schicht

3.1 Services 3-3
3.1.1 Definition 3-3
3.1.2 Service-Beschreibung und Java 3-3
3.1.3 Service-Realisierung in Java 3-4
3.2 Context and Dependency Injection 3-6
3.3 Dekoration 3-9
3.3.1 Business Object 3-9
3.3.2 Container 3-10
3.3.3 Aspektorientierte Programmierung 3-11
3.3.4 Das JEE Pattern Layer Supertype 3-11
3.4 Realisierung 3-12
3.4.1 Stateless SessionBeans 3-12
3.4.2 Erweiterte Paketstruktur 3-13
3.5 Zusammenfassung 3-14
3.6 Beispiel: Eine SessionBean für den KeyGenerator 3-14

3 JEE und die Business Schicht

3.1 Services

3.1.1 Definition

Der Begriff eines Services wird in den unterschiedlichsten Kontexten benutzt und ist deshalb ohne weitere Erläuterung nicht verwendbar.

Wikipedia definiert einen Service wie folgt:

„*the term service refers to a set of related software functionalities that can be reused for different purposes, together with the policies that should control its usage*“

Wichtiger ist noch die Einführung der Service-Beschreibung:

„*A service has a description or specification. This description consists of:*

An explicit and detailed narrative definition supported by a low-level (not implementation-specific) process model. The narrative definition is in some cases augmented by machine-readable semantic information about the service that facilitates service mediation and consistency checking of an enterprise architecture.

A set of performance indicators that address measures and performance parameters, such as availability (when should members of the organization be able to perform the function?), duration (how long should it take to perform the function?), rate (how often will the function be performed over a period of time?), etc.

A link to the organization's information model showing what information the service owns (creates, reads, updates, and deletes) and which information it references that is owned by other services.“

Ein Service hat somit eine wohl definierte Beschreibung, die neben den rein fachlichen Vorgaben auch einen Satz von Policies wie Verfügbarkeit und Performance enthält.

3.1.2 Service-Beschreibung und Java

Bei der eigentlichen Service-Beschreibung muss in Java einiges an Infrastruktur nachgebessert werden. Die naheliegende Benutzung eines Interfaces als Service-Beschreibung ist viel zu grob: Es fehlen sämtliche Policy-relevanten Informationen. Um Policies zu definieren können selbstverständlich Annotationen benutzt werden, diese sind jedoch nur teilweise, wenn überhaupt, in der Spezifikation enthalten. Ebenso sind geeignete Kandidaten häufig in technischen Paketen gelandet: Die

Semantik einer `javax.persistence.Entity` ist nicht notwendig auf einen O/R-Mapper beschränkt.

Es ist deshalb zu empfehlen, für eigene Projekte einen Satz von Annotationen zu erstellen und deren Bedeutung exakt zu definieren[1]. Minimal wird eine `@Service`-Annotation benutzt, die eine Verlinkung auf die Service-Dokumentation enthält. Spezielle Services, werden durch Subklassen beschrieben, beispielsweise ein `@CrudService` für Datenbanknahe Operationen. Ebenso werden Policies durch eine Vererbungshierarchie realisiert.

Sind die Services auf Grund einer Architektur-Entscheidung als Web Services ausgebildet kann natürlich auch die Jax-WS-Annotation `@WebService` benutzt werden, die Service-Beschreibung erfolgt dann in einer WSDL.

Wird UML als Modellierungssprache eingesetzt werden eigene Stereotypen definiert. UML-Stereotypen können eins zu eins auf Annotationen umgesetzt werden und umgekehrt.

3.1.3 Service-Realisierung in Java

Java und die Java Enterprise Edition sind prinzipiell für die Realisierung von Services geeignet. Insbesondere bieten die automatisch für Enterprise JavaBeans vom Applikationsserver gesammelten Metriken sowie die integrierte Überwachungssoftware eine geeignete Technik zur Service-Governance.

Die eigentliche Service-Implementierung sollte jedoch mit möglichst wenig Bezug zu einer bestimmten Technologie erfolgen, um eine möglichst weite Wiederverwendung zu ermöglichen. Dies ist die Grundidee der sogenannten Plain Old Java Objects oder kurz: POJOs.

POJOs waren in der J2EE extrem wichtig, da Enterprise JavaBeans durch Implementierung von Schnittstellen extrem Technik-abhängige Komponenten waren. Diese Problematik hat sich durch Annotationen deutlich verschärft. Ob durch Annotieren einer Klasse diese noch als POJO aufgefasst werden kann oder nicht ist eindeutig geklärt und kann deshalb Projekt-abhängig festgelegt werden.

[1] Das Spring-Framework hat diese Idee mit Annotationen wie @Component, @Controller etc. bereits teilweise umgesetzt.

Beispiel: Ein POJO mit Annotationen:

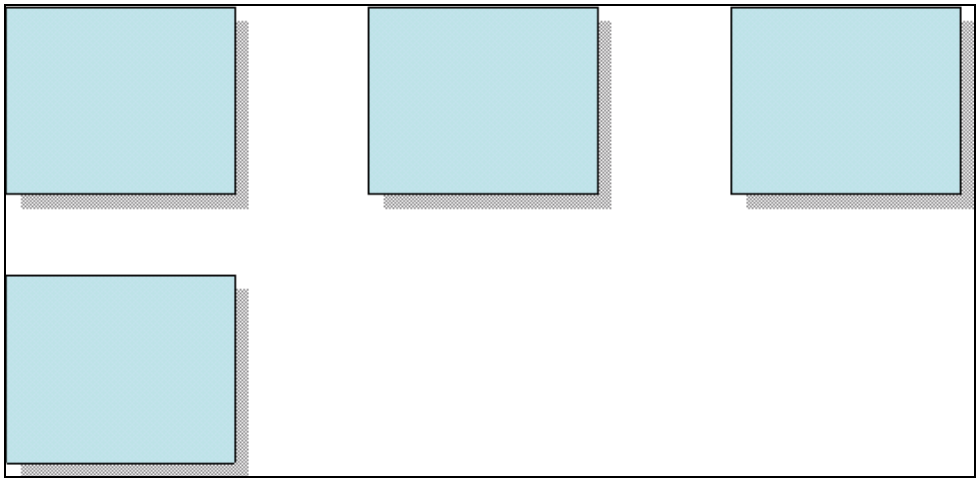
```
<<class>>
ApplicationBean

@Stateless
@Local
@Remote
@TransactionAttribute
@Resource
@PersistenceContext
@WebService
@SoapBinding
@WebMethod
@WebParam
@WebResult
public void business(){
    //…
}
```

3.2 Context and Dependency Injection

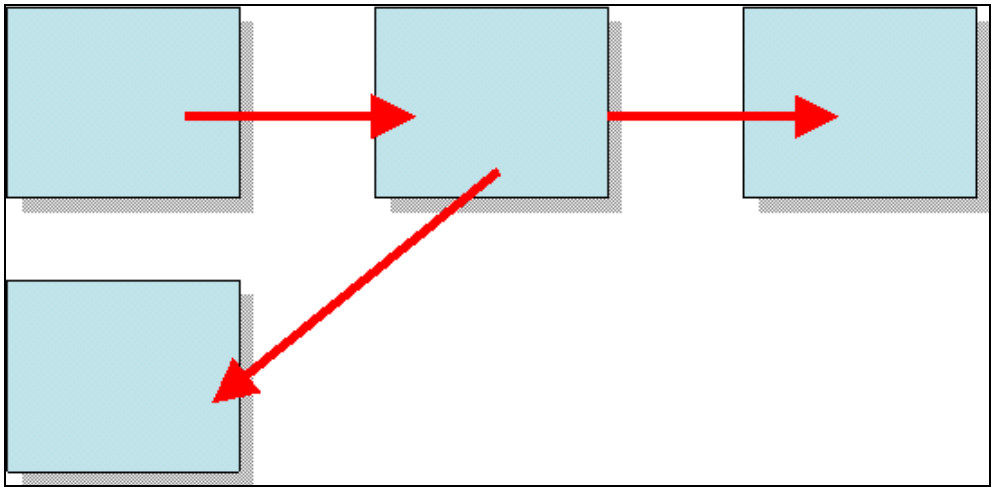
Context and Dependency Injection oder kurz CDI ist in modernen Anwendungen die Grundlage des Anwendungsdesigns. Ob CDI nun als Design Pattern oder als Framework bezeichnet wird mag diskussionswürdig sein. Die Große Menge an verschiedenen Implementierungen auch auf völlig unterschiedlichen Plattformen spricht jedoch klar für den Pattern-Begriff.

Als zentrale konkrete Komponente von CDI existiert die Context-Implementierung. Diese wird in der Regel nicht selbst entwickelt sondern über ein Framework zur Verfügung gestellt. Die Aufgaben des Kontextes lauten:

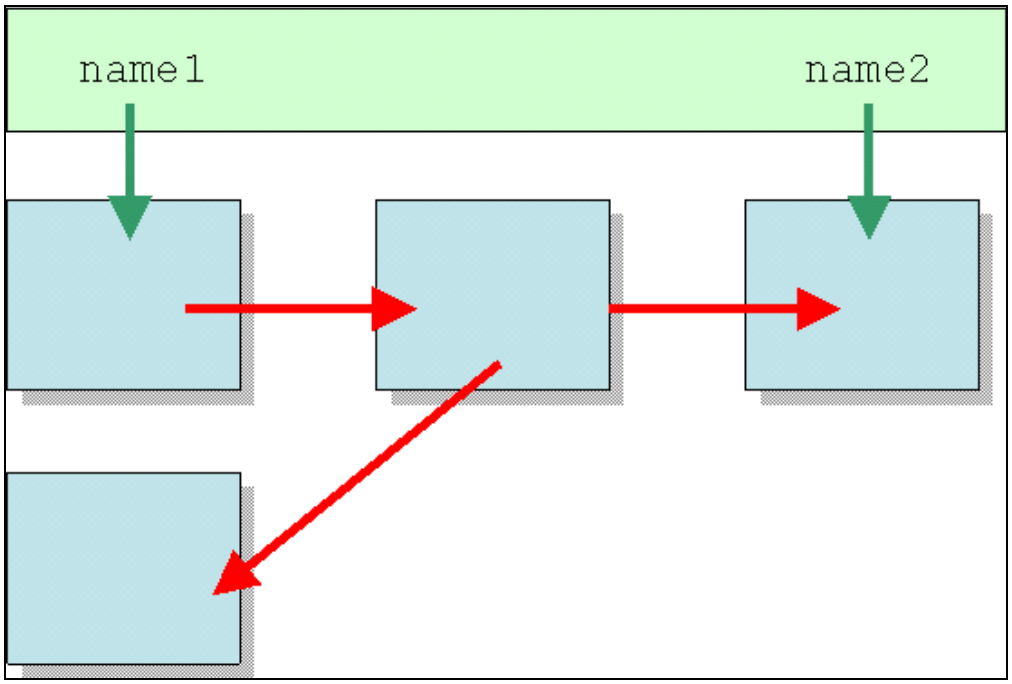
- Erzeuge sämtliche notwendigen Objekte, die für eine Verarbeitungslogik benötigt werden. Benutze dabei Meta-Informationen, um den Lebenszyklus der Objekte komplett kontrollieren zu können. Die erzeugten Objekte werden intern vom Context einem Scope zugeordnet, der die Gültigkeit und damit Lebensdauer des Objekts definiert.



- Benutze Dependency Injection, um Abhängigkeiten zwischen den Objekten aufzubauen. Zur Identifikation der Dependencies werden ebenfalls Meta-Daten benutzt. Damit wird aus den singulären Objekten ein komplexes Objekt-Geflecht.



- Stelle einen einfachen Zugriff auf exponierte Objekte dieses Geflechts zur Verfügung.



Damit sind im Context verschiedene GoF-Pattern im Einsatz: Der Context ist eine Factory für Objekte. Dependency Injection entstammt dem Strategy-Pattern, der gekapselte Zugriff auf das Objekt-Geflecht entspricht der Facade. Im Scope-Begriff verschwunden ist das Singleton-Pattern ebenso wie das Flyweight.

CDI-Frameworks sind schon lange im Einsatz:

- Schon die ursprüngliche J2EE kannte das Context-Konzept über JNDI. Allerdings war Dependency Injection zu dieser Zeit ungebräuchlich und musste deshalb bei Bedarf von den Designern einer Anwendung selbst „entdeckt“ werden.
- Seit 2003 hat die Spring-Community die Initialzündung für CDI gegeben und stellt auch heute noch ein hervorragendes und weit verbreitetes Open Source -Framework zur Verfügung.
- Mit Einführung der Version 5 kam 2006 CDI in der Java Enterprise Edition an. Sowohl JavaServer Faces mit den „Managed Beans“ und „Managed Properties“ als auch die Enterprise Java Beans mit beispielsweise den Annotationen `@EJB` und `@Stateless` definieren Dependency Injection und Scopes.
- 2009 wurde als Bestandteil der JEE 6[2] die nun tatsächlich schlicht „CDI“ genannte Spezifikation veröffentlicht. Damit wurde ein kompletter weiterer Satz von Annotationen eingeführt[3].

[2] Damit ist CDI genau genommen kein notwendiger Bestandteil der Java Enterprise Edition! CDI kann auch außerhalb eines Applikationsservers benutzt werden. Dies beweist beispielsweise die Referenz-Implementierung „Weld“ von JBoss.

[3] Damit stehen innerhalb eines Applikationsservers drei unterschiedliche CDI-Frameworks zur Verfügung: Managed Beans, EJBs und nun CDI. Es ist zu wünschen, dass dies in zukünftigen Versionen der Spezifikation konsistent vereinheitlicht wird.

3.3 Dekoration

Einige Frameworks erweitern die Arbeitsweise des Contexts noch um einen weiteren Punkt:

- Dekoriere die Objekte um erweiterte Funktionalität bzw. lasse die Objekte zusätzlich von einem Container kontrollieren.

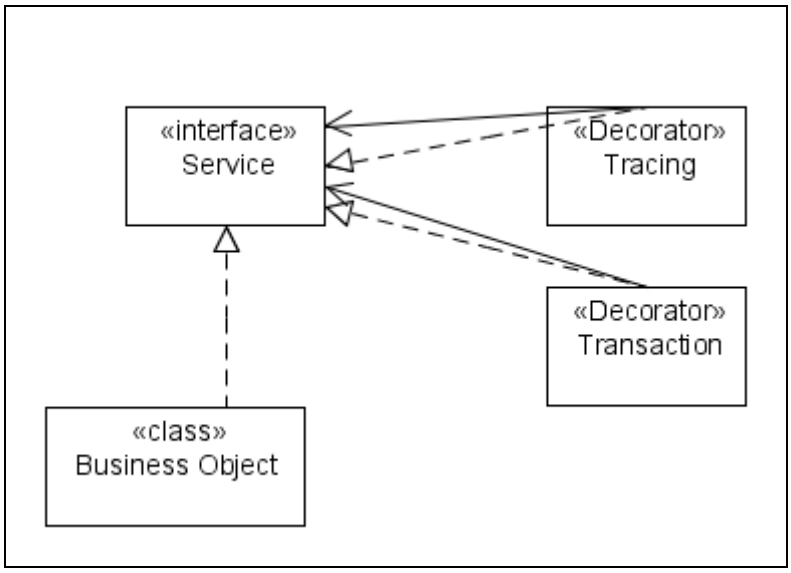
Dazu können verschiedene Techniken benutzt werden, durchaus auch sinnvoll in Kombination. Lassen sich diese Funktionalitäten verallgemeinern, so spricht man von so genannten Querschnittsfunktionen („Cross cutting concerns“).

Beispiele für diese „Zusatz“-Funktionen lassen sich in verschiedene Bereiche gruppieren:

- Fachliche Erweiterungen: So kann ein Business-Objekt auf dem Server als einfacher Daten-Container benutzt werden, auf Client-Seite werden im Rahmen einer MVC-Architektur nun jedoch zusätzlich Events geworfen, um verschiedene Views zu aktualisieren. Ebenso ist Validierung in bestimmten Umgebungen notwendig (Direkte Verarbeitung von Benutzer-Eingaben), in anderen jedoch nicht (Aufruf derselben Logik im Rahmen eines Business Prozesses, der die Validierung in einer vorgeschalteten Schritt bereits erledigt hat.
- Technische Aspekte wie Transaktionssteuerung und Authentifizierung.
- Erweiterungen während der Programmentwicklung bzw. in der Testphase: Hierfür kann das Tracen von Methodenaufrufen als Beispiel genommen werden, aber auch aufwändigere Aktionen wie eine automatisierte Bestimmung des verbrauchten Speichers.

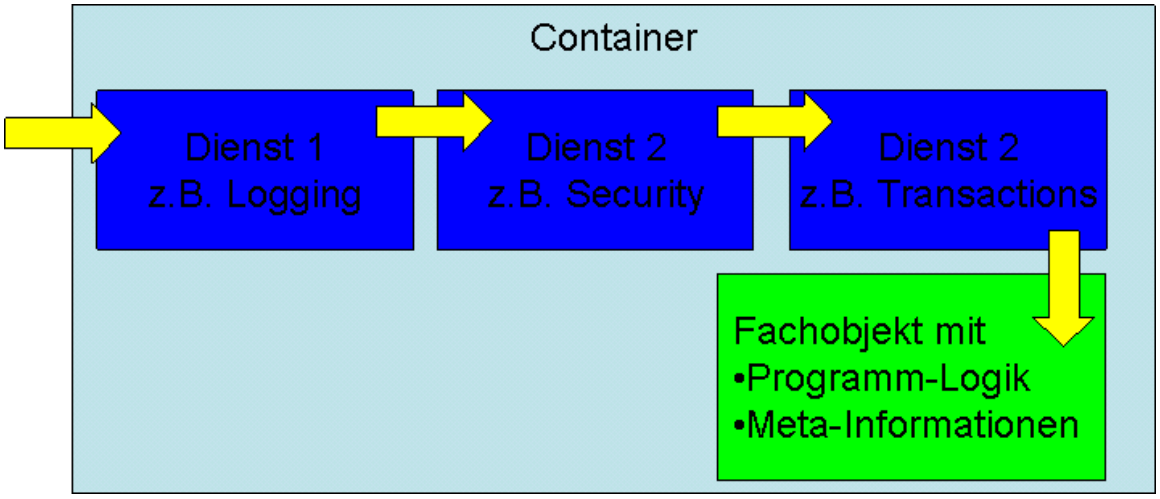
3.3.1 Business Object

Diese zusätzliche Funktionalität kann und wird in verschiedenen Situationen natürlich innerhalb der Fachklasse realisiert werden. Allerdings vermindert dieser Ansatz in der Regel die Wiederverwendbarkeit der Programme und deshalb wird ein anderer Ansatz bevorzugt: Die Dekoration der Fachklasse, die damit ohne Änderungen in verschienen Umgebungen benutzt werden kann.



3.3.2 Container

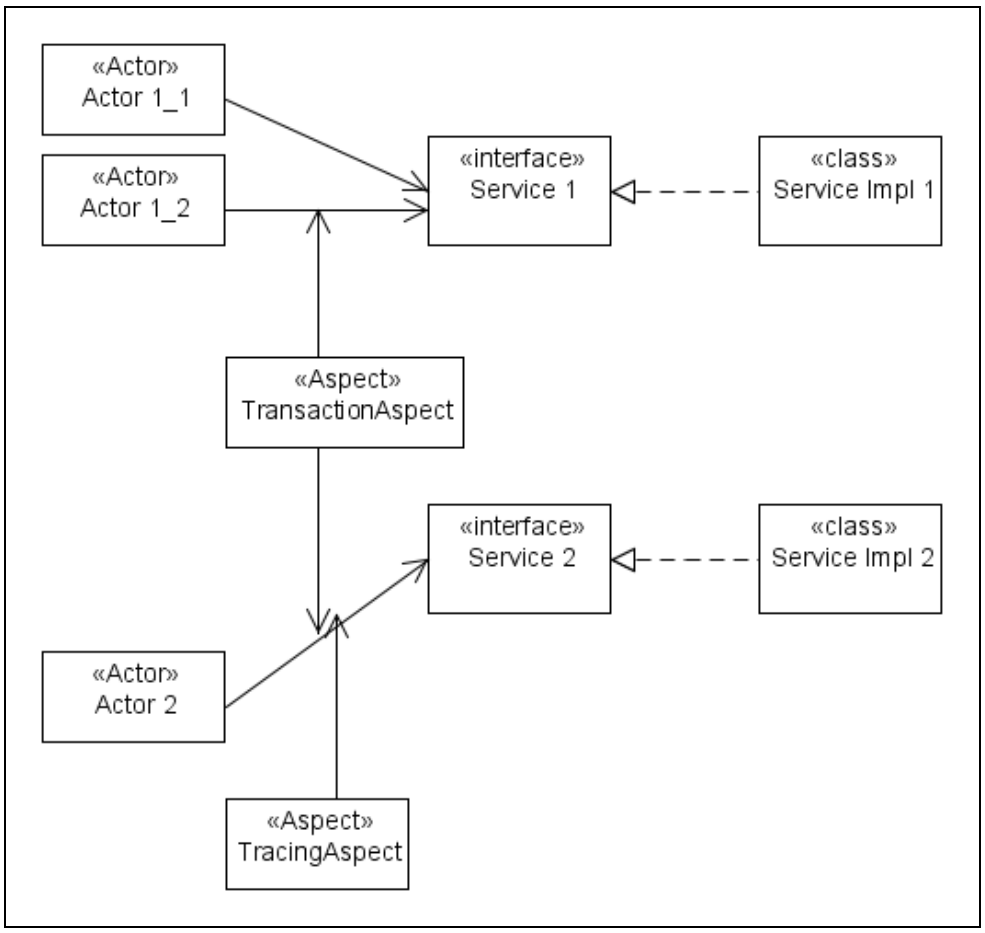
Statt für jede Zusatzfunktion einen eigenen speziellen Decorator zu realisieren kann die Fachklasse auch in einen Container installiert werden, der dann Dienste aufruft. Welche Dienste in welcher Konfiguration benutzt werden sollen werden durch Meta-Daten definiert:



Dieses Modell ist natürlich aus der Java Enterprise Edition bekannt. Die Meta-Daten entstammen aus Annotationen oder einer externen XML-basierten Konfiguration, dem EJB-Deskriptor.

3.3.3 Aspektorientierte Programmierung

Bei beiden vorherigen Beispielen war die Dekoration speziell auf das Fachobjekt zugeschnitten. Es gibt jedoch auch durchaus Situationen, in denen die Dekorationslogik Fach-übergreifend definiert werden kann. Dies war übrigens bereits im Tracing-Beispiel der Dekoration der Fall: Tracing ist im Gegensatz zum Logging allgemein verwendbar. In solchen Situationen



3.3.4 Das JEE Pattern Layer Supertype

Dieses Entwurfsmuster ist von Martin Fowler vorgeschlagen worden. Die Problemstellung lautet:

"JEE Komponenten werden in der Architektur an wohldefinierten Stellen eingesetzt und definieren damit Schichten der Anwendung. Aufgaben, die allgemein in diesen Schichten ausgeführt werden sollen, sollen einheitlich definiert sein".

Zur Lösung dieses Problems kann pro Schicht eine eigene Superklasse zur Verfügung gestellt werden, der Layer Supertype.

3.4 Realisierung

3.4.1 Stateless SessionBeans

Es stellt sich nun die Frage, wie die JEE benutzt werden kann, um ein Business Objekt in den Applikationsservers zu installieren. Dazu werden in den meisten Fällen Stateless SessionBeans benutzt. Diese haben die folgenden Eigenschaften bzw. realisieren die folgenden Dienste:

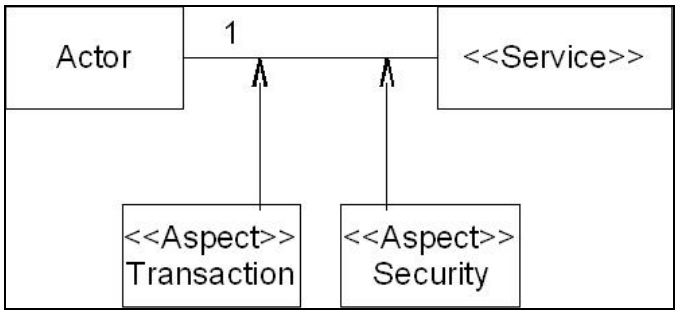
Dienst	**Anmerkung**
Remote Zugriff	Sinnvoll, falls die Architektur den Zugriff über RMI verlangt
Instance Pooling	Rein technisches Detail
Single Threaded Zugriff	Sinnvoll, falls die fachlichen Klassen nicht Thread-sicher konzipiert wurden
Zugriff auf Outbound Connectors	Sinnvoll, falls neben den trivialen `DataSource`s auch andere Backend-Systeme integriert werden sollen
Authentifizierung	Sinnvoll
Autorisierung	Sinnvoll, da eine Stateless SessionBean über den `SessionContext` stets den Prinzipal des Aufrufs aufrufen kann
Transaktionsmanagement	Sinnvoll sowohl in der Variante "Container Managed Transactions" mit deklarativer Transaktionssteuerung und "Bean Managed Transactions", in der die Bean über den `SessionContext` den Zugriff auf die `UserTransaction` ermöglicht.

Damit ist ersichtlich, dass die Kriterien für den Einsatz einer Stateless sessionBean zur Umhüllung eines fachlichen POJO sich in zwei Kategorie aufteilen:

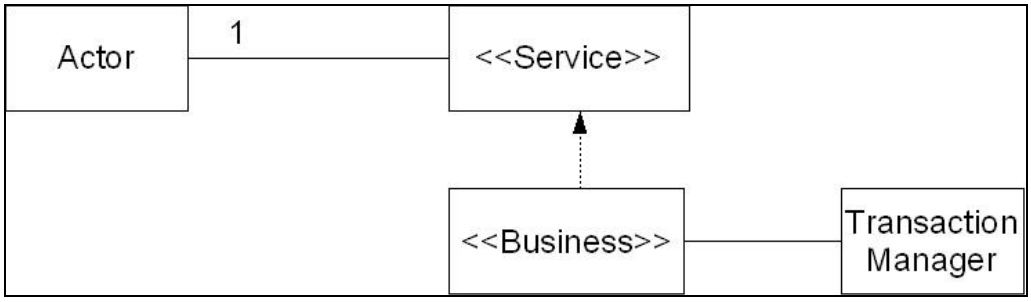
- Technisch: Zugriff über RMI
- Fachlich: Hier übernimmt die Stateless SessionBean bzw. genauer: Der SesionBean Container, Teile der fachlichen Aufgaben, nämlich insbesondere die Transaktionssteuerung.

Innerhalb eines Klassendiagramms erkennt man sinnvolle Einsatzbereiche sehr schnell, sobald die entsprechenden Aspekte oder Abhängigkeiten gefunden werden:

Hier ein Beispiel für eine Service-Benutzung, die deklarative Transaktionssteuerung und Authentifizierung verlangt:



In einem anderen Beispiel übernimmt die fachliche Implementierung die Transaktionssteuerung selber und benötigt dafür einen Transaktionsmanager:



In beiden Fällen ist der Einsatz einer Stateless SessionBean in Betracht zu ziehen.

3.4.2 Erweiterte Paketstruktur

Die Einführung der Session Facade führt zu einer Erweiterung der bisherigen Paketstruktur:

- `<domain>.<service>`
- `<domain>.<service>.value`
- `<domain>.<service>.test`
- `<domain>.<service>.business`
- `<domain>.<service>.aspects`
- **`<domain>.<service>.ejb`**

Darin befindet sich die Bean-Implementierung, also die Klasse mit den JEE-Annotationen.

3.5 Zusammenfassung

Stateless SessionBeans übernehmen innerhalb der JEE die Aufgabe, eine Komponente innerhalb des Applikationsservers installieren zu können. Sie fungieren hierbei als Decorator der Fachlogik. Diese Komponente enthält die Sequenzen der Anwendungslogik, kümmert sich hierbei jedoch nicht um Aufgaben, die der Applikationsserver übernehmen kann:

3.6 Beispiel: Eine SessionBean für den KeyGenerator

Es genügt die folgende Klasse:

```
@Stateless
public class KeyGeneratorEjb3SessionFacadeBean implements
KeyGenerator {


private RandomKeyGenerator randomKeyGenerator;


@PostConstruct
public void initFacade() {
 randomKeyGenerator = new RandomKeyGenerator();
}


public byte[] next() {
 System.out.println(this + " generates key...");
 return randomKeyGenerator.next().getBytes();
}


@Override
public String toString() {
 return "KeyGeneratorEjb3SessionBean@" + hashCode();
}



}
```