

6 Sessions

In den vorigen beiden Kapiteln haben wir die statischen Aspekte von Hibernate kennengelernt: die Abbildung von Objekten in Datenbanktabellen (Mapping) sowie die Konfiguration von Hibernate.

In diesem und dem nächsten Kapitel beschäftigen wir uns mit den dynamischen Aspekten von Hibernate. Dieses Kapitel beschreibt den Umgang mit den Hibernate-Sessions, das Caching und dynamisches Nachladen mit Proxys. Das nachfolgende Kapitel widmet sich der Formulierung von Abfragen (*Queries*).

6.1 Hibernate-Sessions

Wir greifen wieder unseren elektronischen Terminkalender als Beispiel auf (siehe Abb. 6–1).

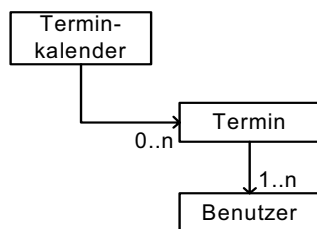


Abb. 6–1
Unser Beispiel

In Kapitel 4 haben wir gesehen, wie wir unsere Klassen Termin und Benutzer auf die Datenbank mappen. Kapitel 5 zeigte, wie man Hibernate konfiguriert, so dass es die richtigen Datenbankeinstellungen benutzt etc.

Um mit unseren persistenten Objekten umgehen zu können, benötigen wir eine Hibernate-Session. Wir haben bereits gesehen, dass die Session in Hibernate eine zentrale Bedeutung hat. Jeder Datenbankzugriff geht von einer Session aus. Die Session verwaltet die Zustände

der Geschäftsobjekte. Sie ist verantwortlich für deren Persistierung, das Laden von Objekten aus der Datenbank und das Löschen von solchen.

Wir können eine Session nicht selbst erzeugen, sondern müssen uns von einer SessionFactory eine neue Session geben lassen. Auch die SessionFactory können wir nicht direkt erzeugen. Sie wird uns von der Configuration zur Verfügung gestellt (siehe Abb. 6–2).

Abb. 6–2
Erzeugen einer Session



Der Quellcode zur Erzeugung einer Session sieht also wie folgt aus:

```

Configuration configuration =
    new Configuration().configure();
SessionFactory sessionFactory =
    configuration.buildSessionFactory();
Session session = sessionFactory.openSession();
  
```

*Configuration als
Wegwerfobjekt*

Die Configuration ist dabei i.d.R. ein Wegwerfobjekt: Man erzeugt sich beim Programmstart ein Configuration-Objekt, lässt sich von diesem eine oder mehrere SessionFactories erzeugen und verwirft das Configuration-Objekt schließlich. Die Configuration wird also meist nur in einer lokalen Variablen gehalten.

6.1.1 Geschäftsobjekte speichern, laden und löschen

Mit Hilfe der Session können wir unsere Geschäftsobjekte laden und speichern. Nachdem wir ein Geschäftsobjekt erzeugt haben, wird es mit save gespeichert.

```

Termin termin =
    new Termin(2005, 12, 24, "Heiligabend");
session.save(termin);
  
```

Damit hat Hibernate den Termin aber noch nicht zwangsläufig in die Datenbank geschrieben, und auf keinen Fall hat Hibernate die zugehörige Datenbanktransaktion geschlossen.

Um Veränderungen an der Datenbank vorzunehmen, muss man mit Transaktionen arbeiten. Dazu lässt man sich von der Session ein Transaktionsobjekt geben und beendet die Transaktion am Ende mit commit (Details zu Transaktionen in Hibernate finden sich in Kapitel 6.2):

```
Termin termin =
    new Termin(2005, 12, 24, "Heiligabend");
Transaction tx = session.beginTransaction();
session.save(termin);
tx.commit();
```

Das `commit` führt intern automatisch ein `flush` auf der `Session` aus, so dass die Daten im Cache zuerst in die Datenbank geschrieben werden.

Man kann die Daten auch explizit mit `flush` in die Datenbank schreiben. Das ist in der Regel aber unnötig.

Am Ende einer Datenbankaktion sollte man die `Session` wieder schließen. Das ist zwar nicht unbedingt notwendig, reduziert aber den Ressourcenverbrauch der Anwendung:

```
Session session = sessionFactory.openSession();
Termin termin =
    new Termin(2005, 12, 24, "Heiligabend");
Transaction tx = session.beginTransaction();
session.save(termin);
tx.commit();
session.close();
```

Die Methode `save` geht davon aus, dass das Geschäftsobjekt neu in die Datenbank geschrieben werden soll. Ruft man zweimal `save` für dasselbe Geschäftsobjekt auf, schreibt Hibernate das Objekt entweder nochmals in die Datenbank oder wirft eine `Exception`. Der erste Fall tritt auf, wenn Hibernate die IDs selbst vergibt. Der zweite Fall tritt auf, wenn die ID von der Anwendung vergeben wird.

Möchte man ein existierendes Objekt in der Datenbank aktualisieren, so muss man die Methode `update` verwenden.

```
session.update(termin);
```

Wenn man nicht genau weiß, ob das Geschäftsobjekt bereits gespeichert wurde, kann man sich das Leben mit der Methode `saveOrUpdate` einfach machen. Existiert das Geschäftsobjekt noch nicht in der Datenbank, speichert Hibernate es neu. Gibt es das Geschäftsobjekt bereits in der Datenbank, aktualisiert Hibernate es dort.

```
session.saveOrUpdate(termin);
```

Will man ein gespeichertes Objekt wieder einladen, so kann man das mit `get` oder `load` erledigen.

```
Termin termin1 =
    session.get(Termin.class, termin1ID);
Termin termin2 =
    session.load(Termin.class, termin2ID);
```

Der Unterschied zwischen `get` und `load` liegt in der Behandlung nicht-existierender Objekte. Existiert kein Objekt mit der angegebenen ID in der Datenbank, liefert `get` `null` zurück, während `load` eine Exception wirft. Wenn es möglich ist, dass das angeforderte Objekt nicht in der Datenbank existiert, sollte man `get` verwenden. `load` sollte nur dann verwendet werden, wenn das Fehlen des Objektes einen Systemfehler darstellt.

Das Einladen von Objekten mit Abfragen (*Queries*) wird in Kapitel 7 beschrieben und hier nicht weiter behandelt.

Das Löschen von Objekten erfolgt mit `delete` an der Session.

```
session.delete(termin);
```

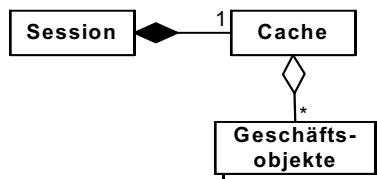
Das Objekt wird erst aus der Datenbank gelöscht, wenn die Transaktion beendet wird. Bemerkenswert ist, dass zum Löschen ein Geschäftsobjekt benötigt wird. In welchem Zustand sich das Geschäftsobjekt befindet, ist allerdings unerheblich. Es muss lediglich die ID des zu löschenden Objektes gesetzt sein.

6.1.2 Caching

Hibernate speichert persistente Objekte in einem Cache, der an die Session gebunden ist (siehe Abb. 6–3). Speichert man Objekte an der Session, werden diese erst einmal nur im Cache zwischengespeichert. Erst bei `flush` werden sie in die Datenbank geschrieben. Lädt man Objekte mit `get` oder `load`, sucht die Session das entsprechende Objekt erst im Cache. Nur wenn das Objekt dort nicht gefunden wurde, wird es in der Datenbank gesucht. Wird das Objekt in der Datenbank gefunden und eingeladen, wird es im Cache vermerkt. Beim nächsten Zugriff auf das Objekt wird die Suche im Cache schließlich erfolgreich sein.

Abb. 6–3

Cache an der Session



Das Caching erfolgt in Hibernate transparent. Der Entwickler muss sich darum i.d.R. keine Gedanken machen. Die Hibernate-Session setzt mit seinem Cache das Entwurfsmuster *Unit-of-Work* um (siehe [Fowler 02]), bei dem Änderungen an Geschäftsobjekten zunächst in einer transienten *Unit-of-Work* gesammelt und später in einer kurzen Transaktion persistiert werden.

Die Verwendung des Caches sorgt bei Hibernate-Neulungen manchmal für Überraschungen. Speichert man ein Objekt mit `save`, so erhält man das Objekt mit `get` oder `load` auch dann zurück, wenn die Transaktion vorher nicht geschlossen wird – `save` speichert das Objekt im Cache, und `get/load` sucht es dort.

6.1.3 Proxys

Caching optimiert den Schreibzugriff sowie den Zugriff auf bereits geladene Objekte. Das dynamische Nachladen über Proxys optimiert das Laden von Objekten.

Hibernate lädt in der Default-Einstellung ein angegebenes Objekt *flach*. Das bedeutet, dass alle an dem Objekt hängenden Objekte zunächst nicht eingeladen werden. Sie werden erst nachgeladen, wenn sie auch tatsächlich benötigt werden (*Lazy-Load* oder *Load-On-Demand*). Dass ein Objekt benötigt wird und daher nachgeladen werden muss, stellt Hibernate am Attributzugriff fest.

Dynamisches Nachladen

Eine solche Funktionalität lässt sich mit Java-Bordmitteln nicht herstellen. Daher arbeitet Hibernate mit der CGLIB, die Bytecode-Generierung zur Laufzeit erlaubt. Mit Hilfe der CGLIB generiert Hibernate zur Laufzeit Subklassen der persistenten Geschäftsobjekt-Klassen. Diese Subklassen sind schlicht *Proxys* (siehe [Gamma et al. 05]), die lediglich dazu dienen, das Nachladen von Objekten zu initiieren. Die generierten Proxyklassen implementieren zusätzlich das Interface `HibernateProxy`.

Proxys mit der CGLIB

Aus der Proxy-Konstruktion folgen einige Konsequenzen für den Entwurf persistenter Geschäftsobjekte:

Wichtige Richtlinien

- Persistente Geschäftsobjekt-Klassen dürfen nicht *final* sein.
- Methoden der persistenten Geschäftsobjekt-Klassen dürfen nicht *final* sein.
- Harmlos scheinende Zugriffe auf Attribute (z.B. bei `equals` oder `hashCode`) können relativ aufwändige Lade-Operationen auslösen.
- Man sollte auf Attribute eines anderen Geschäftsobjektes immer nur über Methoden (Setter und Getter) zugreifen und niemals direkt (auch nicht in Copy-Konstruktoren, `equals` und `compareTo`). Bei direktem Zugriff kann der Proxy den Zugriff nicht erkennen und das Nachladen nicht initiieren – man bekommt dann einfach Default-Werte, die beim nächsten Nachladen überschrieben werden.
- Man sollte nicht auf private Methoden eines anderen Geschäftsobjektes zugreifen (vor allem bei Copy-Konstruktoren, `equals` und `compareTo` relevant), weil diese über die CGLIB in der Proxyklasse nicht redefiniert werden können.

Proxys ohne Session

Wird ein Objekt von seiner Session abgekoppelt (Session geschlossen, Objekt serialisiert, Aufruf von `Session.evict()`), werden die Proxys nicht aufgelöst. Ein Zugriff auf einen Proxy liefert dann eine `LazyInitializationException`.

Mit `Hibernate.initialize` kann der Proxy manuell aufgelöst werden: Er bekommt dann alle seine Attribute gesetzt (allerdings nicht rekursiv). Mit `Hibernate.isInitialized` kann festgestellt werden, ob ein Proxy initialisiert wurde.

6.1.4 Lebensdauer von Sessions

Wenn eine Session eine Exception wirft, kann man über ihren Zustand keine sicheren Aussagen mehr machen. In der Regel kann man mit der Session nicht mehr viel anfangen.

Das bedeutet, dass man Sessions nur kurzzeitig erzeugen und nicht in Attributen speichern sollte (bzw. man muss bei Exceptions sofort reagieren und die Session schließen). In vielen Fällen kann man die Session einfach als Transaktion begreifen und genauso auch verwenden.

```
Session session = _sessionFactory.openSession();
Termin termin =
    new Termin(2005, 12, 24, "Heiligabend");
Transaction tx = session.beginTransaction();
session.save(termin);
tx.commit();
session.close(termin);
session = null;
```

*Schließen von
SessionFactory*

Man kann nicht nur eine Session schließen, sondern auch die `SessionFactory`. Es liegt in der Verantwortung der Anwendung, dass vorher alle von der `SessionFactory` erzeugten Sessions ordnungsgemäß geschlossen wurden. Nach dem Schließen einer `SessionFactory` befinden sich die Sessions in einem undefinierten Zustand und sollten nicht weiter verwendet werden.

6.2 Transaktionen

Wie bereits gesehen, müssen wir explizit mit Transaktionen arbeiten, wenn wir persistente Daten verändern wollen. Man lässt sich ein Transaktionsobjekt von der Session geben und beendet sie mit `commit`.

```
Transaction transaction =
    session.beginTransaction();
Termin termin = session.get(Termin.class, terminID);
termin.setBemerkung("Oma kommt auch");
transaction.commit();
session.close();
```

Man kann mit dem Transaktionsobjekt nicht nur Daten in die Datenbank schreiben, sondern die Transaktion per `rollback` auch abbrechen. Dann werden gar keine Daten in die Datenbank geschrieben.

```
Transaction transaction =
    session.beginTransaction();
Termin termin = session.get(Termin.class, terminID);
termin.setBemerkung("Oma kommt auch");
if (irgendeineBedingung) {
    transaction.commit();
}
else {
    transaction.rollback();
}
session.close();
```

Die Transaktionsobjekte haben in Hibernate einen Verweis auf die erzeugende Session (siehe Abb. 6–4). Das hat einen überraschenden Effekt: Die Methode `beginTransaction` liefert zwar immer unterschiedliche Transaktionsobjekte, die aber auf dieselbe Datenbanktransaktion verweisen. Wurde mit `beginTransaction` bereits eine Transaktion erzeugt und mit `begin` begonnen, kann man sich mit einem erneuten `beginTransaction` diese Transaktion besorgen und ohne zusätzliches `begin` mit ihr arbeiten.

Geschachtelte Transaktionen werden nicht unterstützt. Wir empfehlen, maximal ein Transaktionsobjekt von einer Session erzeugen zu lassen. Alles andere kann beim Lesen des Quellcodes extrem verwirren. Immerhin wird eine Exception geworfen, wenn man `begin` an einer bereits begonnenen Transaction aufruft. Damit fallen die meisten Fehlbenutzungen schnell auf.

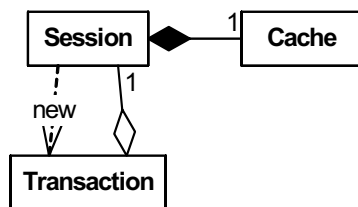


Abb. 6–4

Session, Cache und Transaction

Möchte man mehrere »echte« Transaktionen öffnen, muss man mehrere Sessions erzeugen. Diese laufen vollkommen isoliert voneinander ab. Jede Session hat ihren eigenen Cache. Über eine Session gespeicherte Objekte können über die andere Session nur dann geladen werden, wenn auf der ersten Transaktion `commit` ausgeführt wurde. Es macht für Hibernate keinen Unterschied, ob zwei Sessions auf einem Rechner im selben Prozessraum ablaufen oder auf verschiedenen Rechnern.

Mehrere Sessions

Transaktion Hibernate unterstützt verschiedene Implementierungen von Transaktionen. Im einfachsten Fall greift Hibernate intern auf eine `JDBCTransaction` zurück, die den Transaktionsmechanismus der Datenbank benutzt. Es ist auch möglich, mit *JTA (Java Transaction Architecture)* den Transaktionsmechanismus eines Application Servers zu benutzen. Details zur Anbindung über JTA finden sich in Kapitel 5 (*Konfiguration*).

Schnittstelle Transaction Die Schnittstelle `Transaction` ist in Hibernate denkbar einfach gehalten. Eine Transaktion ist gestartet, wenn sie von der Session erzeugt wurde. Man kann eine Transaktion danach mit `commit` oder mit `rollback` abschließen.

*Beispiel:
Transaktion abbrechen*

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
session.save(termin1);
tx.rollback(); // Transaktion abbrechen
```

*Beispiel:
Transaktion abschließen*

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
session.save(termin1);
tx.commit(); // Transaktion persistieren
```

*Transaktionen
wiederverwenden* Wenn man eine Transaktion mit `rollback` abbricht oder mit `commit` persistiert, kann man sie erst nach Aufruf von `begin` weiterverwenden¹. Die Transaktion wird vollständig unbrauchbar, wenn die zugehörige Session mit `close` geschlossen wird.

*Transaktionen
beeinflussen die
Session nicht.* Transaktionen haben keinen direkten Einfluss auf die Session und die an die Session gebundenen Objekte. Speichert man über die Session ein Objekt und ruft dann an der Transaktion `rollback` auf, dann wird `get` an dieser Session das Objekt ohne Zugriff auf die Datenbank zurückliefern. Da das Objekt nicht in die Datenbank geschrieben wurde, würde eine andere Session mit `get` das Objekt gar nicht oder in einem anderen Zustand bekommen (je nachdem, ob das Objekt vorher bereits persistent war). Im folgenden Codebeispiel liefert `Session.get` das an `meinTermin` gebundene Objekt zurück, während andere Sessions das Objekt gar nicht bekommen können – es wurde ja nie in der Datenbank geschrieben.

```
Termin meinTermin = new Termin();
session.save(meinTermin);
transaction.rollback();
Termin t =
    session.get(Termin.class, meinTermin.id());
```

1. In früheren Hibernate-Versionen war der Aufruf von `begin` nicht notwendig. Nach `rollback/commit` konnte die Transaktion einfach weiterverwendet werden.

Transaktionsisolation

Über die Konfiguration mit dem Property *hibernate.connection.isolation* erlaubt Hibernate die Einstellung der Transaktionsisolation (siehe Kapitel 2.4 sowie Kapitel 5 für die konkrete Konfiguration in Hibernate). Folgende Werte sind möglich: *Read Uncommitted*, *Read Committed*, *Repeatable Read*, *Serializable*.²

Für die meisten Anwendungen stellt *Read Committed* einen guten Kompromiss aus Sicherheit und Performance dar. Da Hibernate Änderungen an Objekten im Cache vermerkt und beim commit komplett speichert, hält Hibernate Transaktionen nur kurze Zeit geöffnet. Daher würde *Read Uncommitted* selten Daten liefern, die noch nicht committed wurden. Manchmal kann dieser Isolation Level eine Möglichkeit darstellen, den Datendurchsatz auf der Datenbank zu erhöhen.

Repeatable Read kann z.B. für die Erzeugung von Drucklisten sinnvoll eingesetzt werden, wenn dieselbe Abfrage mehrfach ausgeführt wird und immer das gleiche Ergebnis herauskommen muss. Solche anspruchsvollen Drucklisten werden daher häufig nachts erstellt, wenn keine oder nur wenige parallele Anwender mit der Datenbank arbeiten.

Serializable ist geeignet, wenn es um umfassende Aufgaben geht, wie z.B. komplette Reorganisationsläufe auf der Datenbank, bestimmte Arten von Datensicherungen etc.

Die Isolationslevel entsprechen genau denen der *java.sql.Connection*. Falls die *Connection* von einer *DataSource* innerhalb eines *Application Servers* bezogen wird, hat diese Einstellung keine Änderung des Levels zur Folge. In diesem Fall muss der gewünschte Transaktionsisolationlevel bei der Erstellung der *DataSource* eingestellt werden. Der Level lässt sich im Nachhinein nicht mehr ändern.

6.3 Das Hibernate-Zustandsmodell persistenter Objekte

Zusätzlich zu den fachlichen Zuständen eines Geschäftsobjektes verwaltet Hibernate seinen Persistenzzustand. So wird explizit gemacht, dass der Zustand eines Objektes im Hauptspeicher nicht immer synchron mit dem Zustand der zugehörigen Tabelleneinträge in der Datenbank ist. Der Persistenzzustand ist kein Attribut der Geschäftsobjekte. Er wird von der *Session* verwaltet.

Ein neues Objekt wird mit *new* zunächst nur im Hauptspeicher erzeugt. Es gibt zu diesem Zeitpunkt noch keine zugehörigen Tabellen-

Zustand »Transient«

2. Die Bedeutung der einzelnen Isolation Levels ist in Kapitel 2.4 erläutert.

einträge in der Datenbank. Das neu erzeugte Objekt im Hauptspeicher befindet sich im Zustand *Transient*.

Zustand »Persistent«

Wenn das Objekt gespeichert wird, bekommt es einen eindeutigen Primärschlüssel (ID) zugewiesen und geht in den Zustand *Persistent* über. Befindet sich ein Objekt im Zustand *Persistent*, bedeutet das zunächst nur, dass Hibernate das Objekt beim nächsten `Session.flush` in die Datenbank schreibt.

Das bedeutet also, dass der Zustand eines persistenten Objektes vom aktuellen Datenbankzustand abweichen kann – z.B. weil ein anderer Benutzer das Objekt in der Datenbank verändert hat. Solange sich das Objekt im Zustand *Persistent* befindet, wirken sich Änderungen am Objekt direkt auf die Datenbank aus (jeweils bei `Session.flush`). Sie müssen nicht explizit mit `save` gespeichert werden. Natürlich werden die Änderungen an der Datenbank dadurch nicht automatisch committed. Dazu ist auf jeden Fall das Beenden der Transaktion notwendig.

Wird ein Objekt aus der Datenbank geladen, befindet es sich auch direkt im Zustand *Persistent*. Die letzte Möglichkeit, ein transientes Objekt in den Zustand *Persistent* zu überführen, besteht darin, eine Referenz von einem persistenten Objekt auf das transiente Objekt anzulegen.

Ein Objekt geht vom Zustand *Persistent* wieder zurück in den Zustand *Transient*, wenn es an der Session gelöscht wird. Jetzt herrscht wieder der Ausgangszustand: Es gibt ein Geschäftsobjekt im Hauptspeicher ohne zugehörige Tabelleneinträge in der Datenbank.

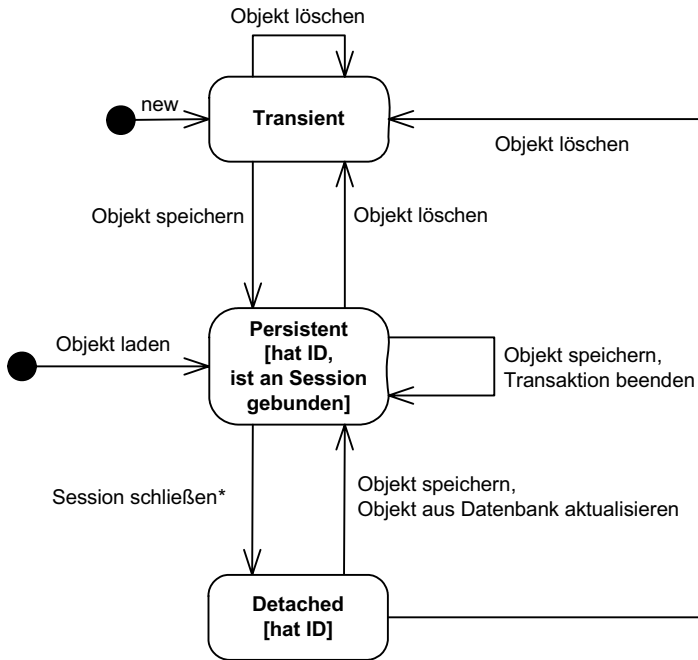
Zustand »Detached«

Ein persistentes Objekt wird von der Session abgekoppelt (*detached*), wenn die Session geschlossen wird. Das Geschäftsobjekt befindet sich dann im Zustand *Detached*. Das bedeutet, dass es zu dem Objekt zugehörige Tabelleneinträge in der Datenbank gibt, dass Änderungen am Objekt aber nicht automatisch in die Datenbank übernommen werden. Das passiert erst, wenn das Objekt wieder an die Session gebunden wird (z.B. mit `save`).

Kein Lazy-Load bei
»Detached«

Das Lazy-Load von Hibernate ist nur für Objekte im Zustand *Persistent* aktiv. So kann es passieren, dass man für Objekte im Zustand *Detached* auf ungültige Referenzen läuft: Man will auf ein Objekt zugreifen, das noch gar nicht nachgeladen wurde. Man muss also vor dem Detachen sicherstellen, dass alle Bestandteile des Objekts, die man benötigt, auch da sind. Mit der statischen Methode `initialize` an der Klasse `Hibernate` kann man das Nachladen einzelner Objekte erzwingen. Mit `Hibernate.isInitialized` kann geprüft werden, ob ein Objekt bereits nachgeladen wurde.

Das Hibernate-Zustandsmodell persistenter Objekte ist in Abbildung 6-5 anhand eines Zustandsübergangsdiagramms visualisiert.



* betrifft alle Objekte einer Session

Abb. 6-5

Zustandsmodell
persistenter Objekte

Ein Objekt ist dann an eine Session gebunden, wenn es sich im Zustand *Persistent* befindet. Ein Objekt wird an eine Session gebunden, wenn das Objekt über die Session geladen oder gespeichert wird. Die Verbindung wird wieder gelöst, wenn die Session geschlossen wird. Während ein Objekt an eine Session gebunden ist, kann es nicht an eine andere Session gebunden werden. Es ist also nicht möglich, ein Objekt über eine Session zu laden und über eine andere Session wieder zu speichern, solange die erste Session noch offen ist.

Die folgende Tabelle zeigt an unserem Terminplaner-Beispiel die Zustandsübergänge von Geschäftsobjekten. Für das Beispiel nehmen wir an, dass in den Mapping-Dateien vollständige Kaskadierung für Termine und Benutzer eingestellt ist (fachlich wird man die Kaskadierung für das Löschen natürlich abschalten – das Löschen eines Termins soll ja nicht die angehängten Benutzer mit löschen).

Code	Zustand Termin	Zustand Teilnehmer
Termin termin = new Termin(2006, 01, 01, »Neujahr«);	transient	–
Benutzer sr = new Benutzer(»sr«, »Stefan«, »Roock«);	transient	transient
session1.save(termin);	persistent	transient
termin.fuegeTeilnehmerHinzu(sr);	persistent	persistent
session1.close();	detached	detached
...		
session2.saveOrUpdate(termin);	persistent	persistent
session2.close();	detached	detached
...		
session3.delete(termin);	transient	transient
session3.close();	transient	transient
...		

6.4 Configuration, SessionFactory, Session, Transaction im Konzert

Die Configuration bezieht sich i.d.R. auf eine Anwendung. Sie wird meist nur als Wegwerfobjekt bei Programmstart erzeugt, um eine SessionFactory zu generieren. Die SessionFactory stellt die Verbindung zur Datenbank her. Es muss ein SessionFactory-Objekt je Datenbank existieren. Die meisten Anwendungen arbeiten auf einer Datenbank und benötigen daher auch nur ein SessionFactory-Objekt. Die Session existiert je »Arbeitszusammenhang« und die Transaction je atomare Datenbankänderung.

In unserem Terminplaner arbeiten wir mit nur einer Datenbank. Wir benötigen also nur eine Configuration und nur eine SessionFactory. Unsere Anwendung besteht aus einer Menge von Werkzeugen, von denen mehrere gleichzeitig aktiv sein können. Jedes Werkzeug hält eine Referenz auf die SessionFactory als Quelle für die benötigten Sessions. Bei jeder Interaktion mit der Datenbank erzeugen sich die Werkzeuge neue Sessions. Transaktionen werden nicht benötigt, weil kein explizites rollback erforderlich ist. Abbildung 6–6 zeigt die Struktur an einem Objektdiagramm. Die TerminplanerAnwendung erzeugt ein Configuration- und ein SessionFactory-Objekt. Mit der SessionFactory parametrisiert die TerminplanerAnwendung die beiden Werkzeuge TerminEditor und TerminUebersicht bei ihrer Erzeugung. Jedes Mal, wenn TerminEditor oder TerminUebersicht auf die Datenbank zugreifen, erzeugen sie sich neue temporäre Sessions, die sie nach dem Datenbankzugriff gleich wieder schließen.

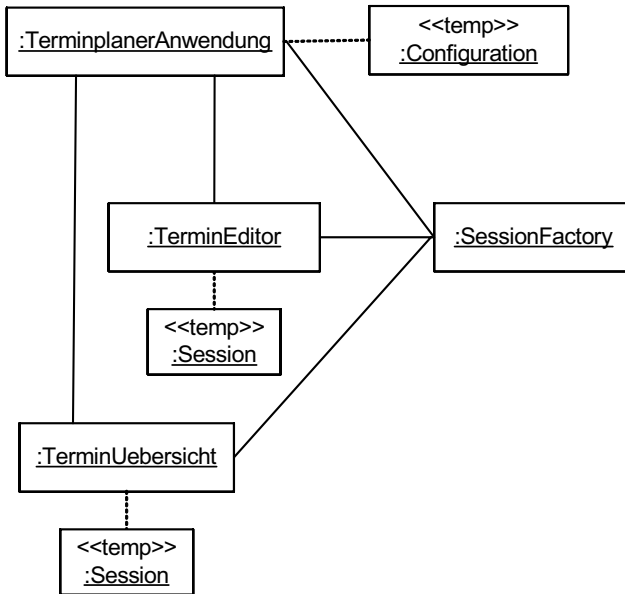


Abb. 6-6

Objektdiagramm:
Sessions im Terminplaner

In unserem Beispiel erzeugen die Werkzeuge neue Termine wie gewohnt mit `new` und speichern sie mit `saveOrUpdate`. Nach dem Speichern wird die `Session` sofort geschlossen, wodurch die Geschäftsobjekte in den Zustand *Detached* übergehen. Nimmt der Anwender weitere Änderungen an dem Geschäftsobjekt vor, wird es erneut mit `saveOrUpdate` gespeichert und durch das Schließen der `Session` wieder in den Zustand *Detached* überführt.

6.5 Persistenz über Erreichbarkeit

Hibernate setzt *Persistenz über Erreichbarkeit* (*Persistence by Reachability*) um. Wenn ein Objekt gespeichert wird, werden die referenzierten Objekte ebenfalls gespeichert. Wenn ein Objekt geladen wird, werden die referenzierten Objekte auch geladen.

Anwendungsentwickler müssen sich also nicht um das Speichern oder Laden referenzierter Objekte kümmern. Man spricht auch von *transparenter Persistenz*, weil das Speichern und Laden referenzierter Objekte für den Anwendungsentwickler transparent erfolgt.

In Kapitel 12 wird beschrieben, wie das Kaskadieren beim Speichern konfiguriert werden kann. So können wir bei Bedarf noch Einfluss nehmen und müssen nicht immer das volle Erreichbarkeitsmodell verwenden. Im Rahmen der Kaskadierung beim Speichern gibt es eine zusätzliche Möglichkeit, ein transientes Objekt in den Zustand *Per-*

Transparente Persistenz

Kaskadierung
beim Speichern

sistent zu überführen. Es reicht aus, wenn man es von einem persistenten Objekt aus referenziert.

Lazy Loading

Beim Einladen verwendet Hibernate standardmäßig *Lazy Loading* (auch *Load on Demand*). Wird ein Objekt eingeladen, werden nicht sofort alle Objekte eingeladen, die direkt oder indirekt vom geladenen Objekt referenziert werden. Das könnte sehr lange dauern und sehr viel Speicherplatz beanspruchen. Häufig benötigt man die ganzen referenzierten Objekte in einem bestimmten Arbeitsschritt auch gar nicht. Also wird nur das Wurzelobjekt geladen und die referenzierten Objekte erst, wenn sie benötigt werden.

1+n-Problem

Beim schrittweisen Nachladen entsteht häufig das sogenannte 1+n-Problem. Nehmen wir in unserem Beispiel an, wir wollen eine Liste aller Termine mit Teilnehmern anzeigen. Dann würde Hibernate zuerst die Termine liefern, ohne gleich die Teilnehmer mit einzuladen. Beim Anzeigen der Termine würde Hibernate für jeden Termin einzeln die Referenzen auf die Teilnehmer auflösen und die Teilnehmer nachladen. Man benötigt also *eine* Datenbankabfrage, um die Termine einzuladen, und *n* Datenbankabfragen, um die Teilnehmer nachzuladen. In einem Beispiel mit 1.000 Terminen würde Hibernate 1.001 Datenbankabfragen durchführen.

Es ist offensichtlich, dass hier ein großes Performance-Problem liegen kann. Kapitel 11 klärt diese und andere Performance-Fragen im Detail.

6.6 Sperren

RDBMS-
Standardverhalten:
Der Letzte gewinnt.

In Kapitel 2 haben wir gesehen, dass Transaktionen konsistente Datenänderungen garantieren. Allerdings bezieht sich die Konsistenzforderung im Wesentlichen auf die technischen Aspekte der Datenhaltung. In interaktiven Anwendungen können zwei Anwender denselben Datensatz gleichzeitig bearbeiten und dann speichern. Der zweite Anwender würde die Änderungen des ersten Anwenders überschreiben. Die Situation ist in der folgenden Tabelle dargestellt.

Anwender 1	Anwender 2
Einladen Termin A (SELECT)	
Anzeigen des Termins an der Benutzungsoberfläche zum Bearbeiten durch Anwender 1	Einladen Termin A (SELECT)
Speichern Termin A (UPDATE)	Anzeigen des Termins an der Benutzungsoberfläche zum Bearbeiten durch Anwender 1
	Speichern Termin A (UPDATE)

Das Problem tritt natürlich nur dann auf, wenn das Einladen und das Speichern des Termins in unterschiedlichen Transaktionen ausgeführt werden. Prinzipiell ist es auch möglich, Einladen und Speichern in *einer* Transaktion (mit ausreichend hohem Isolation Level) auszuführen. Allerdings ist nicht vorhersehbar, wie lange der Anwender zum Bearbeiten des Termins benötigt. Man würde also mit *langen Transaktionen* arbeiten. Davon raten die Lehrbücher für RDBMS aber mit gutem Grund ab: Der Verwaltungsaufwand für lange Transaktionen wird in der Datenbank sehr groß, so dass die Datenbank-Performance insbesondere bei vielen parallelen Benutzern darunter leiden kann.

Lange Transaktionen

Dass der Letzte gewinnt, ist auch genau das Standardverhalten von Hibernate. Zwei Sessions können parallel dasselbe Objekt laden, verändern und wieder speichern. Das zweite Speichern überschreibt die Änderungen des ersten Speicherns.

Hibernate-

Standardverhalten:

Der Letzte gewinnt.

Dieses Verhalten ist für viele Anwendungssysteme nicht wünschenswert. Also reichen die Datenbanktransaktionen für viele Anwendungssysteme nicht aus. Man arbeitet zusätzlich zu Transaktionen mit *Sperren*, die versehentliches Überschreiben von Änderungen anderer Anwender verhindern. Meistens sperrt man auf Ebene von Datensätzen bzw. Objekten. Es werden pessimistische und optimistische Sperrstrategien³ unterschieden. Die pessimistische ist die klassische Sperrstrategie.

Sperren

Bei der pessimistischen Sperrstrategie wird das Objekt gesperrt, sobald ein Anwender mit der Bearbeitung beginnt. Andere Anwender können dieses Objekt erst wieder bearbeiten, wenn der erste Anwender durch Speichern oder Abbruch der Bearbeitung das Objekt wieder *entsperrt*. Meistens können andere Anwender gesperrte Objekte jedoch lesen. Sie sehen dann den Objektzustand, der in der Datenbank gespeichert ist. Die Änderungen des bearbeitenden Anwenders bleiben zunächst unsichtbar.

Pessimistische

Sperrstrategie

Die optimistische Sperrstrategie ist eigentlich falsch benannt, weil sie ohne Sperren arbeitet. Beliebige viele Anwender können parallel das gleiche Objekt bearbeiten. Es kann jedoch nur der erste Anwender das Objekt speichern. Alle nachfolgenden Speicherversuche führen zu einer Fehlermeldung.

Optimistische

Sperrstrategie

Auf den ersten Blick erscheint die pessimistische Sperrstrategie die einzig vernünftige zu sein. Sie ist die einzige Variante, die sicher vor Datenverlust schützt. Schließlich können bei der optimistischen Sperr-

Vergleich

3. Die optimistische Sperrstrategie ist etwas eigenartig benannt, weil sie ganz ohne Sperren arbeitet. Allerdings hat sich der Begriff eingebürgert, und er macht die gegensätzlichen Ansätze des pessimistischen und optimistischen »Sperrens« plastisch.

strategie bei paralleler Bearbeitung eines Objektes auch Daten verloren gehen – nämlich immer dann, wenn eine Session versucht, ein Objekt zu speichern, das bereits durch eine andere Session gespeichert wurde.

Allerdings haben auch pessimistische Sperrstrategien erhebliche Nachteile. Wenn ein Anwender ein Objekt für die Bearbeitung sperrt und dann in den Urlaub fährt, werden seine Kollegen mitunter erheblich in ihrer Arbeit behindert. Stürzt der PC eines Anwenders ab, der Objekte gesperrt hat, tritt das gleiche Problem auf. Wenn zigtausend Anwender über das Internet mit der Anwendung arbeiten, können gesperrte Objekte das ganze System lahmlegen. Und nicht zuletzt sind pessimistische Sperren aufwändiger zu programmieren und weniger performant als optimistische Sperren (pessimistische Sperren brauchen mehr Datenbankzugriffe als optimistische Sperren).

Ein unbestreitbarer Vorteil pessimistischer Sperren für den Anwender ist die Tatsache, dass er vor der Bearbeitung informiert wird, dass das angeforderte Objekt bereits von einem anderen Anwender bearbeitet wird.

Daher haben sowohl optimistische wie auch pessimistische Sperrstrategien ihre Daseinsberechtigung. Welche man wählt, hängt vom Anwendungssystem ab. Wenn viele anonyme Anwender mit dem System arbeiten (z.B. über das Internet) oder Konfliktsituationen selten auftreten, sind optimistische Sperrstrategien geeignet. Wenn wenige In-House-Anwender mit dem System arbeiten und Konfliktsituationen häufig auftreten, sind pessimistische Strategien besser geeignet.

In Hibernate kann man das Sperrverhalten anpassen und damit sowohl pessimistische wie auch optimistische Sperrstrategien umsetzen.

*Pessimistische Sperren
in Hibernate*

Das pessimistische Sperren setzt Hibernate über die Mechanismen der Datenbank um (mit `SELECT FOR UPDATE`). Im Programmtext fordert man eine Sperre für ein Objekt entweder gleich beim Laden eines Objektes an oder nachdem man das Objekt geladen hat. Dabei werden drei verschiedene Sperrmodi unterschieden, die in der Klasse `LockMode` definiert sind.

1. `LockMode.READ`: Eine *Lesesperre* zeigt an, dass das Programm ein Objekt nur lesen möchte, aber sichergestellt werden muss, dass sich das Objekt eine Zeit lang nicht ändert. Das bedeutet, dass mehrere Anwender gleichzeitig eine Lesesperre besitzen können. Existieren jedoch Lesesperren auf einem Objekt, kann niemand eine Schreibsperre bekommen.
2. `LockMode.UPGRADE`: Eine wartende Schreibsperre bekommt das Programm nur, wenn keine andere Lese- oder Schreibsperre existiert. Wenn bei Anforderung der Sperre bereits eine Lese- oder Schreib-

sperre existiert, wartet Hibernate (bzw. die Datenbank), bis die Sperren aufgehoben werden.

3. `LockMode.UPGRADE_NOWAIT`: Eine nicht wartende Schreibsperre bekommt das Programm nur, wenn keine andere Lese- oder Schreibsperre existiert. Wenn bei Anforderung der Sperre bereits eine Lese- oder Schreibsperre existiert, wirft Hibernate eine `LockAcquisitionException`.

Pessimistische Sperren sind in Hibernate nur innerhalb einer Transaktion gültig. Wird die Transaktion beendet, verschwinden auch die Sperren. Möchte man »persistenter« Sperren haben, muss man die selbst implementieren, z.B. indem man jedem Geschäftsobjekt ein persistentes Zusatzattribut `istGesperrt` spendiert.

Optimistische Sperren werden in Hibernate meist mit Versionsnummern realisiert (siehe auch Kapitel 4: *Mapping*). Vor dem Speichern wird zuerst geprüft, ob die Versionsnummer im Objekt mit der Versionsnummer in der Datenbank übereinstimmt. Nur wenn das der Fall ist, wird die Versionsnummer im Objekt erhöht und das Objekt dann gespeichert.

*Optimistische Sperren
in Hibernate*

Die Versionsnummer muss explizit im Objekt als Attribut definiert werden und in der Mapping-Definition als Versionsnummer deklariert werden. Für unser Beispiel wollen wir eine optimistische Strategie über Versionsnummern umsetzen. Das folgende Beispiel zeigt Ausschnitte aus der Klasse `Termin` und der zugehörigen Mapping-Datei:

```
public class Termin {
    private int _version;

    public int getVersion() {
        return _version;
    }

    public void setVersion(int version) {
        _version = version;
    }
    ...
}

<class name="Termin" dynamic-update="true">
    <id name="id">
        <generator class="native"/>
    </id>

    <version name="version"/>
    <property name="titel"/>
    ...
</class>
```

Tiefe und flache Sperren

Sowohl bei pessimistischen wie auch bei optimistischen Sperren wird flach und nicht tief gesperrt. Bei einer Enthaltenseinbeziehung ist es möglich, dass eine Session das Vaterobjekt sperrt und gleichzeitig eine andere Session ein enthaltenes Objekt. In unserem Beispiel kann also eine Session einen Benutzer sperren und eine andere Session einen seiner Termine.

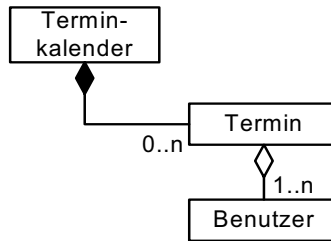
Tiefes Sperren ist aus fachlichen Gründen häufig gewünscht, aber teuer, was die Laufzeit angeht. In relevanten Anwendungssystemen müsste man schnell mehrere hundert oder tausend Objekte sperren. Sinnvollerweise löst man das Problem mit passenden Entwurfsmustern wie dem *Coarse-Grained-Lock* (siehe [Fowler 02]): Es wird nur das Wurzelobjekt gesperrt, und per Konvention wird sichergestellt, dass keine enthaltenen Objekte bearbeitet werden, ohne dass man das Wurzelobjekt gesperrt hat. Man spricht auch von *hierarchischen Sperren* oder *Baumsperren*.

Sperren im elektronischen Terminplaner

In unserem Terminplaner müssen wir für den Terminkalender, die Termine und die Benutzer festlegen, welche Sperrstrategie verwendet werden soll.

Abb. 6-7

*Persistente Klassen
des Terminkalenders*



Wir können für Benutzer pessimistische oder optimistische Sperren verwenden. Parallele Änderungen an den Benutzerdaten kommen so gut wie nie vor. Da Benutzer nur wenige Daten enthalten, wäre ein Datenverlust aufgrund von Konflikten beim optimistischen Sperren akzeptabel.

Auch bei den Terminen kann man prinzipiell pessimistische oder optimistische Sperrstrategien einsetzen. Verwendet man eine pessimistische Strategie, darf man allerdings nicht tief sperren. Würde man beim Bearbeiten eines Termins auch die beteiligten Benutzer sperren, wäre die parallele Arbeit im Terminkalender nur noch sehr eingeschränkt möglich.

Genauso verhält es sich mit dem Terminkalender. Soll eine pessimistische Sperrstrategie eingesetzt werden, dürfen Termine und Benut-

zer nicht automatisch mit gesperrt werden. Sonst könnte nur ein Benutzer zur gleichen Zeit Termine eintragen oder ändern.

6.7 Schnittstelle der Session

Wir haben in diesem Kapitel bereits die wesentlichen Teile der Session-Schnittstelle dargestellt. Hier beschreiben wir die Gesamtschnittstelle im Überblick. Die Methoden der Klasse `Session` lassen sich wie folgt gruppieren:

- Allgemeine Verwaltung
- Umgang mit `Connections`
- Umgang mit einzelnen Objekten
- Umgang mit Abfragen (*Queries*)
- Umgang mit dem Cache

Fast jede Methode kann eine `HibernateException` werfen. Sie ist bei der Beschreibung der Methoden hier nicht angegeben, um die Übersichtlichkeit zu erhöhen.

6.7.1 Allgemeine Verwaltung

```
EntityMode getEntityMode();
```

- Liefert den Entity-Mode der Session. Über den Entity-Mode kann festgelegt werden, dass die Geschäftsobjekte nicht als Java-Klassen modelliert werden, sondern dynamisch als Maps von Maps oder über XML. Dann benötigt man neben den Mapping-Dateien keine eigenen persistenten Klassen mehr.

```
Session getSession(EntityMode entityMode);
```

- Öffnet eine neue Session.

```
SessionFactory getSessionFactory();
```

- Liefert die `SessionFactory`, die diese Session erzeugt hat.

```
void clear();
```

- Löscht den Inhalt der Session. Alle geladenen Objekte werden in den Zustand *detached* überführt, und alle Änderungen im Cache werden verworfen.

6.7.2 Umgang mit Connections

boolean **isOpen**();

- Ist die Session offen?

boolean **isConnected**();

- Ist die Session mit einer Datenbank verbunden?

boolean **isDirty**();

- Enthält die Session Änderungen, die noch mit der Datenbank abgeglichen werden müssen? Wenn das Ergebnis true ist, würde flush Änderungen in die Datenbank schreiben.

Connection **connection**();

- Liefert die Connection der Session.

Connection **disconnect**();

- Entkoppelt die Session von der Connection.

void **reconnect**();

- Verbindet die Session mit einer neuen Connection.

void **reconnect**(Connection connection);

- Verbindet die Session mit der angegebenen Connection.

Connection **close**();

- Schließt die Session und gibt die Connection wieder frei.

6.7.3 Umgang mit einzelnen Objekten

Serializable **getIdentifizier**(Object object);

- Liefert die ID des angegebenen Objektes. Das Objekt muss in der Session vorhanden sein.

boolean **contains**(Object object);

- Enthält die Session das angegebene Objekt?

void **evict**(Object object);

- Entkoppelt das angegebene Objekt, so dass es in den Zustand *detached* überführt wird. Normalerweise muss man diese Methode nicht aufrufen. Das implizite Abkoppeln über close bzw. Objektserialisierung reicht i.d.R. aus.

```

Object load(Class theClass, Serializable id,
             LockMode lockMode);
Object load(String entityName, Serializable id,
             LockMode lockMode);
Object load(Class theClass, Serializable id);
Object load(String entityName, Serializable id);
void load(Object object, Serializable id);

```

- Lädt das Objekt mit der angegebenen ID. Kann das angeforderte Objekt nicht gefunden werden, wird eine Exception geworfen. Die `load`-Methoden sollten nur aufgerufen werden, wenn das angeforderte Objekt existiert. Sie sollten nicht verwendet werden, um festzustellen, ob ein Objekt existiert. Zu diesem Zweck sollte die `get`-Methode verwendet werden.⁴

```

void replicate(Object object,
               ReplicationMode replicationMode);
void replicate(String entityName, Object object,
               ReplicationMode replicationMode);

```

- Persistiert den Zustand des angegebenen Objektes, dass sich im Zustand *detached* befindet. Über den *Replication-Mode* wird definiert, wie mit bereits existierenden Datensätzen in der Datenbank umzugehen ist.

```

Serializable save(Object object);
void save(Object object, Serializable id);
Serializable save(String entityName, Object object);
void save(String entityName, Object object,
           Serializable id);

```

- Speichert das Objekt neu in der Datenbank (INSERT). Wird keine ID verwendet, setzt Hibernate die ID entsprechend der Konfiguration in der Mapping-Datei.

```

void saveOrUpdate(Object object);
void saveOrUpdate(String entityName, Object object);

```

- Speichert das angegebene Objekt. Je nach Objektzustand wird `save` oder `update` gerufen.

4. Ein weiterer Unterschied ist, dass `load` einen Proxy liefert und `get` immer das konkrete Objekt. Für die meisten Anwendungsfälle macht das jedoch keinen Unterschied.

```
void update(Object object);  
void update(Object object, Serializable id);  
void update(String entityName, Object object);  
void update(String entityName, Object object,  
    Serializable id);
```

- Aktualisiert das angegebene Objekt in der Datenbank (UPDATE). Das übergebene Objekt muss sich im Zustand *detached* befinden. Es muss ein Objekt mit der ID bereits in der Datenbank existieren.

```
Object merge(Object object);  
Object merge(String entityName, Object object);
```

- Persistiert den Zustand des angegebenen Objektes, das sich im Zustand *detached* befindet. Diese Methode gehört zum JPA (siehe JSR-220).

```
void persist(Object object);  
void persist(String entityName, Object object);
```

- Persistiert ein transientes Objekt. Das Objekt darf sich nicht im Zustand *detached* befinden. Diese Methode gehört zum JPA (siehe JSR-220).

```
void delete(Object object);  
void delete(String entityName, Object object);
```

- Löscht das übergebene Objekt aus der Datenbank. Das Objekt muss eine gültige ID haben. Je nach *Cascade*-Einstellung im Mapping werden enthaltene Objekte automatisch mit gelöscht.

```
void lock(Object object, LockMode lockMode);  
void lock(String entityName, Object object,  
    LockMode lockMode);
```

- Sperrt das angegebene Objekt.

```
void refresh(Object object);  
void refresh(Object object, LockMode lockMode);
```

- Aktualisiert den Objektzustand aus der Datenbank.

```
LockMode getCurrentLockMode(Object object);
```

- Liefert den Sperrmodus für das angegebene Objekt.

```
Transaction beginTransaction();
```

- Startet eine Transaktion. Mehrfacher Aufruf dieser Methode liefert unterschiedliche Transaction-Objekte, die alle auf dieselbe Datenbank-Transaktion verweisen.

Transaction **getTransaction()**;

- Liefert das Transaction-Objekt, das mit dieser Session assoziiert ist.

Object **get(Class clazz, Serializable id)**;

Object **get(Class clazz, Serializable id,
LockMode lockMode)**;

Object **get(String entityName, Serializable id)**;

Object **get(String entityName, Serializable id,
LockMode lockMode)**;

- Liefert das Objekt mit der angegebenen ID. Existiert kein persistentes Objekt mit der angegebenen ID, wird null geliefert.

String **getEntityName(Object object)**;

- Liefert den Entity-Namen für das angegebene persistente Objekt. Objekte einer Klasse können in Hibernate in unterschiedlichen Tabellen gespeichert werden. Über den Entity-Namen wird gesteuert, welche Tabelle verwendet wird. Der Entity-Name muss in den Mapping-Dateien definiert sein.

void **setReadOnly(Object entity, boolean readOnly)**;

- Definiert das angegebene Objekt als eines, das nur gelesen wird (bei `readOnly==true`), bzw. setzt den Zustand für ein Objekt wieder zurück (bei `readOnly==false`). Bei Read-only-Objekten findet kein *Dirty-Checking* statt.

6.7.4 Umgang mit Abfragen (Querys)

Hier erfolgt nur ein Überblick über Methoden an der Session. Details zur Formulierung von Abfragen finden sich in Kapitel 7.

Criteria **createCriteria(Class persistentClass)**;

Criteria **createCriteria(Class persistentClass,
String alias)**;

- Erzeugt eine Abfrage, die alle Objekte vom angegebenen Typ liefert.

Criteria **createCriteria(String entityName)**;

Criteria **createCriteria(String entityName,
String alias)**;

- Erzeugt eine Abfrage, die alle Objekte mit dem angegebenen Entity-Namen liefert.

Query **createQuery(String queryString)**;

- Erzeugt eine Abfrage mit der angegebenen HQL-Abfrage.

SQLQuery **createQuery**(String queryString);

- Erzeugt eine Abfrage mit der angegebenen SQL-Abfrage.

Query **createFilter**(Object collection,
String queryString);

- Erzeugt eine Abfrage für die angegebene persistente Collection und den angegebenen Filter (HQL). Mit dieser Funktion können Abfrageergebnisse weiter eingeschränkt werden. Die Einschränkung erfolgt nicht durch Iterieren im Hauptspeicher, sondern durch Datenbankzugriff.

Query **getNamedQuery**(String queryName);

- Liefert die benannte Abfrage mit dem angegebenen Namen.

void **cancelQuery**();

- Bricht die aktuelle Abfrage ab. Diese Methode kann aus einem anderen Thread aufgerufen werden, um eine laufende Abfrage abzubrechen. Die Methode sollte nur mit Umsicht verwendet werden.

Filter **enableFilter**(String filterName);

- Aktiviert den angegebenen Filter für diese Session.

Filter **getEnabledFilter**(String filterName);

- Liefert den aktiven Filter mit dem angegebenen Namen.

void **disableFilter**(String filterName);

- Deaktiviert den angegebenen Filter für diese Session.

6.7.5 Umgang mit dem Cache

public void **flush**() throws HibernateException;

- Schreibt die Änderungen im Cache in die Datenbank.

public void **setFlushMode**(FlushMode flushMode);

- Setzt den Flush-Modus, der definiert, wann Hibernate Änderungen am Cache in die Datenbank schreibt. Default-Wert ist auto, der vor jeder Query (außer SQL-Query) und bei Transaction.commit in die Datenbank schreibt.

public FlushMode **getFlushMode**();

- Liefert den Flush-Modus.

public void **setCacheMode**(CacheMode cacheMode);

- Setzt den Cache-Modus, der definiert, wie die Session mit dem Second-Level-Cache und dem Query-Cache interagiert.

public CacheMode **getCacheMode**();

- Liefert den Cache-Modus.

6.7.6 Hibernate-Zustandsmodell verfeinert

Mit der Session-Schnittstelle als Hintergrund können wir das Hibernate-Zustandsmodell für persistente Objekte verfeinern (siehe Abb. 6–8). Bemerkenswert ist auf jeden Fall, dass das Ausführen einer Abfrage⁵ dazu führt, dass `flush` an der zugehörigen Session aufgerufen wird. Das ist notwendig, weil man sonst schnell ein Durcheinander aus Cache und per Abfrage geladenen Objekten bekommen kann.

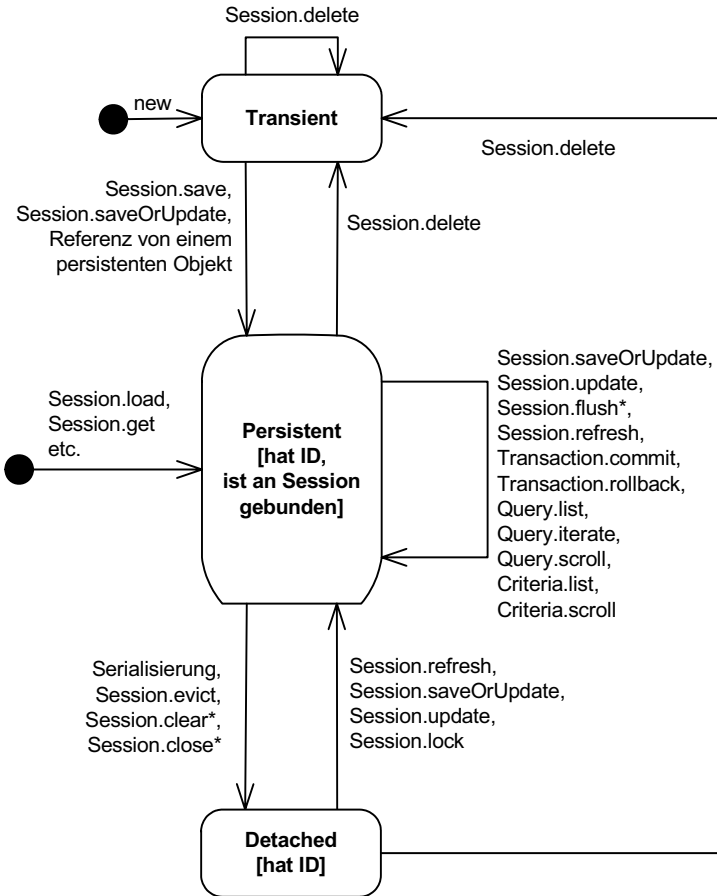


Abb. 6–8

Zustandsmodell
persistenter Objekte

* betrifft alle Objekte einer Session

- Jedenfalls wenn die Abfrage über HQL oder das Criteria-API definiert wird. Bei SQL-Abfragen führt Hibernate nicht automatisch `flush` vor dem Ausführen der Abfrage durch.

6.7.7 Das Statistics-API

Hibernate bietet die Möglichkeit, eine Reihe von internen Informationen mitzuprotokollieren, die man dann mit `SessionFactory.getStatistics` abrufen kann.

Dieses Mitprotokollieren verbraucht Ressourcen, weshalb es per Default ausgeschaltet ist. Um es zu aktivieren, muss man die Property `hibernate.generate_statistics` in der Konfiguration (siehe Kapitel 5) auf `true` setzen. Man sollte nicht vergessen, für Performancetests und erst recht für den produktiven Einsatz diese Protokollierung abzuschalten.

`getStatistics` liefert ein Exemplar des `Statistics`-Interface zurück, das eine Reihe von allgemeinen Informationen bereitstellt. Zunächst einmal gibt es Methoden, um abzufragen, wie oft die verschiedenen Arten von Operationen (`insert`, `update` etc.) ausgeführt wurden. Außerdem stellt das Interface Statistikdaten zu Transaktionen und Caches bereit. Zusätzlich gibt es die Methoden `getEntityStatistics`, `getCollectionStatistics` und `getQueryStatistics`, mit denen man detailliertere Daten zu einer einzelnen Geschäftsobjektklasse, einer bestimmten `Collection`-Rolle oder einer konkreten `Query` abrufen kann.

6.8 Application Server

Hibernate kann sowohl mit wie auch ohne Application Server eingesetzt werden. Wird Hibernate mit einem Application Server eingesetzt, kann es mit den Transaktionsmechanismen des Application Servers über JTA (*Java Transaction Architecture*, siehe auch Kapitel 5: *Konfiguration*) zusammenarbeiten. Hierfür gibt es zwei Möglichkeiten:

- Das `Transaction`-Objekt wird über *JNDI* vom Application Server bezogen. Dazu werden eine `JTATransactionFactory` und ein `JNDI`-Name in der Konfiguration angegeben. Diese Verwendung ist sinnvoll, wenn Hibernate innerhalb einer `Session Bean` eines `EJB-Containers` genutzt wird und das `Session Bean` für *Bean-Managed Transaction (BMT)* konfiguriert ist. In diesem Fall ist der Code der `Session Bean` zuständig für das Starten und Abschließen (`commit`) bzw. Abbrechen (`rollback`) der Transaktion.
- Läuft Hibernate in einer `Session Bean`, das für *Container-Managed Transaction (CMT)* konfiguriert ist, wird die Transaktion vom `EJB-Container` verwaltet. Hibernate stellt dazu die Klasse `CMTTransactionFactory` zur Verfügung. Im Unterschied zur *Bean-Managed*

Transaction ist es in dieser Umgebung nicht nötig, eine Transaktion explizit zu bestätigen oder abzurechnen. Das erledigt der Application Server. Im Deployment-Deskriptor für das Session Bean ist konfiguriert, wie sich der Application Server bei CMT genau verhalten soll.

Läuft Hibernate im Application Server, ändert sich die Configuration, die Erzeugung der Session und evtl. die Handhabung der Transaktionen ein wenig. Der eigentliche Anwendungscode bleibt aber unverändert. Wenn eine Session Bean Methoden anderer Session Beans aufruft, kann es zu einem Thread-Wechsel kommen. In diesem Fall ist es nötig, die Session an die aufgerufenen Methoden weiterzureichen.

Um diese komplizierte Handhabung der Session zu vermeiden, bietet Hibernate die nützliche Methode `getCurrentSession` an der `SessionFactory` an. Diese Methode liefert innerhalb einer Session-Bean-Methode immer dieselbe Session. Die Session über die Methode `getCurrentSession` zu beziehen, ist in einem Application Server immer empfehlenswert – unabhängig davon, ob BMT oder CMT verwendet wird.

Hilfsmethode

```
getCurrentSession()
```

Es ist zu beachten, dass die Methode `getCurrentSession` nur dann eine Session liefern kann, wenn JTA für Transaktionen verwendet wird. Kapitel 10 beschreibt genauer, wie Hibernate im Application Server eingesetzt werden kann.

6.9 Caches

Um einen häufigen lesenden Zugriff auf Daten zu beschleunigen, benutzt Hibernate Caches. Wenn eine Session ein Objekt in der Datenbank speichert oder aus der Datenbank lädt, hält es sich diese Objektversion im Cache. Bei einer anschließenden Anfrage auf dieses Objekt ist kein weiterer Datenbankzugriff nötig. Die Session kann das Objekt direkt aus dem Cache zurückliefern. Des Weiteren unterstützt Hibernate das Puffern von Querys.

Caches sind besonders bei sehr vielen lesenden Zugriffen sinnvoll, um die Performance zu erhöhen. Das gilt vor allem, wenn die Daten selten oder gar nicht verändert werden.

Ein Cache kann sich auf eine Session, einen Prozess bzw. eine Virtual Machine oder einen Cluster beziehen. Bei Hibernate wird der Cache innerhalb einer Transaktion als *First-Level Cache* bezeichnet. Arbeitet der Cache auf Ebene der VM oder eines Clusters, redet man von einem *Second-Level Cache*.

<i>First-Level Cache für Transaktionen</i>	Der <i>First-Level Cache</i> (Cache auf Ebene einer Session) speichert alle Objekte, die in einer Session geladen oder gespeichert wurden. Mit dem First-Level Cache von Hibernate haben wir in diesem Kapitel bereits Bekanntschaft gemacht.
<i>Second-Level Cache für Prozess/VM</i>	Ein <i>Second-Level Cache</i> auf Ebene Prozess/Virtual Machine speichert alle Objekte, die in einer Virtual Machine geladen oder gespeichert wurden. Sehen sich z.B. in einem Webshop viele Anwender immer wieder dieselben Artikel an, müssen die Artikel nicht für jeden Anwender neu geladen werden. Nachdem sich der erste Anwender einen Artikel angesehen hat, steht dieser über den Second-Level Cache auch den anderen Anwendern zur Verfügung. Das Thema <i>Second-Level Cache</i> wird im Detail in Kapitel 11 beschrieben.

6.10 Hibernate-Exceptions

Sehr viele Methoden der Hibernate-Klassen können Exceptions werfen. Das ist nur logisch: Hibernate verwendet JDBC, und die meisten JDBC-Methoden werfen SQL-Exceptions. Auch das wiederum ist logisch, weil JDBC auf Datenbanksystemen aufsetzt und z.B. durch Netzwerkprobleme vom einen Moment auf den nächsten unerreichbar sein können.

Während alle SQL-Exceptions Checked-Exceptions sind (also gefangen werden müssen), sind alle Hibernate-Exceptions Runtime-Exceptions. Man *kann* die Hibernate-Exceptions also fangen, *muss* es aber nicht. Diese Strategie ist im Großen und Ganzen auch sinnvoll. Bei den meisten Datenbankproblemen kann die Anwendung sowieso nicht sinnvoll reagieren. Außer einem Hinweis an den Anwender, dass die Datenbank nicht erreichbar ist, und der Empfehlung, es noch einmal zu versuchen, kann die Anwendung kaum noch etwas Sinnvolles tun.

Also fängt man die Hibernate-Exceptions meistens gar nicht oder nur generisch auf oberster Ebene. Und gerade bei dieser Strategie bedeuten Runtime-Exceptions deutlich weniger Tipparbeit. Die folgende Tabelle zeigt die Hibernate-Exception-Hierarchie.

Exception	Beschreibung
HibernateException	Exception, die von Hibernate ausgelöst wird.
BatchException	Problem beim Batch-Update.
CacheException	Problem mit dem oder im Cache.
NoCachingEnabledException	Der Second-Level Cache wurde gefordert, ist aber nicht verfügbar.
CallbackException	Diese Exception sollte bei Problemen in Lifecycle- oder Interceptor-Callback geworfen werden.
IdentifizierGenerationException	Wird von IdentifizierGenerator-Klassen geworfen, wenn die Generierung der ID nicht möglich war.
InstantiationException	Hibernate konnte ein Objekt nicht instanzieren. Hinweis: nicht zu verwechseln mit der gleichnamigen Exception aus dem JDK.
JDBCException	JDBC-Problem aufgetreten. Kapselt SQLException.
ConstraintViolationException	Eine Datenbankaktion hat einen Constraint verletzt.
DataException	Der SQL-Befehl war syntaktisch korrekt, passte aber nicht zum Datenbankschema (z.B. wg. eines Typfehlers).
GenericJDBCException	Generische unspezifische JDBC-Exception.
JDBCConnectionException	Es sind Probleme bei der Kommunikation mit der Datenbank aufgetreten. Mögliche Ursachen: falsche Zugriffsdaten in der Konfiguration oder Verbindungsabbruch.
LockAcquisitionException	Hibernate hat versucht, eine Sperre auf der Datenbank zu setzen, und dieser Versuch schlug fehl.
SQLGrammarException	Die an die Datenbank gesendeten SQL-Befehle waren fehlerhaft.
LazyInitializationException	Es wurde versucht, Objekte nachzuladen, obwohl das Vaterobjekt nicht mehr mit einer Session verbunden ist (detached).
MappingException	Das OR-Mapping wurde falsch konfiguriert. Diese Exception tritt meistens zur Konfigurationszeit der Anwendung auf.
DuplicateMappingException	In der Mapping-Beschreibung wurden Elemente doppelt verwendet (z.B. dieselbe Tabelle für verschiedene Geschäftsobjekte). →

Exception	Beschreibung
InvalidMappingException	Das Mapping ist fehlerhaft. Diese Exception ist sehr ähnlich zur MappingException, enthält aber mehr Informationen über das Problem.
MappingNotFoundException	Zu einer Mapping-Datei konnte die zugehörige persistente Klasse nicht gefunden werden.
PropertyNotFoundException	Ein von Hibernate erwarteter Getter oder Setter fehlte in einem Geschäftsobjekt.
NonUniqueObjectException	Es wurde versucht, zwei Objekte derselben Klasse mit derselben ID einer Session hinzuzufügen.
NonUniqueResultException	Es wurde Query.uniqueResult gerufen, obwohl die Abfrage mehr als ein Ergebnis lieferte. Dies ist die einzige Exception, nach der die Session und Abfrage weiterbenutzt werden können.
PersistentObjectException	Es wurde ein persistentes Objekt an eine Methode übergeben, die ein transientes Objekt erwartete.
PropertyAccessException	Es trat ein Problem beim Zugriff auf ein Attribut eines Geschäftsobjektes auf. Mögliche Ursachen: Setter oder Getter ist nicht public; Setter oder Getter hat eine Exception geworfen; ein null-Wert in der Datenbank wird auf einen Java-Basisdatentyp gemappt; Spaltentyp in der Datenbank und Attributtyp lassen sich nicht ineinander überführen.
PropertyValueException	Ein Attributwert kann nicht persistiert werden. Mögliche Ursachen: Das Attribut ist null, obwohl es als not-null gemappt wurde; ein Attribut verweist auf ein ungespeichertes transientes Objekt.
QueryException	Es ist ein Problem bei der Transformation einer Hibernate-Query (HQL) in SQL aufgetreten. Häufigste Ursache: Syntaxfehler in der HQL-Abfrage.
QueryExecutionRequestException	Es wurde versucht, eine illegale Abfrage auszuführen.
QueryParameterException	Parameter ungültig oder in der Query nicht vorhanden.
QuerySyntaxException	Syntaxfehler in einer HQL-Abfrage.
SerializationException	Ein Attribut eines Geschäftsobjektes ließ sich nicht serialisieren oder deserialisieren. →

Exception	Beschreibung
SessionException	Es wurde eine Session-Methode gerufen, obwohl die Session in einem für den Aufruf ungültigen Zustand war (z.B. geschlossen).
StaleStateException	Ein Objekt hat einen veralteten Zustand. Mögliche Ursachen: Es wurde versucht, mit einer optimistischen Sperrstrategie ein Objekt zu speichern, das bereits von einem anderen Benutzer gespeichert wurde; es wurde versucht, einen nicht existierenden Datensatz zu löschen oder zu aktualisieren.
StaleObjectStateException	Wie StaleStateException, aber mit zusätzlichen Informationen über das problematische Objekt.
TooManyRowsAffectedException	Durch eine Operation auf der Datenbank wurden mehr Zeilen verändert als erwartet. Deutet häufig darin hin, dass mehrere Tabellenzeilen denselben Primärschlüssel haben.
BatchedTooManyRowsAffectedException	Wie TooManyRowsAffectedException nur für Batch-Update. Enthält zusätzlich die fehlerhafte Position im Batch.
TransactionException	Eine Transaktion konnte nicht begonnen oder mit commit bzw. rollback beendet werden.
TransientObjectException	Es wurde ein transientes Objekt an eine Methode übergeben, die ein persistentes Objekt erwartet.
TypeMismatchException	Typen passen nicht zusammen.
UnresolvableObjectException	Hibernate konnte zu einer ID kein zugehöriges Objekt laden. Die Exception tritt vor allem auf, wenn beim Nachladen eines Attributs das referenzierte Objekt nicht existiert.
ObjectDeletedException	Es wurde etwas Unerlaubtes mit einem gelöschten Objekt gemacht.
ObjectNotFoundException	Session.load konnte das angeforderte Objekt nicht finden.
ValidationFailure	Wird von Validatable.validate geworfen, wenn eine Invariante verletzt wurde.
WrongClassException	Session.load hat auf einen Datensatz zugegriffen, der als Diskriminator eine Klasse benennt, die nicht zur angegebenen Klasse passt.

Die meisten der Hibernate-Exceptions deuten auf Programmier- oder Konfigurationsfehler hin. Es ist also nicht sinnvoll, diese Exceptions früh zu fangen, um die Anwendung wieder aufsetzen zu lassen (*Recovery*). Angemessen ist meist die harte generische Variante: Abbruch der Datenbankaktion durch Wegwerfen der Session und allgemeine Meldung an den Benutzer.

Nur bei wenigen Exceptions kann ein frühes Fangen und automatische Exception-Behandlung sinnvoll sein:

- `LockAcquisitionException`: Man kann die Exception fangen und dem Anwender die Rückmeldung geben, dass ein gewünschtes Objekt gerade von einem anderen Anwender in Bearbeitung ist.
- `QuerySyntaxError`: Das Fangen dieser Exception kann sinnvoll sein, wenn der Anwender durch Freitexteingaben die Form einer HQL-Abfrage beeinflussen kann. Der Anwender müsste dann die Rückmeldung erhalten, dass seine Eingaben fehlerhaft waren. Nach Möglichkeit sollte die Überprüfung der Benutzereingaben aber oberflächennah und ohne Rückgriff auf Hibernate erfolgen.
- `StaleStateException`, `StaleObjectStateException`: Bei Versionskonflikten aufgrund der optimistischen Sperrstrategie kann es z.B. sinnvoll sein, die Session wegzuworfen und die im Hauptspeicher befindlichen Objekte aus der Datenbank zu aktualisieren (*refresh*).
- `UnresolvableObjectException`: Es kann in einigen Anwendungen durchaus in Ordnung sein, wenn referenzierte Objekte nicht mehr existieren. Die Anwendung kann das dann entsprechend visualisieren und das Attribut auf `null` setzen.

JDBCException

Ein wiederkehrendes Problem bei der direkten Arbeit mit JDBC ist die Interpretation der Exception-Ursache. Dummerweise können sowohl Programmierfehler (z.B. fehlerhaftes SQL) wie auch unerwartete Datenzustände `SQLExceptions` erzeugen. Die Anwendung muss jeweils unterschiedlich reagieren. Während sie sich im Falle eines Programmierfehlers lieber nicht um die Exception kümmern sollte, muss sie im zweiten Fall eine sinnvolle Behandlung (mind. aussagekräftige Meldung an den Anwender oder Systemadministrator) bereitstellen.

Die genaue Ursache für die `SQLException` über JDBC zu ermitteln, ist erstens oft umständlich und zweitens auf jeden Fall abhängig von der konkret verwendeten Datenbank. Meistens muss man datenbank-spezifische Fehlercodes auswerten. Hibernate erleichtert uns die Arbeit hier ganz wesentlich, indem es uns die Interpretation der Exception-

Ursache abnimmt. Hibernate interpretiert die `SQLException` und erzeugt ggf. eine Exception der Subklassen von `SQLException`.

Die Interpretation der `SQLExceptions` übernimmt ein `SQLExceptionConverter`, der nur eine Methode definiert:

```
public interface SQLExceptionConverter {
    public JDBCException convert(
        SQLException sqlException,
        String message,
        String sql);
}
```

Auch wenn man Hibernate nicht für sein OR-Mapping benutzen kann oder will, kann man dennoch von seinem `SQLExceptionConverter` profitieren. Man kann sich einfach ein `Dialect`-Objekt für seine Datenbank erzeugen und sich mit `buildSQLExceptionConverter` einen passenden `SQLExceptionConverter` erzeugen lassen. Und diesen Converter kann man dann für die Interpretation seiner `SQLExceptions` verwenden.

*Verwendung von
JDBCException ohne
Hibernate*

```
Dialect db2 = new DB2Dialect();
SQLExceptionConverter conv =
    db2.buildSQLExceptionConverter();
JDBCException e = conv.convert(sqlExc, msg, sql);
```

Wir haben schon gesehen, dass sich die `Session` in einem undefinierten Zustand befindet, wenn eine Methode der `Session`, des `Caches` oder einer `Transaktion` eine `Exception` wirft. In einigen Fällen kann die `Session` nach einer `Exception` durch Aufruf von `clear` wieder initialisiert werden. Das ist insbesondere dann der Fall, wenn sich eine `Transaktion` nicht mit `commit` abschließen lässt, z.B. weil bei optimistischem Sperren ein anderer Anwender sein Objekt vorher gespeichert hat. Allerdings garantiert Hibernate nicht, dass die `Session` nach einer `Exception` mit `clear` wieder initialisiert werden kann. Der einzige sichere Weg besteht im Wegwerfen der `Session` und der Erzeugung einer neuen `Session`.

Exceptions in Sessions

6.11 Referenzen

[Fowler 02] Martin Fowler: *Patterns of Enterprise Application Architecture*, Addison-Wesley, 2002

Fowler beschreibt Entwurfsmuster für Unternehmensanwendungen. Ein relevanter Teil der Muster beschäftigt sich mit Datenbankanbindung und Transaktionsbehandlung.

[Gamma et al. 05] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Entwurfsmuster – Elemente wiederverwendbarer objektorientierter Software*, Addison-Wesley, 1995

Das Standardwerk zu Entwurfsmustern enthält auch das Singleton-Entwurfsmuster.

[Hibernate] <http://www.hibernate.org/5.html>

Die Hibernate-Dokumentation zur Behandlung von Sessions, Transaktionen und Caches