HANSER

Christian Bauer, Gavin King

Java Persistence mit Hibernate

ISBN-10: 3-446-40941-6
ISBN-13: 978-3-446-40941-5

Leseprobe

Weitere Informationen oder Bestellungen unter
http://www.hanser.de/978-3-446-40941-5
sowie im Buchhandel

4

4 Mapping von Persistenzklassen

Die Themen dieses Kapitels:

- Das Konzept der Entity- und Wert-Typen
- Mapping von Klassen mit XML und Annotationen
- Feingranulierte Eigenschafts- und Komponenten-Mappings

Dieses Kapitel macht Sie mit grundlegenden Mapping-Optionen bekannt und erklärt, wie Klassen und Eigenschaften auf Tabellen und Spalten gemappt werden. Wir erläutern, wie Sie mit der Datenbank-Integrität und Primärschlüsseln umgehen und wie verschiedene Einstellungen für Metadaten benutzt werden können, um Hibernate für das Laden und Speichern von Objekten anzupassen. Bei allen Mapping-Beispielen arbeiten Sie sowohl mit dem nativen XML-Format von Hibernate als auch mit JPA-Annotationen sowie XML-Deskriptoren. Wir schauen uns das Mapping von feingranulierten Domain-Modellen genau an und wie Eigenschaften und eingebettete Komponenten gemappt werden.

Zuerst werden wir allerdings erst einmal den wesentlichen Unterschied zwischen Entities- und Wert-Typen definieren und erklären, wie Sie an das objekt-relationale Mapping Ihres Domain-Modells herangehen sollten.

4.1 Entities- und Wert-Typen

Entities sind persistente Typen, die Business-Objekte erster Ordnung repräsentieren (der Begriff Objekt wird hier in seinem natürlichen Sinn gebraucht). Anders gesagt sind manche der Klassen und Typen, mit denen Sie es bei einer Applikation zu tun bekommen, wichtiger und andere dementsprechend weniger wichtig. Sie werden wahrscheinlich zustimmen, dass `Item` in CaveatEmptor eine bedeutendere Klasse ist als `String`. `User` ist wahrscheinlich wichtiger als `Address`. Wodurch wird etwas wichtig? Schauen wir uns dieses Thema aus einer anderen Perspektive an.

4.1.1 Feingranulierte Domain-Modelle

Ein Hauptziel von Hibernate ist die Unterstützung feingranulierter Domain-Modelle, die wir als die wichtigste Voraussetzung für ein Rich Domain Model isoliert haben. Das ist einer der Gründe, warum wir mit POJOs arbeiten. Ganz grob gesagt bedeutet *feingranuliert* dass es mehr Klassen als Tabellen gibt.

Ein Anwender könnte zum Beispiel sowohl eine Rechnungs- als auch eine Post/Lieferadresse haben. In der Datenbank haben Sie dann eine Tabelle USERS mit den Spalten `BILLING_STREET`, `BILLING_CITY` und `BILLING_ZIPCODE` und dann noch `HOME_STREET`, `HOME_CITY` und `HOME_ZIPCODE`. (Erinnern Sie sich noch an das Problem der SQL-Typen, das wir in Kapitel 1 angesprochen haben?)

Im Domain-Modell können Sie genauso vorgehen, wobei die beiden Adressen als sechs Eigenschaften mit String-Werten der Klasse `User` repräsentiert werden. Doch es ist besser, das mit einer `Address`-Klasse zu modellieren, bei der der `User` die Eigenschaften `billingAddress` und `homeAddress` hat, und somit werden für eine Tabelle drei Klassen benutzt.

Durch ein solches Domain-Modell wird ein verbesserter Zusammenhalt und eine umfassendere Wiederverwendung von Code erreicht, und es ist verständlicher als SQL-Systeme mit unflexiblen Typensystemen. In der Vergangenheit haben viele ORM-Lösungen keinen sonderlich guten Support für diese Art von Mapping geboten.

Hibernate unterstreicht die Nützlichkeit von feingranulierten Klassen zur Implementierung von Typensicherheit und Verhalten. Vielerorts wird beispielsweise eine E-Mail-Adresse als eine Eigenschaft von `User` mit einem String-Wert gebildet. Ein durchdachterer Ansatz ist, eine `EmailAddress`-Klasse zu definieren, welche Semantiken und Verhalten auf höherer Ebene hinzufügt – wie zum Beispiel eine `sendEmail()`-Methode.

Dieses Problem der Granularität führt uns zu einer Unterscheidung von zentraler Bedeutung im ORM. In Java sind alle Klassen gleichermaßen angesehen: Alle Objekte haben ihre eigene Identität und einen eigenen Lebenszyklus.

Gehen wir mal gemeinsam ein Beispiel durch.

4.1.2 Konzeptdefinition

Zwei Personen leben in der gleichen Wohnung und beide lassen sich bei CaveatEmptor mit einem Konto registrieren. Natürlich wird jedes Konto von einer Instanz namens `User` repräsentiert, also haben Sie zwei Entity-Instanzen. Beim CaveatEmptor-Modell hat die Klasse `User` eine `homeAddress`-Assoziation mit der Klasse `Address`. Haben beide `User`-Instanzen eine Laufzeit-Referenz auf die gleiche `Address`-Instanz oder hat jede `User`-Instanz eine Referenz auf seine eigene `Address`? Wenn `Address` gemeinsam genutzte Laufzeit-Referenzen unterstützen soll, ist es ein Entity-Typ. Falls nicht, ist es wahrscheinlich ein Wert-Typ und hängt von daher von einer Referenz einer besitzenden Entity-Instanz ab, was ebenfalls eine Identität bietet.

Wir befürworten ein Design mit mehr Klassen als Tabellen: Eine Zeile repräsentiert mehrere Instanzen. Weil Datenbankidentität durch den Wert des Primärschlüssels implemen-

tiert wird, werden einige persistente Objekte keine eigene Identität haben. Als Folge davon implementiert der Persistenzmechanismus *pass-by-value*-Semantiken für einige Klassen! Eines der in der Zeile repräsentierten Objekte hat eine eigene Identität, und andere hängen davon ab. Im vorigen Beispiel hängen die Spalten in der `USERS`-Tabelle, in der die Adressinformationen enthalten sind, vom Identifikator des Anwenders ab, dem Primärschlüssel der Tabelle. Eine Instanz von `Address` hängt von einer Instanz von `User` ab.

Hibernate nimmt die folgende wesentliche Unterscheidung vor:

- Ein Objekt des Typs Entity hat seine eigene Datenbankidentität (Primärschlüsselwert). Eine Objektreferenz auf eine Entity-Instanz wird als Referenz in der Datenbank persistiert (ein Fremdschlüsselwert). Eine Entity hat ihren eigenen Lebenszyklus, sie kann unabhängig von anderen Entities existieren. In CaveatEmptor sind Beispiele dafür `User`, `Item` und `Category`.
- Ein Objekt mit dem Typ Wert hat keine Datenbankidentität, es gehört zu einer Entity-Instanz und sein Persistenzstatus ist in der Tabellenzeile der besitzenden Entity eingebettet. Wert-Typen haben keine Identifikatoren oder Identifikatoren-Eigenschaften. Die Lebensspanne einer Instanz des Typs Wert ist an die Lebensspanne der besitzenden Entity-Instanz gebunden. Ein Wert-Typ unterstützt keine gemeinsam genutzten Referenzen: Wenn zwei Anwender in der gleichen Wohnung leben, haben sie beide eine Referenz auf ihre eigene `homeAddress`-Instanz. Die offensichtlichsten Wert-Typen sind Klassen wie `Strings` und `Integers`, doch alle JDK-Klassen werden als Wert-Typen betrachtet. Benutzerdefinierte Klassen können auch als Wert-Typen gemappt werden; CaveatEmptor hat beispielsweise `Address` und `MonetaryAmount`.

Die Identifizierung von Entities und Wert-Typen in Ihrem Domain-Modell ist keine ad hoc-Aufgabe, sondern folgt einer bestimmten Prozedur.

4.1.3 Identifizierung von Entities und Wert-Typen

Vielleicht finden Sie es hilfreich für Sie, Ihren UML-Klassendiagrammen stereotype Informationen hinzuzufügen, damit Sie Entities und Wert-Typen auf einen Blick voneinander unterscheiden können. Diese Praxis zwingt Sie auch dazu, über diese Unterscheidung für all Ihre Klassen nachzudenken, was ein erster Schritt in Richtung eines optimalen Mappings und einer gut funktionierenden Persistenzschicht ist. Schauen Sie sich die Abbildung 4.1 als Beispiel an.

Die Klassen `Item` und `User` sind offensichtlich Entities. Sie haben alle ihre eigene Identität, ihre Instanzen haben Verweise von vielen anderen Instanzen (gemeinsame Verweise) und sie besitzen unabhängige Lebenszyklen.

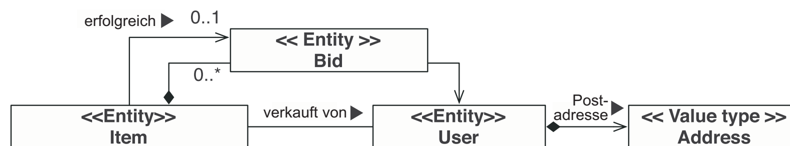

Abbildung 4.1 Stereotype für Entities und Wert-Typen werden in das Diagramm eingefügt.

Es ist ebenfalls einfach, die `Address` als Wert-Typ zu identifizieren: Auf eine bestimmte `Address`-Instanz wird nur von einer einzigen `User`-Instanz verwiesen. Das wissen Sie, weil die Assoziation als Komposition erstellt wurde, bei der die `User`-Instanz vollkommen für den Lebenszyklus der referenzierten `Address`-Instanz verantwortlich ist. Von daher kann auf `Address`-Objekte nicht von anderer Stelle aus verwiesen werden und sie brauchen keine eigene Identität.

Die `Bid`-Klasse ist ein Problem. Beim objektorientierten Modeling verwenden Sie eine Komposition (die Assoziation zwischen `Item` und `Bid` mit der ausgefüllten Raute), und ein Item verwaltet den Lebenszyklus aller `Bid`-Objekte, auf die es einen Verweis hat (es ist eine Sammlung von Verweisen). Das scheint vernünftig zu sein, weil die Gebote (*bids*) nutzlos wären, wenn es kein `Item` mehr gäbe. Doch gleichzeitig gibt es eine andere Assoziation zu `Bid`: Ein `Item` kann einen Verweis auf dessen `successfulBid` enthalten. Das erfolgreiche Gebot muss auch eines der Gebote sein, auf die von der Collection verwiesen wird, doch dies wird im UML-Diagramm nicht ausgedrückt. Auf jeden Fall müssen Sie mit möglichen gemeinsamen Verweisen auf `Bid`-Instanzen umgehen, von daher muss die `Bid`-Klasse eine Entity sein. Sie hat einen abhängigen Lebenszyklus, muss aber eine eigene Identität haben, um gemeinsame Verweise unterstützen zu können.

Sie finden diese Art von gemischtem Verhalten des Öfteren; jedoch sollte Ihre erste Reaktion sein, aus allem eine Klasse mit Wert-Typen zu machen und sie nur dann in eine Entity zu verwandeln, wenn es unbedingt nötig ist. Versuchen Sie, Ihre Assoziationen einfach zu halten: Collections tragen manchmal zur Komplexität bei, ohne irgendwelche Vorteile zu bringen. Anstatt eine persistente Collection von `Bid`-Verweisen zu mappen, können Sie eine Query schreiben, um alle Gebote für ein `Item` zu bekommen (auf diesen Punkt kommen wir in Kapitel 7 noch einmal zurück).

Als nächsten Schritt nehmen Sie das Diagramm Ihres Domain-Modells und implementieren POJOs für alle Entity- und Wert-Typen. Sie müssen sich um drei Sachen kümmern:

- *Gemeinsame Verweise:* Schreiben Sie Ihre POJO-Klassen so, dass gemeinsame Verweise auf Instanzen von Wert-Typen vermieden werden. Achten Sie beispielsweise darauf, dass nur von einem `User` auf ein `Address`-Objekt verwiesen werden kann. Sie können es zum Beispiel unveränderlich machen und die Beziehung mit dem `Address`-Konstruktor erzwingen.
- *Abhängigkeiten im Lebenszyklus:* Wie schon besprochen ist der Lebenszyklus einer Instanz mit Wert-Typ an die besitzende Entity-Instanz gebunden. Wenn ein `User`-Objekt gelöscht wird, müssen auch die von `Address` abhängigen Objekt(e) gelöscht werden. Für so etwas gibt es in Java kein Konzept oder Schlüsselwort, doch der Workflow Ihrer Applikation und die Benutzerschnittstelle müssen so designt werden, dass sie Abhängigkeiten im Lebenszyklus respektieren und erwarten. Persistenz-Metadaten enthalten die kaskadierenden Regeln für alle Abhängigkeiten.
- *Identität:* Entity-Klassen brauchen in fast allen Fällen eine Identifikator-Eigenschaft. Benutzerdefinierte Klassen mit Wert-Typ (und JDK-Klassen) haben keine Identifikator-Eigenschaft, weil Instanzen über die besitzende Identität identifiziert werden.

Wir kommen auf Klassen-Assoziationen und Regeln für Lebenszyklen zurück, wenn wir später in diesem Buch fortschrittlichere Mappings besprechen. Allerdings ist die Objektidentität ein Thema, das Sie sich an dieser Stelle zu eigen machen müssen.

4.2 Entities mit Identität mappen

Es ist besonders wichtig, den Unterschied zwischen Objektidentität und Objektgleichheit zu verstehen, bevor wir Begriffe wie Datenbankidentität besprechen und die Art, wie Hibernate Identität verwaltet. Als Nächstes untersuchen wir, wie Objektidentität und -gleichheit zur Datenbankidentität (Primärschlüssel) in Beziehung stehen.

4.2.1 Identität und Gleichheit bei Java

Java-Entwickler verstehen den Unterschied zwischen Objektidentität und Objektgleichheit bei Java. Die Objektidentität (`==`) ist ein Konzept, das anhand von Javas virtueller Maschine definiert wird. Zwei Objektverweise sind identisch, wenn sie auf den gleichen Speicherort zeigen.

Objektgleichheit dagegen ist ein Konzept, das von Klassen definiert wird, die die Methode `equals()` implementieren, die manchmal auch als Äquivalenz bezeichnet wird. Äquivalenz bedeutet, dass zwei verschiedene (nichtidentische) Objekte den gleichen Wert haben. Zwei verschiedene Instanzen von `String` sind gleich, wenn sie die gleiche Folge von Zeichen repräsentieren, auch wenn beide ihren eigenen Speicherort in der virtuellen Maschine haben. (Wenn Sie ein Java-Guru sind, stimmen wir zu, dass `String` ein Sonderfall ist. Gehen Sie davon aus, dass wir eine andere Klasse verwendet haben, um diesen Punkt zu verdeutlichen.)

Dieses Bild wird durch die Persistenz verkompliziert. Mit objekt-relationaler Persistenz ist ein persistentes Objekt eine Repräsentation einer bestimmten Zeile einer Datenbanktabelle im Speicher. Gemeinsam mit der Java-Identität (Standort im Speicher) und Objektgleichheit gibt es nun auch die Datenbankidentität (das ist der Standort im persistenten Datenspeicher). Sie haben nun drei Methoden für die Identifizierung von Objekten:

- Objekte sind identisch, wenn sie den gleichen Speicherort in der JVM belegen. Das kann über den Operator `==` geprüft werden. Dieses Konzept nennt man Objektidentität.
- Objekte sind gleich, wenn sie den gleichen Wert haben, wie es durch die Methode `equals(Object o)` definiert wird. Klassen, die diese Methode nicht explizit überschreiben, erben die Implementierung, die von `java.lang.Object` definiert wird, die die Objektidentität überprüft. Dieses Konzept nennt man Gleichheit.
- Objekte, die in einer relationalen Datenbank gespeichert werden, sind identisch, wenn sie die gleiche Zeile repräsentieren oder äquivalent die gleiche Tabelle und den gleichen Primärschlüsselwert aufweisen. Dieses Konzept nennt man Datenbankidentität.

Wir sollten uns nun anschauen, wie die Datenbankidentität in Hibernate mit der Objektidentität in Beziehung steht und wie die Datenbankidentität über die Mapping-Metadaten ausgedrückt wird.

4.2.2 Umgang mit Datenbankidentität

Hibernate gibt der Applikation eine Datenbankidentität auf zweierlei Weise an:

- Über den Wert der Identifikator-Eigenschaft einer Persistenzinstanz
- Über den Wert, der von `Session.getIdentifier(Object entity)` zurückgegeben wird

Einfügen einer Identifikator-Eigenschaft bei Entities

Die Identifikator-Eigenschaft ist ein Sonderfall – ihr Wert ist der Wert des Primärschlüssels der Datenbankzeile, die von der Persistenzinstanz repräsentiert wird. Wir zeigen in den Diagrammen des Domain-Modells die Identifikator-Eigenschaft normalerweise nicht. In den Beispielen wird diese Eigenschaft stets als `id` bezeichnet. Wenn `myCategory` eine Instanz von `Category` ist, gibt der Aufruf von `myCategory.getId()` den Wert des Primärschlüssels der Zeile zurück, die von `myCategory` in der Datenbank repräsentiert wird.

Wir wollen nun eine Identifikator-Eigenschaft für die Klasse `Category` implementieren:

```
public class Category {
    private Long id;
    ...
    public Long getId() {
        return this.id;
    }

    private void setId(Long id) {
        this.id = id;
    }
    ...
}
```

Sollten Zugriffsmethoden für die Identifikator-Eigenschaft privat oder öffentlich gemacht werden? Nun, Datenbank-Identifikatoren werden von der Applikation oft als praktisches Handle für eine bestimmte Instanz benutzt, sogar außerhalb der Persistenzschicht. Es ist beispielsweise für Webapplikationen üblich, dem Anwender die Resultate eines Suchlaufs als Liste mit einer Zusammenfassung der Informationen anzuzeigen. Wenn der Anwender ein bestimmtes Element auswählt, muss sich die Applikation dieses gewählte Objekt holen, und es ist üblich, für diesen Zweck den Identifikator nachzuschlagen – Sie haben Identifikatoren wahrscheinlich schon auf diese Weise benutzt, auch in Applikationen, die mit JDBC arbeiten. Es ist normalerweise auch angemessen, die Datenbankidentität über eine öffentliche Zugriffsmethode bereitzustellen. Andererseits deklarieren Sie normalerweise die Methode `setId()` als privat und lassen Hibernate den Identifikator-Wert generieren und setzen. Oder Sie mappen ihn mit direktem Feld-Zugriff und implementieren nur eine Getter-Methode. (Die Ausnahme dieser Regel sind Klassen mit natürlichen Schlüsseln, wo der Wert des Identifikators von der Applikation zugewiesen wird, bevor das Objekt persistent gemacht wird, anstatt von Hibernate generiert zu werden. Diese natürlichen Schlüssel besprechen wir in Kapitel 8.) Hibernate lässt Sie den Identifikator-Wert einer Persistenzinstanz nicht verändern, nachdem er einmal zugewiesen wurde. Ein Primärschlüsselwert ändert sich nie – anderenfalls wäre das Attribut kein würdiger Kandidat für einen Primärschlüssel!

Der Java-Typ der Identifikator-Eigenschaft, `java.lang.Long` im vorigen Beispiel, hängt vom Typ des Primärschlüssels der `CATEGORY`-Tabelle ab und wie er in Hibernate-Metadaten gemappt ist.

Mapping der Identifikator-Eigenschaft

Eine reguläre (nicht zusammengesetzte) Identifikator-Eigenschaft wird in Hibernate XML-Dateien mit dem `<id>`-Element gemappt:

```
<class name="Category" table="CATEGORY">
<id name="id" column="CATEGORY_ID" type="long">
    <generator class="native"/>
</id>

...
</class>
```

Die Identifikator-Eigenschaft wird auf die Primärschlüsselspalte `CATEGORY_ID` der Tabelle `CATEGORY` gemappt. Der Hibernate-Typ für diese Eigenschaft ist `long`, wird in den meisten Datenbanken auf den Spaltentyp `BIGINT` gemappt und ist auch so ausgewählt worden, um zum Typ des Identitätswerts zu passen, der vom `native` Identifikator-Generator produziert wurde. (Im nächsten Abschnitt besprechen wir Strategien zur Generierung von Identifikatoren.)

Für eine JPA-Entity-Klasse nutzen Sie Annotationen im Java-Quellcode, um die Identifikator-Eigenschaft zu mappen:

```
@Entity
@Table(name="CATEGORY")
public class Category {
    private Long id;
    ...

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "CATEGORY_ID")
    public Long getId() {
        return this.id;
    }

    private void setId(Long id) {
        this.id = id;
    }
    ...
}
```

Die `@Id`-Annotation der Getter-Methode markiert sie als Identifikator-Eigenschaft, und `@GeneratedValue` mit der Option `GenerationType.AUTO` übersetzt in eine native Strategie zur Identifikator-Generierung, wie die Option `native` in XML-Hibernate-Mappings. Beachten Sie, dass der Default ebenfalls `GenerationType.AUTO` ist, wenn Sie keine `strategy` definieren, also könnten Sie dieses Attribut auch ganz weglassen. Sie geben auch eine Datenbankspalte an – anderenfalls würde Hibernate den Eigenschaftsnamen nehmen. Der Mapping-Typ wird durch den Java-Eigenschaftstyp `java.lang.Long` impliziert.

Natürlich können Sie auch für alle Eigenschaften einen direkten Feldzugriff einschließlich des Datenbank-Identifikators verwenden:

```
@Entity
@Table(name="CATEGORY")
public class Category {

    @Id @GeneratedValue
    @Column(name = "CATEGORY_ID")
    private Long id;
    ...

    public Long getId() {
        return this.id;
    }
    ...
}
```

Mapping-Annotationen werden wie im Standard definiert bei der Feld-Deklaration platziert, wenn der direkte Feldzugriff aktiviert wird.

Ob Feld- oder Eigenschaftszugriff für eine Entity aktiviert ist, hängt von der Position der vorgeschriebenen `@Id`-Annotation ab. Im vorigen Beispiel ist sie bei einem Feld vorhanden, also wird Hibernate auf alle Attribute der Klasse über Felder zugreifen. Beim Beispiel davor, bei dem auf der Methode `getId()` annotiert wurde, wird der Zugriff auf alle Attribute über Getter- und Setter-Methoden aktiviert.

Alternativ können Sie JPA-XML-Deskriptoren nehmen, um Ihr Identifikator-Mapping zu erstellen:

```
<entity class="auction.model.Category" access="FIELD">
    <table name="CATEGORY"/>
    <attributes>
        <id name="id">
            <generated-value strategy="AUTO"/>
        </id>
        ...
    </attributes>
</entity>
```

Zusätzlich zu Operationen, bei denen die Java-Objektidentität (`a == b`) und die Objektgleichheit (`a.equals(b)`) geprüft wird, können Sie nun eine `a.getId().equals( b.getId() )` verwenden, um die Datenbankidentität zu prüfen. Was haben diese Konzepte gemeinsam? In welchen Situationen geben sie alle `true` zurück? Wenn alle wahr sind, nennt man diesen Zeitraum den Bereich der garantierten Objektidentität. Wir kommen auf dieses Thema in Kapitel 9, Abschnitt 9.2 „Objektidentität und Objektgleichheit“, zurück.

Die Arbeit mit Datenbank-Identifikatoren ist bei Hibernate einfach und unkompliziert. Die Wahl eines guten Primärschlüssels (und die Strategie zur Schlüsselgenerierung) könnte schwieriger sein. Dieses Problem wollen wir als Nächstes angehen.

4.2.3 Primärschlüssel für Datenbanken

Hibernate muss wissen, wie Sie bei der Generierung von Primärschlüsseln am liebsten vorgehen. Von daher werden wir zuerst die Primärschlüssel definieren.

Auswahl eines Primärschlüssels

Der Kandidat für den Schlüssel ist eine Spalte oder eine Gruppe von Spalten, die verwendet werden kann, um eine spezielle Zeile in einer Tabelle zu identifizieren. Um ein Primärschlüssel zu werden, muss der Kandidat die folgenden Eigenschaften erfüllen:

- Sein Wert (für jede Spalte des Schlüsselkandidaten) ist nie null.
- Jede Zeile hat einen eindeutigen Wert.
- Der Wert einer speziellen Zeile verändert sich nie.

Wenn eine Tabelle nur ein Attribut zur Identifizierung hat, ist es der Definition nach der Primärschlüssel. Allerdings können mehrere Spalten oder Kombinationen von Spalten diese Eigenschaft bei einer bestimmten Tabelle erfüllen. Sie entscheiden sich für den Schlüsselkandidaten, um für die Tabelle den besten Primärschlüssel zu bekommen. Schlüsselkandidaten, die nicht für Primärschlüssel gewählt wurden, sollten zu *unique* Schlüsseln in der Datenbank deklariert werden.

Viele SQL-Datenmodelle aus Altsystemen arbeiten mit natürlichen Primärschlüsseln. Ein natürlicher Schlüssel ist einer, der im Business-Modell eine Bedeutung hat: ein Attribut (oder eine Kombination von Attributen), das aufgrund seiner Business-Semantiken eindeutig ist. Beispiele solcher natürlicher Schlüssel sind die US-amerikanische Sozialversicherungsnummer oder auch Steuernummern. Es ist einfach, natürliche Schlüssel zu erkennen: Wenn das Attribut eines Kandidatenschlüssels außerhalb des Datenbankkontexts eine Bedeutung hat, ist es ein natürlicher Schlüssel, egal ob er automatisch generiert wurde oder nicht. Denken Sie an die Nutzer der Applikation: Wenn diese sich auf ein Schlüsselattribut beziehen, wenn sie über die Applikation sprechen oder mit ihr arbeiten, ist es ein natürlicher Schlüssel.

Die Erfahrung hat gezeigt, dass natürliche Schlüssel auf lange Sicht fast immer Probleme verursachen. Ein guter Primärschlüssel muss eindeutig, konstant und notwendig sein (nie null oder unbekannt). Nur wenige Entity-Attribute erfüllen diese Anforderungen, und manche, die es tun, können nicht effektiv von SQL-Datenbanken indexiert werden (obwohl das ein Implementierungsdetail ist und keine primäre Begründung für oder gegen einen bestimmten Schlüssel sein sollte). Obendrein sollten Sie sicherstellen, dass eine Definition für einen Schlüsselkandidaten sich niemals im Laufe der Lebenszeit der Datenbank ändern kann, bevor Sie ihn zum Primärschlüssel machen. Es ist eine frustrierende Aufgabe, den Wert (oder auch nur die Definition) eines Primärschlüssels und aller Fremdschlüssel, die sich auf ihn beziehen, zu ändern. Obendrein sind natürliche Schlüsselkandidaten oft nur durch die Kombination mehrerer Spalten in einem zusammengesetzten natürlichen Schlüssel zu erhalten. Obwohl sie gewiss für bestimmte Beziehungen passend sind (wie für eine Link-Tabelle in einer many-to-many-Beziehung), erschweren diese zusammengesetzten Schlüssel gewöhnlich die Wartung, die ad-hoc-Abfragen und die Schemaevolution.

Aus diesen Gründen empfehlen wir mit Nachdruck, dass Sie mit synthetischen Identifikatoren arbeiten, auch Surrogatschlüssel genannt. Surrogatschlüssel haben keine Business-Bedeutung – sie sind eindeutige Werte, die von der Datenbank oder Applikation generiert wurden. Die Anwender einer Applikation sehen diese Schlüsselwerte idealerweise nicht

oder brauchen sich nicht darauf zu beziehen; sie sind Teil der Interna des Systems. Die Einführung einer Surrogatschlüsselspalte ist auch in einer anderen üblichen Situation angemessen: Wenn es keine Schlüsselkandidaten gibt, ist eine Tabelle per Definition keine Beziehung, wie es durch das relationale Modell definiert wird – sie erlaubt doppelte Zeilen –, und so müssen Sie eine Surrogatschlüsselspalte einfügen. Es gibt verschiedene bekannte Vorgehensweisen, um Werte für Surrogatschlüssel zu generieren.

Wahl eines Schlüsselgenerators

Hibernate hat mehrere eingebaute Strategien für die Generierung von Identifikatoren. Wir führen die nützlichsten Optionen in Tabelle 4.1 auf.

Tabelle 4.1 Die in Hibernate eingebauten Module zur Identifikator-Generierung

Name des Generators	JPA-GenerationType	Optionen	Beschreibung
`native`	`AUTO`	–	Der `native`-Identitätsgenerator wählt abhängig von den Fähigkeiten der zugrunde liegenden Datenbank andere Identitätsgeneratoren wie `identity`, `sequence` oder `hilo`. Nehmen Sie diesen Generator, um Ihre Mapping-Metadaten auf verschiedene Datenbankmanagementsysteme portierbar zu halten.
`identity`	`IDENTITY`	–	Dieser Generator unterstützt Identitätsspalten in DB2, MySQL, MS SQL Server, Sybase und HypersonicSQL. Der zurückgegebene Identifikator ist vom Typ `long`, `short` oder `int`.
`sequence`	`SEQUENCE`	`sequence`, `parameters`	Dieser Generator erstellt eine Sequenz in DB2, PostgreSQL, Oracle, SAP DB oder McKoi oder ein Generator in Interbase wird verwendet. Der zurückgegebene Identifikator ist vom Typ `long`, `short` oder `int`. Nehmen Sie die Option `sequence`, um einen Katalognamen für die Sequenz zu definieren (Default ist `hibernate_sequence`) und `parameters`, wenn Sie zusätzliche Einstellungen brauchen, um eine Sequenz zu erstellen, die der DDL hinzugefügt werden kann.
`increment`	(Nicht verfügbar)	–	Beim Start von Hibernate liest dieser Generator den maximalen (numerischen) Wert der Primärschlüsselspalte der Tabelle und inkrementiert ihn jedes Mal um eins, wenn eine neue Zeile eingefügt wird. Der zurückgegebene Identifikator ist vom Typ `long`, `short` oder `int`. Dieser Generator ist vor allem dann effizient, wenn die Einzelserver- Hibernate- Applikation den exklusiven Zugriff auf die Datenbank hat, aber in keinem anderen Szenario verwendet werden sollte.
`hilo`	(Nicht verfügbar)	`table`, `column`, `max_lo`	Ein High/Low-Algorithmus ist ein effektiver Weg, um Identifikatoren des Typs `long` zu generieren, wenn eine Tabelle und Spalte (als Default `hibernate_unique_key` bzw. `next`) als Quelle hoher Werte gegeben ist. Der High/Low-Algorithmus generiert Identifikatoren, die nur für eine bestimmte Datenbank eindeutig sind. Hohe Werte werden aus einer globalen Quelle ausgelesen und durch Hinzufügen eines lokalen niedrigen Werts eindeutig

Name des Generators	JPA-GenerationType	Optionen	Beschreibung
			gemacht. Dieser Algorithmus verhindert Überlast, wenn auf eine Quelle für Identifikatorwerte für viele Einfügungen zugegriffen werden muss. In „Data Modeling 101“ (Ambler, 2002) finden Sie mehr Informationen über den High/Low-Ansatz für eindeutige Identifikatoren. Dieser Generator muss von Zeit zu Zeit eine separate Datenbankverbindung benutzen, also kann er nicht über Datenbankverbindungen verwendet werden, die vom Anwender aufgebaut werden. Anders gesagt sollten Sie ihn nicht mit `sessionFactory.openSession(myConnection)` verwenden. Die Option `max_lo` definiert, wie viele niedrige Werte hinzugefügt werden, bis ein neuer hoher Wert geholt wird. Nur Einstellungen höher als 1 sind sinnvoll, der Default ist 32767 (`Short.MAX_VALUE`).-
`seqhilo`	(Nicht verfügbar)	`sequence, parameters, max_lo`	Dieser Generator funktioniert wie der reguläre `hilo`-Generator, außer dass er eine benannte Datenbanksequenz benutzt, um hohe Werte zu generieren.
(Nur `JPA`)	`TABLE`	`table, catalog, schema, pkColumnName, valueColumnName, pkColumnValue, allocationSize`	Ähnlich wie die `hilo`-Strategie von Hibernate verlässt sich `TABLE` auf eine Datenbanktabelle, in der der zuletzt generierte Integer-Primärschlüsselwert enthalten ist, und jeder Generator wird auf eine Zeile in dieser Tabelle gemappt. Jede Zeile hat zwei Spalten: `pkColumnName` und `valueColumnName`. `pkColumnValue` weist jede Zeile einem bestimmten Generator zu, und die Wert-Spalte enthält den zuletzt ausgelesenen Primärschlüssel. Der Persistenz-Provider alloziert bei jedem Durchlauf bis zu `allocationSize` Integer.
`uuid.hex`	(Nicht verfügbar)	`separator`	Dieser Generator ist ein 128-Bit-UUID (ein Algorithmus, der Identifikatoren des Typs `string` generiert, die in einem Netzwerk eindeutig sind). Die IP-Adresse wird zusammen mit einem eindeutigen Zeitstempel verwendet. Die UUID wird als String von hexadezimalen Ziffern der Länge 32 codiert, wobei sich ein optionaler `separator`-String zwischen jeder Komponente der UUID-Repräsentation befindet. Arbeiten Sie mit dieser Generator-Strategie nur, wenn Sie global eindeutige Identifikatoren brauchen, z.B. wenn Sie regelmäßig Datenbanken zusammenführen müssen.
`guid`	(Nicht verfügbar)	-	Mit diesem Generator bekommen Sie einen datenbankgenerierten, global eindeutigen Identifikator-String bei MySQL und SQL Server.
`select`	(Nicht verfügbar)	`key`	Dieser Generator liest einen Primärschlüssel aus, der von einem Datenbank-Trigger zugewiesen wird, indem die Zeile durch einen eindeutigen Schlüssel ausgewählt und der Primärschlüsselwert ausgelesen wird. Eine zusätzliche eindeutige Spalte mit einem Kandidatenschlüssel ist für diese Strategie erforderlich, und die Option `key` muss auf den Namen der Spalte der eindeutigen Schlüssel gesetzt sein.

Manche der eingebauten Identifikator-Generatoren können über Optionen konfiguriert werden. Bei einem nativen Hibernate XML-Mapping definieren Sie Optionen als Paare von Schlüsseln und Werten:

```
<id column="MY_ID">
    <generator class="sequence">
        <para name="sequence">MY_SEQUENCE</para>
        <para name="parameters">
            INCREMENT BY 1 START WITH 1
        </para>
    </generator>
</id>
```

Sie können die Identifikator-Generatoren von Hibernate mit Annotationen verwenden, auch wenn keine direkte Annotation verfügbar ist:

```
@Entity
@org.hibernate.annotations.GenericGenerator(
    name = "hibernate-uuid",
    strategy = "uuid"
)
class name MyEntity {

    @Id
    @GeneratedValue(generator = "hibernate-uuid")
    @Column(name = "MY_ID")
    String id;
}
```

Die Hibernate-Extension `@GenericGenerator` kann verwendet werden, um einem Identifikator-Generator von Hibernate einen Namen zu geben, in diesem Fall `hibernate-uuid`. Auf diesen Namen wird dann vom standardisierten `generator`-Attribut referenziert.

Diese Deklaration eines Generators und seine Zuweisung nach Namen muss auch für sequenz- oder tabellenbasierte Identifikator-Generierung mit Annotationen angewendet werden. Nehmen wir an, dass Sie in allen Ihren Entity-Klassen einen benutzerdefinierten Sequenz-Generator verwenden wollen. Weil dieser Identifikator-Generator global sein muss, wird er in orm.xml deklariert:

```
<sequence-generator name="mySequenceGenerator"
    sequence-name="MY_SEQUENCE"
    initial-value="123"
    allocation-size="20"/>
```

Damit wird deklariert, dass eine Datenbanksequenz namens `MY_SEQUENCE` mit einem anfänglichen Wert von `123` als Quelle für die Generierung von Datenbank-Identifikatoren verwendet werden soll, und dass die Persistenz-Engine sich jedes Mal 20 Werte holen soll, wenn sie Identifikatoren braucht. (Beachten Sie allerdings, dass Hibernate Annotations – während wir dieses schreiben – die Einstellung `initialValue` ignoriert.)

Um diesen Identifikator-Generator auf eine bestimmte Entity anzuwenden, verwenden Sie seinen Namen:

```
@Entity
class name MyEntity {

    @Id @GeneratedValue(generator = "mySequenceGenerator")
    String id;
}
```

Wenn Sie einen weiteren Generator mit dem gleichen Namen auf der Entity-Stufe vor dem Schlüsselwort `class` deklariert hätten, würde der globale Identifikator-Generator überschrieben. Der gleiche Ansatz kann verwendet werden, um einen `@TableGenerator` zu deklarieren und anzuwenden.

Sie sind nicht auf diese eingebauten Strategien festgelegt, sondern können durch Implementierung des Interface `IdentifierGenerator` von Hibernate eigene Identifikator-Generatoren erstellen. Wie immer ist es eine ganz gute Strategie, sich als Inspiration den Quellcode der vorhandenen Identifikator-Generatoren von Hibernate anzuschauen.

Es ist auch möglich, die Identifikator-Generatoren für Persistenzklassen in einem Domain-Modell zu mischen, doch für Daten, die nicht aus Altsystemen stammen, empfehlen wir, bei allen Entities die gleiche Strategie für die Generierung von Identifikatoren zu nehmen.

Bei Daten aus Altsystemen und von der Applikation zugewiesenen Identifikatoren wird das Bild etwas komplizierter. In diesem Fall sind wir oft auf natürliche Schlüssel und vor allem zusammengesetzte Schlüssel festgelegt. Ein zusammengesetzter Schlüssel ist ein natürlicher Schlüssel, der aus verschiedenen Tabellenspalten besteht. Weil es etwas schwerer sein kann, mit zusammengesetzten Identifikatoren zu arbeiten, die oft nur in Schemata von Altsystemen auftauchen, besprechen wir diese nur im Kontext von Kapitel 8, Abschnitt 8.1 „Integration von Datenbanken aus Altsystemen“.

Wir gehen von nun an davon aus, dass Sie den Entity-Klassen Ihres Domain-Modells Identifikator-Eigenschaften hinzugefügt haben, und dass Sie, nachdem Sie das grundlegende Mapping jeder Entity und deren Identifikator-Eigenschaften abgeschlossen haben, mit dem Mapping der Wert-Typ-Eigenschaften der Entities fortfahren. Allerdings können einige spezielle Optionen Ihre Klassen-Mappings vereinfachen oder erweitern.

4.3 Optionen für das Mapping von Klassen

Wenn Sie die `<hibernate-mapping>`- und `<class>`-Elemente in der DTD (oder der Referenzdokumentation) durchsehen, finden Sie ein paar Optionen, die bisher noch nicht angesprochen wurden:

- Dynamische Generierung von CRUD-SQL-Anweisungen
- Konfiguration der Veränderbarkeit von Entities
- Benennung von Entities für Abfragevorgänge
- Mapping von Paketnamen
- Anführungszeichen bei Schlüsselwörtern und reservierten Datenbank-Identifikatoren
- Implementierung von Namenskonventionen bei Datenbanken

4.3.1 Dynamische SQL-Generierung

Standardmäßig erstellt Hibernate beim Startup SQL-Anweisungen für jede Persistenzklasse. Diese Anweisungen sind einfache Operationen für Erstellen, Lesen, Aktualisieren und Löschen für das Lesen, Löschen etc. einer Zeile.

Wie kann Hibernate eine UPDATE-Anweisung beim Startup erstellen? Immerhin sind die Spalten, die aktualisiert werden sollen, zu dieser Zeit noch gar nicht bekannt. Die Antwort lautet, dass die generierte SQL-Anweisung alle Spalten aktualisiert, und wenn der Wert einer bestimmten Spalte nicht modifiziert wird, setzt die Anweisung sie auf ihren alten Wert.

In manchen Situationen wie bei Tabellen aus Altsystemen mit Hunderten von Spalten, bei denen die SQL-Anweisungen auch für die einfachsten Operationen recht groß sein werden (gehen wir mal davon aus, dass nur eine Spalte aktualisiert werden muss), müssen Sie diese SQL-Generierung beim Startup abschalten und statt dessen dynamische Anweisungen verwenden, die zur Laufzeit generiert werden. Eine extrem große Zahl von Entities kann sich auch auf die Startup-Dauer auswirken, weil Hibernate im Vorfeld alle SQL-Anweisungen für CRUD generieren muss. Der Speicherverbrauch für diese Abfrageanweisung wird auch sehr hoch sein, wenn ein Dutzend Anweisungen für Tausende von Entities gecachet werden muss (das ist gewöhnlich aber kein Problem).

Im Mapping-Element <class> sind zwei Attribute für die Deaktivierung von CRUD-SQL-Generierung beim Startup verfügbar:

```
<class name="Item"
   dynamic-insert="true"
   dynamic-update="true">
...
</class>
```

Das Attribut dynamic-insert sagt Hibernate, ob Eigenschaftswerte mit null bei einem SQL-INSERT eingeschlossen werden sollen, und das Attribut dynamic-update informiert Hibernate, ob nicht-modifizierte Eigenschaften in das SQL-UPDATE eingeschlossen werden sollen.

Wenn Sie JDK 5.0-Annotation-Mappings verwenden, brauchen Sie eine native Hibernate-Annotation, um die dynamische SQL-Generierung zu aktivieren:

```
@Entity
@org.hibernate.annotations.Entity(
    dynamicInsert = true, dynamicUpdate = true
)
public class Item { ...
```

Die zweite Annotation @Entity aus dem Hibernate-Paket erweitert die JPA-Annotation um zusätzliche Optionen, einschließlich dynamicInsert und dynamicUpdate.

Manchmal können Sie es vermeiden, eine UPDATE-Anweisung generieren zu müssen, wenn die Persistenzklasse als unveränderlich gemappt ist.

4.3.2 Eine Entity unveränderlich machen

Instanzen einer bestimmten Klasse können unveränderlich sein. In CaveatEmptor ist zum Beispiel ein Bid (Gebot) für einen Artikel unveränderlich. Von daher muss niemals eine UPDATE-Anweisung in der BID-Tabelle ausgeführt werden. Hibernate kann auch ein paar andere Optimierungen vornehmen, beispielsweise das Vermeiden von Dirty Checking, wenn Sie eine unveränderliche Klasse mappen, bei der das Attribut mutable auf false gesetzt ist:

```
<hibernate-mapping default-access="field">
    <class name="Bid" mutable="false">
    ...
    </class>
</hibernate-mapping>
```

Ein POJO ist unveränderlich, wenn keine öffentlichen Setter-Methoden für Eigenschaften der Klasse vorhanden sind – alle Werte werden im Konstruktor gesetzt. Anstatt privater Setter-Methoden werden Sie es oft vorziehen, dass Hibernate einen direkten Feldzugriff für unveränderliche Persistenzklassen hat, von daher brauchen Sie dann keine nutzlosen Zugriffsmethoden schreiben. Sie können eine unveränderliche Entity über Annotationen mappen:

```
@Entity
@org.hibernate.annotations.Entity(mutable = false)
@org.hibernate.annotations.AccessType("field")
public class Bid { ...
```

Auch hier erweitert die native `@Entity`-Annotation von Hibernate die JPA-Annotation wieder mit zusätzlichen Optionen. Wir haben hier auch die Hibernate Extension-Annotation `@AccessType` gezeigt – das ist eine Annotation, die Sie selten verwenden werden. Wie bereits erklärt, wird die Default-Zugriffsstrategie für eine bestimmte Entity-Klasse von der Position der erforderlichen `@Id`-Eigenschaft vorgegeben. Allerdings können Sie `@AccessType` nehmen, um eine feiner granulierte Strategie zu erzwingen; sie kann bei Klassendeklarationen (wie im vorigen Beispiel) oder sogar bei bestimmten Feldern oder Zugriffsmethoden eingesetzt werden.

Schauen wir uns kurz ein weiteres Thema an: die Benennung von Entities für Abfragen.

4.3.3 Bezeichnung von Entities für Abfragen

Standardmäßig werden alle Klassennamen automatisch in den Namensraum der Hibernate-Abfragesprache HQL „importiert". Dadurch können Sie in HQL also die kurzen Klassennamen ohne ein Paketpräfix verwenden, was ganz praktisch ist. Allerdings kann dieser Auto-Import auch abgeschaltet werden, wenn zwei Klassen mit dem gleichen Namen bei einer bestimmten `SessionFactory` vorkommen, vielleicht in unterschiedlichen Paketen des Domain-Modells.

Wenn ein solcher Konflikt existiert und Sie die Default-Einstellungen nicht ändern, weiß Hibernate nicht, auf welche Klasse Sie sich in HQL beziehen. Sie können den Auto-Import von Namen in den HQL-Namensraum für bestimmte Mapping-Dateien durch die Einstellung `auto-import="false"` im root-Element `<hibernate-mapping>` ausschalten.

Entity-Namen können auch explizit in den HQL-Namensraum importiert werden. Sie können sogar Klassen und Interfaces importieren, die nicht explizit gemappt sind, damit eine Kurzform in polymorphen HQL-Abfragen benutzt werden kann:

```
<hibernate-mapping>
    <import class="auction.model.Auditable" rename="IAuditable"/>
</hibernate-mapping>
```

Sie können nun eine HQL-Abfrage wie `from IAuditable` benutzen, um alle persistenten Instanzen von Klassen auszulesen, die das Interface `auction.model.Auditable` imple-

mentieren. (Keine Sorge, wenn Sie an diesem Punkt noch nicht wissen, ob dieses Feature für Sie relevant ist; wir kommen später noch einmal auf Abfragen zurück.) Beachten Sie, dass das Element `<import>` wie alle anderen direkten Child-Element von `<hibernate-mapping>` eine applikationsweite Deklaration ist. Von daher müssen (und können) Sie das nicht in anderen Mapping-Dateien duplizieren.

Mit Annotationen können Sie einer Entity explizit einen Namen geben, wenn die Kurzform zu einer Kollision im JPA QL- oder HQL-Namensraum führen würde:

```
@Entity(name="AuctionItem")
public class Item { ... }
```

Schauen wir uns noch einen weiteren Aspekt von Bezeichnungen an: die Deklaration von Paketen.

4.3.4 Deklaration eines Paketnamens

Alle Persistenzklassen von CaveatEmptor sind im Java-Paket `auction.model` deklariert. Allerdings möchten Sie sicher nicht den vollständigen Paketnamen immer wiederholen, wo diese oder irgendeine andere Klasse in einem Assoziations-, Unterklassen- oder Komponenten-Mapping bezeichnet wird. Stattdessen geben Sie ein `package`-Attribut an:

```
<hibernate-mapping package="auction.model">
    <class name="Item" table="ITEM">
      ...
    </class>
</hibernate-mapping>
```

Nun bekommen alle nichtqualifizierten Klassennamen, die in diesem Mapping-Dokument erscheinen, den deklarierten Paketnamen als Präfix. Wir gehen von dieser Einstellung in allen Mapping-Beispielen dieses Buches aus und verwenden für die Modellklassen von CaveatEmptor nichtqualifizierte Namen.

Namen von Klassen und Tabellen müssen sehr sorgfältig ausgewählt werden. Ein Name, den Sie ausgewählt haben, kann auch vom SQL-Datenbanksystem reserviert worden sein, in dem Fall muss der Name in Anführungszeichen erscheinen.

4.3.5 Quoting von SQL-Identifikatoren

Standardmäßig setzt Hibernate die Namen von Tabellen und Spalten im generierten SQL nicht in Anführungszeichen. Das macht das SQL etwas leichter lesbar, und Sie können sich die Tatsache zunutze machen, dass die meisten SQL-Datenbanken *case sensitive* sind, wenn sie nicht in Anführungszeichen stehende Identifikatoren vergleichen. Gelegentlich – vor allem in den Datenbanken von Altsystemen – stoßen Sie auf Identifikatoren mit merkwürdigen Zeichen oder Leerzeichen oder Sie möchten die Beachtung von Groß- und Kleinschreibung erzwingen. Oder – wenn Sie sich auf die Defaults von Hibernate verlassen – der Name einer Klasse oder Eigenschaft in Java könnte automatisch in einen Tabellen- oder Spaltennamen übersetzt worden sein, der in Ihrem Datenbankmanagementsystem nicht erlaubt ist. Die Klasse `User` könnte beispielsweise auf eine Tabelle `USER` gemappt

sein, was gewöhnlich ein reserviertes Schlüsselwort in SQL-Datenbanken ist. Hibernate kennt keine SQL-Schlüsselworte von Datenbankmanagementprodukten, von daher wird das Datenbanksystem beim Startup oder während der Laufzeit eine Exception werfen.

Wenn Sie im Mapping-Dokument einen Tabellen- oder Spaltennamen in einfache Anführungszeichen (*backticks*) setzen, wird Hibernate diesen Identifikator im generierten SQL immer in Anführungszeichen setzen. Die folgende Eigenschafts-Deklaration zwingt Hibernate dazu, SQL zu generieren, bei dem der Spaltenname `"DESCRIPTION"` in Anführungszeichen gesetzt ist. Hibernate weiß auch, dass Microsoft SQL Server die Variante `[DESCRIPTION]` braucht und dass MySQL `` `DESCRIPTION` `` erwartet.

```
<property name="description"
          column="`DESCRIPTION`"/>
```

Es gibt keine Möglichkeit, Hibernate dazu zu zwingen, überall Identifikatoren in Anführungszeichen zu verwenden (abgesehen davon, alle Tabellen- und Spaltennamen in einfache Anführungszeichen zu setzen). Sie sollten sich überlegen, ob Sie, wo immer es möglich ist, die Tabellen oder Spalten mit reservierten Schlüsselwortnamen umbenennen. Das Setzen in einfache Anführungszeichen funktioniert bei Annotations-Mappings, doch es ist ein Implementierungsdetail von Hibernate und nicht Bestandteil der JPA-Spezifikation.

4.3.6 Implementierung von Namenskonventionen

Wir haben oft mit Organisationen zu tun, die strenge Konventionen für die Namen von Datenbanktabellen und -spalten haben. Hibernate bietet ein Feature, mit dem Sie bei den Bezeichnungen automatisch bestimmte Standards erzwingen können.

Nehmen wir an, dass alle Tabellennamen in CaveatEmptor dem Muster `CE_<table name>` folgen sollen. Eine Lösung ist, manuell das Attribut `table` bei allen `<class>`- und Collection-Elementen der Mapping-Dateien anzugeben. Doch ein solches Vorgehen ist zeitaufwändig und gerät leicht in Vergessenheit. Stattdessen können Sie das Hibernate-Interface `NamingStrategy` implementieren (siehe Listing 4.1).

Listing 4.1 Implementierung von `NamingStrategy`

```
public class CENamingStrategy extends ImprovedNamingStrategy {

      public String classToTableName(String className) {
            return StringHelper.unqualify(className);
      }
      public String propertyToColumnName(String propertyName) {
          return propertyName;
    }
      public String tableName(String tableName) {
            return "CE_" + tableName;
      }
      public String columnName(String columnName) {
          return columnName;
    }
      public String propertyToTableName(String className,
                                        String propertyName) {
            return "CE_"
                       + classToTableName(className)
                       + '_'
```

```
                                  + propertyToColumnName(propertyName);
        }
}
```

Sie erweitern die `ImprovedNamingStrategy`, die Default-Implementierungen für alle Methoden von `NamingStrategy` bietet, die Sie nicht von Grund auf neu implementieren wollen (schauen Sie sich die API-Dokumentation und den Quellcode an). Die Methode `classToTableName()` wird nur aufgerufen, wenn ein `<class>`-Mapping keinen expliziten `table`-Namen angibt. Die Methode `propertyToColumnName()` wird aufgerufen, wenn eine Eigenschaft keinen expliziten `column`-Namen hat. Die Methoden `tableName()` und `columnName()` werden aufgerufen, wenn ein expliziter Name deklariert wird.

Wenn Sie diese `CENamingStrategy` aktivieren, führt die Klassen-Mapping-Deklaration

```
<class name="BankAccount">
```

zu `CE_BANKACCOUNT` als Namen der Tabelle.

Wenn allerdings ein Tabellenname wie hier angegeben ist:

```
<class name="BankAccount" table="BANK_ACCOUNT">
```

dann ist `CE_BANK_ACCOUNT` der Name der Tabelle. In diesem Fall wird `BANK_ACCOUNT` der Methode `tableName()` übergeben.

Das beste Feature des Interfaces `NamingStrategy` ist das Potenzial für ein dynamisches Verhalten. Um eine spezielle Namensstrategie zu aktivieren, können Sie beim Startup eine Instanz der Hibernate-`Configuration` übergeben:

```
Configuration cfg = new Configuration();
cfg.setNamingStrategy( new CENamingStrategy() );
SessionFactory sessionFactory sf =
                    cfg.configure().buildSessionFactory();
```

So können Sie mehrere `SessionFactory`-Instanzen haben, die auf den gleichen Mapping-Dokumenten basieren und jeweils eine andere `NamingStrategy` verwenden. Das ist bei einer Installation mit mehreren Clients äußerst hilfreich, bei der eindeutige Tabellennamen (aber das gleiche Datenmodell) für jeden Client erforderlich sind. Allerdings sollten Sie bei dieser Art von Anforderung besser mit einem SQL-Schema (einer Art Namensraum) arbeiten, wie bereits in Kapitel 3, Abschnitt 3.3.4 „Umgang mit globalen Metadaten“, besprochen.

Sie können über die Option `hibernate.ejb.naming_strategy` in Ihrer Datei `persistence.xml` die Implementierung einer Namensstrategie für Java Persistence angeben.

Nachdem wir nun die Konzepte und wichtigsten Mappings für Entities durchgegangen sind, sollen die Wert-Typen gemappt werden.

4.4 Feingranulierte Modelle und Mappings

Nachdem wir die erste Hälfte dieses Kapitels fast ausschließlich mit Entities bzw. den grundlegenden Optionen für das persistente Klassen-Mapping zugebracht haben, konzentrieren wir uns nun auf die verschiedenen Formen der Wert-Typen. Zwei verschiedene Arten kommen einem sofort in den Sinn: Klassen mit Wert-Typen, die im JDK enthalten sind (zum Beispiel `String` oder Primitive Datentypen), und wert-typisierte Klassen, die vom Applikationsentwickler definiert wurden (zum Beispiel `Address` und `MonetaryAmount`).

Zuerst mappen Sie die Eigenschaften der Persistenzklassen, die JDK-Typen verwenden, und lernen die grundlegenden Mapping-Elemente und -Attribute kennen. Dann geht es mit benutzerdefinierten Klassen mit Wert-Typen weiter und wie Sie diese als einbettbare Komponenten mappen.

4.4.1 Mapping von grundlegenden Eigenschaften

Wenn Sie eine Persistenzklasse mappen, egal ob sie eine Entity oder ein Wert-Typ ist, müssen alle Persistenzeigenschaften explizit in der XML-Mapping-Datei gemappt werden. Wenn andererseits eine Klasse mit Annotationen gemappt ist, werden alle ihre Eigenschaften standardmäßig als persistent betrachtet. Sie können Eigenschaften mit der Annotation `@javax.persistence.Transient` auszeichnen, um sie auszuschließen, oder das Java-Schlüsselwort `transient` verwenden (das normalerweise nur Felder für die Java-Serialisierung ausschließt).

In einem JPA-XML-Deskriptor können Sie ein bestimmtes Feld oder eine bestimmte Eigenschaft ausschließen:

```
<entity class="auction.model.User" access="FIELD">
    <attributes>
        ...
        <transient name="age"/>
    </attributes>
</entity>
```

Ein typisches Mapping von Eigenschaften bei Hibernate definiert den Eigenschaftsnamen eines POJOs, einen Spaltennamen in einer Datenbank und den Namen eines Hibernate-Typs. Oft ist es auch möglich, den Typ wegzulassen. Wenn `description` die Eigenschaft des (Java-)Typs `java.lang.String` ist, verwendet Hibernate standardmäßig den Hibernate-Typ `string` (wir kommen im nächsten Kapitel noch einmal auf das Hibernate-Typensytem zurück).

Hibernate arbeitet mit Reflektion, um den Java-Typ der Eigenschaft zu bestimmen. Von daher sind die folgenden Mappings gleichwertig:

```
<property name="description" column="DESCRIPTION" type="string"/>
<property name="description" column="DESCRIPTION"/>
```

Es ist sogar möglich, den Spaltennamen auszulassen, wenn es der gleiche wie der Eigenschaftsname ist (wobei Groß- und Kleinschreibung ignoriert wird). (Das ist einer der vernünftigen Defaults, die wir weiter oben erwähnt haben.)

Für ein paar ungewöhnliche Fälle, von denen Sie später noch mehr kennenlernen werden, könnten Sie in Ihrem XML-Mapping ein `<column>`-Element statt des `column`-Attributs nehmen. Das `<column>`-Element bietet mehr Flexibilität: Es hat mehr optionale Attribute und kann mehr als einmal erscheinen. (Eine einzelne Eigenschaft kann mehr als eine Spalte mappen; diese Technik besprechen wir im nächsten Kapitel.) Die folgenden beiden Eigenschafts-Mapping sind äquivalent:

```
<property name="description" column="DESCRIPTION" type="string"/>
<property name="description" type="string">
    <column name="DESCRIPTION"/>
</property>
```

Das `<property>`-Element (und vor allem das `<column>`-Element) definiert auch bestimmte Attribute, die hauptsächlich auf die automatische Generierung des Datenbankschemas angewandt werden. Wenn Sie nicht mit dem `hbm2ddl`-Tool arbeiten (siehe Kapitel 2, Abschnitt 2.14 „Starten und Testen der Applikation"), um das Datenbankschema zu generieren, können Sie diese einfach weglassen. Allerdings ist es vorzuziehen, wenigstens das Attribut `not-null` einzuschließen, weil Hibernate dann illegale null-Eigenschaftswerte melden kann, ohne über die Datenbank zu gehen:

```
<property name="initialPrice" column="INITIAL_PRICE" not-null="true"/>
```

JPA basiert auf einem Configuration-by-Exception-Modell, von daher können Sie sich auf die Defaults verlassen. Wenn die Eigenschaft einer Persistenzklasse nicht annotiert ist, gelten die folgenden Regeln:

- Wenn die Eigenschaft vom Typ JDK ist, ist sie automatisch persistent. Anders gesagt wird sie wie `<property name="propertyName"/>` in einer Hibernate XML-Mapping-Datei behandelt.
- Anderenfalls wird sie als Komponente der besitzenden Klasse gemappt, wenn die Klasse der Eigenschaft als `@Embeddable` annotiert ist. Das Einbetten von Komponenten werden wir später in diesem Kapitel besprechen.
- Wenn andernfalls der Eigenschaftstyp `Serializable` ist, wird der Wert in seiner serialisierten Form gespeichert. Das ist normalerweise nicht erwünscht, und Sie sollten Java-Klassen immer mappen, anstatt in der Datenbank einen Haufen Bytes zu speichern. Malen Sie sich einmal aus, eine Datenbank mit solchen binären Informationen zu pflegen, wenn in ein paar Jahren die Applikation weg ist.

Wenn Sie sich nicht auf diese Defaults verlassen wollen, wenden Sie die Annotation `@Basic` auf eine bestimmte Eigenschaft an. Die Annotation `@Column` ist die Entsprechung des XML-`<column>`-Elements. Hier ist ein Beispiel, wie Sie den Wert einer Eigenschaft wie erforderlich deklarieren:

```
@Basic(optional = false)
@Column(nullable = false)
public BigDecimal getInitialPrice { return initialPrice; }
```

Die `@Basic`-Annotation markiert die Eigenschaft als nicht optional auf der Stufe des Java-Objekts. Die zweite Einstellung des Spalten-Mappings, `nullable = false`, ist nur verantwortlich für die Generierung eines `NOT NULL`-Datenbank-Constraints. Die Hibernate

JPA-Implementierung behandelt beide Optionen auf jeden Fall in gleicher Weise, somit können Sie auch bloß eine dieser Annotationen zu diesem Zweck verwenden.

In einem JPA-XML-Deskriptor sieht dieses Mapping genauso aus:

```
<entity class="auction.model.Item" access="PROPERTY">
    <attributes>
        ...
        <basic name="initialPrice" optional="false">
            <column nullable="false"/>
        </basic>
    </attributes>
</entity>
```

Es gibt eine ganze Reihe von Optionen in Hibernate-Metadaten, um Schema-Constraints zu deklarieren, zum Beispiel `NOT NULL` bei einer Spalte. Außer für einfache *null-Prüfung beim Speichern von Objekten* werden sie jedoch nur gebraucht, um DDL zu produzieren, wenn Hibernate ein Datenbankschema aus Mapping-Metadaten exportiert. Wir besprechen die Anpassung von SQL einschließlich DDL in Kapitel 8, Abschnitt 8.3 „Verbesserung der Schema-DDL“. Andererseits enthält das Hibernate Annotations-Paket ein fortschrittlicheres und ausgefeilteres Framework zur Datenvalidierung, das Sie nicht nur zu Definierung von Datenbankschema-Constraints in DDL nehmen können, sondern auch zur Datenvalidierung zur Laufzeit. Das werden wir in Kapitel 17 besprechen.

Befinden sich Annotationen für Eigenschaften immer bei den Zugriffsmethoden?

Anpassung des Zugriffs auf Eigenschaften

Auf Eigenschaften einer Klasse greift die Persistenz-Engine entweder direkt (über Felder) oder indirekt (über Getter- und Setter- Zugriffsmethoden) zu. In XML-Mapping-Dateien steuern Sie die Default-Zugriffsstrategie für eine Klasse mit dem Attribut `default-access="field|property|noop|custom.Class"` des root-Elements `hibernate-mapping`. Eine annotierte Entity erbt den Default von der Position der erforderlichen `@Id`-Annotation. Wenn beispielsweise `@Id` für ein Feld und nicht für eine Getter-Methode deklariert wurde, werden alle anderen Annotationen für das Eigenschafts-Mapping wie der Name der Spalte für die Eigenschaft `description` des Elements ebenfalls für Felder deklariert:

```
@Column(name = "ITEM_DESCR")
private String description;

public String getDescription() { return description; }
```

Das ist das durch die JPA-Spezifikation definierte Default-Verhalten. Allerdings erlaubt Hibernate eine flexible Anpassung der Zugriffsstrategie mit der Annotation `@org.hibernate.annotations.AccessType(<strategy>)`:

- Wenn `AccessType` auf der Ebene Klasse/Entity gesetzt ist, wird entsprechend auf alle Attribute der Klasse der ausgewählten Strategie zugegriffen. Annotationen auf der Stufe der Attribute werden abhängig von der Strategie entweder bei Feld- oder Getter-Methoden erwartet. Diese Einstellung überschreibt alle Defaults aus der Position der Standard-`@Id`-Annotationen.

- Wenn bei einer Entity als Default oder explizit der Feld-Zugriff gesetzt ist, wechselt durch die Annotation `AccessType("property")` bei einem Feld für dieses spezielle Attribut zur Laufzeit auf den Zugriff über Getter/Setter-Methoden. Die Position der Annotation `AccessType` ist immer noch das Feld.
- Wenn bei einer Entity als Default oder explizit der Eigenschafts-Zugriff gesetzt ist, wechselt durch die Annotation `AccessType("field")` bei einer Getter-Methode auf dieses spezielle Attribut zur Laufzeit auf den Zugriff über ein Feld des gleichen Namens. Die Position der Annotation `AccessType` ist immer noch die Getter-Methode.
- Alle `@Embedded`-Klassen erben den Default oder die explizit deklarierte Zugriffsstrategie der besitzenden root-Entity-Klasse.
- Auf alle `@MappedSuperclass`-Eigenschaften wird mit dem Default oder der explizit deklarierten Zugriffsstrategie der gemappten Entity-Klasse zugegriffen.

Sie können die Zugriffsstrategien auf Eigenschaftsebene in Hibernate XML-Mappings mit dem Attribut `access` steuern:

```
<property name="description"
          column="DESCR"
          access="field"/>
```

Oder Sie setzen die Zugriffsstrategie für alle Klassen-Mappings in einem root-`<hibernate-mapping>`-Element mit dem Attribut `default-access`.

Neben Feld- und Eigenschaftszugriff kann als eine weitere Strategie auch `noop` nützlich sein. Damit wird eine Eigenschaft gemappt, die nicht in der Java-Persistenzklasse existiert. Das hört sich eigenartig an, doch damit können Sie sich auf diese „virtuelle" Eigenschaft in HQL-Abfragen beziehen (anders gesagt, Sie können die Datenbankspalte nur in HQL-Abfragen verwenden).

Wenn keine der eingebauten Zugriffsstrategien passend ist, könne Sie sich eine eigene für den Zugriff auf die Eigenschaften definieren, indem Sie das Interface `org.hibernate.property.PropertyAccessor` implementieren. Setzen Sie den (vollqualifizierten) Klassennamen auf das `access`-Mapping-Attribut oder die `@AccessType`-Annotation. Schauen Sie sich als Inspiration den Hibernate-Quellcode an, damit ist das eine ganz einfache Übung.

Manche Eigenschaften werden gar nicht auf eine Spalte gemappt. Insbesondere eine abgeleitete Eigenschaft entnimmt ihren Wert einem SQL-Ausdruck.

Abgeleitete Eigenschaften

Der Wert einer abgeleiteten Eigenschaft wird zur Laufzeit durch Evaluation eines Ausdrucks berechnet, den Sie im Attribut `formula` definieren. Sie können zum Beispiel die Eigenschaft `totalIncludingTax` mit einem SQL-Ausdruck mappen:

```
<property name="totalIncludingTax"
          formula="TOTAL + TAX_RATE * TOTAL"
          type="big_decimal"/>
```

Die gegebene SQL-Formel wird jedes Mal evaluiert, wenn die Entity aus der Datenbank ausgelesen wird (und zu keiner anderen Zeit, von daher könnte das Resultat veraltet sein, wenn andere Eigenschaften modifiziert werden). Die Eigenschaft hat kein Spaltenattribut (oder Unterelement) und erscheint nie in einem SQL-`INSERT` oder -`UPDATE`, nur in `SELECT`s. Formeln können sich auf Spalten der Datenbanktabelle beziehen, sie können SQL-Funktionen aufrufen und sogar SQL-Subselects enthalten. Der SQL-Ausdruck wird so wie er ist der zugrunde liegenden Datenbank übergeben; hierbei passiert es leicht, dass Sie Ihre Mapping-Datei für ein bestimmtes Datenbankprodukt erstellen, wenn Sie nicht aufpassen und sich auf herstellerspezifische Operatoren oder Schlüsselwörter verlassen.

Formeln gibt es auch mit einer Hibernate-Annotation:

```
@org.hibernate.annotations.Formula("TOTAL + TAX_RATE * TOTAL")
public BigDecimal getTotalIncludingTax() {
    return totalIncludingTax;
}
```

Das folgende Beispiel verwendet eine damit in Beziehung stehende Unterabfrage (*subselect*), um den Mittelwert aller Gebote für ein Element zu berechnen:

```
<property
   name="averageBidAmount"
   type="big_decimal"
   formula=
    "( select AVG(b.AMOUNT) from
       BID b where b.ITEM_ID = ITEM_ID )"/>
```

Beachten Sie, dass nichtqualifizierte Spaltennamen sich auf Spalten der Tabelle der Klasse beziehen, zu denen die abgeleitete Eigenschaft gehört.

Eine weitere besondere Art von Eigenschaften beruht auf datenbankgenerierten Werten.

Generierte und Default-Werte für Eigenschaften

Nehmen wir an, dass der Wert einer bestimmten Eigenschaft einer Klasse von der Datenbank generiert wird, gewöhnlich wenn die Entity-Zeile zum ersten Mal eingefügt wird. Typische, von der Datenbank generierte Werte sind der Zeitstempel der Erstellung, ein Standardpreis für einen Artikel und ein Trigger, der bei jeder Änderung ausgelöst wird.

Üblicherweise müssen die Hibernate-Applikationen Objekte „auffrischen“, in denen Eigenschaften enthalten sind, für die die Datenbank Werte generiert. Wenn Eigenschaften als generiert ausgezeichnet sind, kann die Applikation diese Verantwortlichkeit allerdings Hibernate überlassen. Wenn Hibernate ein SQL-`INSERT` oder -`UPDATE` für eine Entity ausgibt, die definierte generierte Eigenschaften hat, führt es gleich danach ein `SELECT` durch, um die generierten Werte auszulesen. Nehmen Sie den Switch `generated` bei einem `property`-Mapping, um diese automatische Auffrischung zu aktivieren:

```
<property name="lastModified"
          column="LAST_MODIFIED"
          update="false"
          insert="false"
          generated="always"/>
```

Eigenschaften, die als von der Datenbank generiert ausgezeichnet sind, müssen zusätzlich nicht-einfügbar und nicht-aktualisierbar sein, was Sie über die Attribute `insert` und `up-`

`date` steuern können. Wenn beide auf `false` gesetzt sind, erscheinen die Spalten der Eigenschaft nie in den `INSERT`- oder `UPDATE`-Anweisungen – der Eigenschafts-Wert ist *Nur lesen*. Auch fügen Sie gewöhnlich keine öffentliche Setter-Methode in Ihrer Klasse für eine unveränderliche Eigenschaft ein (und wechseln auf Feldzugriff).

Mit Annotationen deklarieren Sie die Unveränderbarkeit (und die automatische Auffrischung) mit der Hibernate-Annotation `@Generated`:

```
@Column(updatable = false, insertable = false)
@org.hibernate.annotations.Generated(
    org.hibernate.annotations.GenerationTime.ALWAYS
)
private Date lastModified;
```

Die verfügbaren Einstellungen sind `GenerationTime.ALWAYS` und `GenerationTime.INSERT`, und die entsprechenden Optionen in XML-Mappings lauten `generated="always"` und `generated="insert"`.

Ein Sonderfall der datenbankgenerierten Eigenschaftswerte sind Default-Werte. Sie wollen vielleicht eine Regel implementieren, dass jeder in einer Auktion angebotene Artikel mindestens einen Euro kosten soll. Zuerst fügen Sie das in Ihren Datenbankkatalog als Default-Wert für die Spalte `INITIAL_PRICE` ein:

```
create table ITEM (
    ...
    INITIAL_PRICE number(10,2) default '1',
    ...
);
```

Wenn Sie hbm2ddl, das Hibernate-Tool zum Schemaexport, verwenden, können Sie diesen Output durch Einfügen eines `default`-Attributs beim Eigenschafts-Mapping aktivieren:

```
<class name="Item" table="ITEM"
       dynamic-insert="true" dynamic-update="true">
    ...
    <property name="initialPrice" type="big_decimal">
        <column name="INITIAL_PRICE"
                default="'1'"
                generated="insert"/>
    </property>
    ...
 </class>
```

Beachten Sie, dass Sie auch das Erstellen von Anweisungen zum dynamischen Einfügen und Updaten aktivieren müssen, damit die Spalte mit dem Default-Wert nicht in jeder Anweisung enthalten ist, wenn dessen Wert `null` ist (anderenfalls würde statt des Default-Werts eine `NULL` eingefügt). Obendrein hätte eine persistent gemachte Instanz von `Item`, die aber noch nicht zur Datenbank geflusht und noch nicht aufgefrischt worden ist, nicht den Default-Wert-Satz bei der Objekteigenschaft. Anders gesagt: Sie müssen explizit einen Flush durchführen:

```
Item newItem = new Item(...);
session.save(newItem);

newItem.getInitialPrice();   // ist null
session.flush();             // ein INSERT auslösen
// Hibernate führt automatisch ein SELECT durch
newItem.getInitialPrice();   // ist 1EUR
```

Weil Sie `generated="insert"` gesetzt haben, weiß Hibernate, dass ein sofortiger zusätzlicher `SELECT` erforderlich ist, um den datenbankgenerierten Eigenschaftswert zu lesen.

Sie können Default-Spaltenwerte mit Annotationen als Teil der DDL-Definition für eine Spalte mappen:

```
@Column(name = "INITIAL_PRICE",
        columnDefinition = "number(10,2) default '1'")
@org.hibernate.annotations.Generated(
    org.hibernate.annotations.GenerationTime.INSERT
)
private BigDecimal initalPrice;
```

Das Attribut `columnDefinition` umfasst die vollständigen Eigenschaften für die Spalte DDL mit Datentyp und allen Constraints. Bedenken Sie, dass ein echter nicht-portierbarer SQL-Datentyp Ihr Annotations-Mapping an ein bestimmtes Datenbankmanagementsystem binden kann.

Wir kommen auf das Thema Constraints und DDL-Anpassung in Kapitel 8, Abschnitt 8.3 „Verbesserung der Schema-DDL“ zurück.

Als Nächstes mappen Sie benutzerdefinierte Klassen, die Wert-Typen sind. Sie finden diese ganz leicht in Ihren UML-Klassendiagrammen, wenn Sie nach einer zusammengesetzten Beziehung zwischen zwei Klassen suchen. Eine von ihnen ist eine abhängige Klasse, eine Komponente.

4.4.2 Mapping von Komponenten

Bisher waren alle Klassen des Objektmodells Entity-Klassen, jede mit einem eigenen Lebenszyklus und eigener Identität. Die `User`-Klasse hat allerdings eine besondere Art der Assoziation mit der Klasse `Address` (siehe Abbildung 4.2).

Um es in der Sprache des Object-Modellings auszudrücken, ist diese Assoziation eine Art Aggregation – eine *Teil-von*-Beziehung. Aggregation ist eine starke Form der Assoziation, und dazu gehören im Hinblick auf den Lebenszyklus von Objekten einige weitere Semantiken. In diesem Fall haben Sie sogar noch eine stärkere Form (Komposition), bei der der Lebenszyklus eines Teils vollständig vom Lebenszyklus des Ganzen abhängig ist.

Experten für das Object Modelling und UML-Designer behaupten, es gäbe keinen Unterschied zwischen dieser Komposition und anderen, schwächeren Stilen der Assoziation, wenn es um die eigentliche Java-Implementierung geht. Doch im Kontext von ORM ist da ein großer Unterschied: Eine zusammengesetzte Klasse ist oft ein möglicher Wert-Typ.

Sie mappen `Address` als Wert-Typ und `User` als eine Entity. Wirkt sich das auf die Implementierung der POJO-Klassen aus?

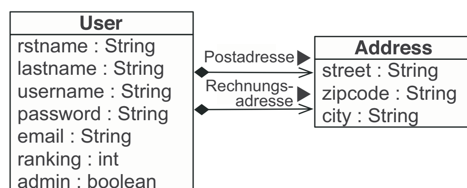

Abbildung 4.2
Beziehungen zwischen `User` und `Address` unter Verwendung von Komposition

Java hat kein Konzept von Komposition – eine Klasse oder ein Attribut kann nicht als Komponente oder Komposition ausgezeichnet werden. Der einzige Unterschied ist der Objekt-Identifikator: Eine Komponente hat keine individuelle Identität, von daher erfordert die persistente Komponentenklasse keine Identifikator-Eigenschaft oder kein Identifikator-Mapping. Es ist ein einfaches POJO:

```
public class Address {

    private String street;
    private String zipcode;
    private String city;
    public Address() {}
    public String getStreet() { return street; }
    public void setStreet(String street) { this.street = street; }
    public String getZipcode() { return zipcode; }
    public void setZipcode(String zipcode) {
        this.zipcode = zipcode; }
    public String getCity() { return city; }
    public void setCity(String city) { this.city = city; }
}
```

Die Komposition zwischen `User` und `Address` wird auf der Metadaten-Ebene festgelegt; Sie brauchen Hibernate nur zu sagen, dass die `Address` ein Wert-Typ im Mapping-Dokument oder mit Annotationen ist.

Mapping von Komponenten in XML

Hibernate verwendet den Begriff Komponente für eine benutzerdefinierte Klasse, die auf die gleiche Tabelle persistiert wie die besitzende Entity (ein Beispiel dafür sehen Sie in Listing 4.2). Die Verwendung des Worts Komponente hat hier nichts mit dem Konzept einer Architektur zu tun, so wie bei Software-Komponenten.)

Listing 4.2 Mapping der Klasse `User` mit einer Komponente `Address`

```
<class name="User" table="USER">

    <id name="id" column="USER_ID" type="long">
        <generator class="native"/>
    </id>
    <property name="loginName" column="LOGIN" type="string"/>
    <component name="homeAddress" class="Address">      ❶
        <property name="street" type="string"
                  column="HOME_STREET" not-null="true"/>
        <property name="city" type="string"
                  column="HOME_CITY" not-null="true"/>
        <property name="zipcode" type="string"
                  column="HOME_ZIPCODE" not-null="true"/>
    </component>
    <component name="billingAddress" class="Address">      ❷
        <property name="street" type="string"
                  column="BILLING_STREET" not-null="true"/>
        <property name="city" type="string"
                  column="BILLING_CITY" not-null="true"/>
        <property name="zipcode" type="string"
                  column="BILLING_ZIPCODE" not-null="true"/>
    </component>
    ...
</class>
```

❶ Sie deklarieren die Persistenzattribute von `Address` im Element `<component>`. Die Eigenschaft der Klasse `User` hat den Namen `homeAddress`.

❷ Sie verwenden die gleiche Komponentenklasse wieder, um eine andere Eigenschaft dieses Typs mit der gleichen Tabelle zu mappen.

Abbildung 4.3 zeigt, wie die Attribute der Klasse `Address` mit der gleichen Tabelle wie die Entity `User` persistiert werden.

<<Tabelle>>
USERS
FIRSTNAME
LASTNAME
USERNAME
PASSWORD
EMAIL
...
HOME_STREET
HOME_ZIPCODE
HOME_CITY
Komponenten-spalten
BILLING_STREET
BILLING_ZIPCODE
BILLING_CITY
Komponenten-spalten

Abbildung 4.3
Tabellenattribute von `User` mit der Komponente `Address`

Beachten Sie, dass Sie in diesem Beispiel die Assoziation der Komposition als unidirektional modellieren. Sie können nicht von `Address` in Richtung `User` navigieren. Hibernate unterstützt sowohl uni- als auch bidirektionale Kompositionen, doch unidirektionale sind weitaus mehr verbreitet. Ein Beispiel für ein bidirektionales Mapping sehen Sie in Listing 4.3.

Listing 4.3 Einen Back-Pointer in einer Komposition einfügen

```
<component name="homeAddress" class="Address">
    <parent name="user"/>
    <property name="street" type="string"
              column="HOME_STREET" not-null="true"/>
    <property name="city" type="string"
              column="HOME_CITY" not-null="true"/>
    <property name="zipcode" type="stringshort"
              column="HOME_ZIPCODE" not-null="true"/>
</component>
```

Im Listing 4.3 mappt das `<parent>`-Element eine Eigenschaft des Typs `User` mit der besitzenden Entity, die in diesem Beispiel die Eigenschaft namens `user` ist. Sie können dann `Address.getUser()` aufrufen, um in die andere Richtung zu navigieren. Das ist also eine ganz einfache Referenz auf das `<parent>`-Element (ein Back-Pointer).

Eine Hibernate-Komponente kann andere Komponenten besitzen und sogar Assoziationen zu anderen Entities haben. Diese Flexibilität ist die Grundlage für die Unterstützung von Hibernate für feingranulierte Objektmodelle. Sie können beispielsweise eine `Location`-Klasse mit detaillierten Informationen über die Postanschrift des `Address`-Besitzers erstellen:

```
<component name="homeAddress" class="Address">
    <parent name="user"/>
    <component name="location" class="Location">
        <property name="streetname" column="HOME_STREETNAME"/>
        <property name="streetside" column="HOME_STREETSIDE"/>
```

```
            <property name="housenumber" column="HOME_HOUSENR"/>
            <property name="floor" column="HOME_FLOOR"/>
        </component>
        <property name="city" type="string" column="HOME_CITY"/>
        <property name="zipcode" type="string" column="HOME_ZIPCODE"/>
    </component>
```

Das Design der Klasse `Location` ist genauso wie das der Klasse `Address`. Sie haben nun drei Klassen, eine Entity und zwei Wert-Typen, die alle auf die gleiche Tabelle gemappt sind.

Nun wollen wir die Komponenten mit JPA-Annotationen mappen.

Annotation von eingebetteten Klassen

Die Java Persistence-Spezifikation bezeichnet Komponenten als eingebettete Klassen. Um eine eingebettete Klasse mit Annotationen zu mappen, können Sie eine bestimmte Eigenschaft in der besitzenden Entity-Klasse als `@Embedded` deklarieren, in diesem Fall die `homeAddress` des `User`s:

```
@Entity
@Table(name = "USERS")
public class User {

    ...
    @Embedded
    private Address homeAddress;
    ...
}
```

Wenn Sie eine Eigenschaft nicht als `@Embedded` deklarieren und sie nicht vom Typ JDK ist, sucht Hibernate in den assoziierten Klassen nach der `@Embeddable`-Annotation. Wenn sie vorhanden ist, wird die Eigenschaft automatisch als eine abhängige Komponente gemappt.

So sieht eine eingebettete Klasse aus:

```
@Embeddable
public class Address {

    @Column(name = "ADDRESS_STREET", nullable = false)
    private String street;
    @Column(name = "ADDRESS_ZIPCODE", nullable = false)
    private String zipcode;
    @Column(name = "ADDRESS_CITY", nullable = false)
    private String city;
    ...
}
```

Sie können die individuellen Eigenschafts-Mappings in der eingebetteten Klasse noch weiter an Ihre Bedürfnisse anpassen, so wie mit der Annotation `@Column`. Die Tabelle `USERS` enthält nun unter anderem die Spalten `ADDRESS_STREET`, `ADDRESS_ZIPCODE` und `ADDRESS_CITY`. Jede andere Entity-Tabelle, die Komponentenfelder enthält (beispielsweise eine Klasse `Order`, die auch eine `Address` beinhaltet), verwendet die gleichen Spaltenoptionen. Sie können der einbettbaren Klasse `Address` auch eine Back-Pointer-Eigenschaft hinzufügen und sie mit `@org.hibernate.annotations.Parent` mappen.

Manchmal wollen Sie die in der einbettbaren Klasse vorgenommenen Einstellungen für eine bestimmte Entity auch von außen überschreiben. So können Sie beispielsweise die Spalten umbenennen:

```
@Entity
@Table(name = "USERS")
public class User {

    ...
    @Embedded
    @AttributeOverrides( {
        @AttributeOverride(name   = "street",
                           column = @Column(name="HOME_STREET") ),
        @AttributeOverride(name   = "zipcode",
                           column = @Column(name="HOME_ZIPCODE") ),
        @AttributeOverride(name   = "city",
                           column = @Column(name="HOME_CITY") )
    })
    private Address homeAddress;
    ...
}
```

Die neuen `@Column`-Deklarationen in der Klasse `User` überschreiben die Einstellungen der einbettbaren Klasse. Beachten Sie, dass alle Attribute bei der eingebetteten `@Column`-Annotation ersetzt werden und somit nicht länger `nullable = false` sind.

In einem JPA-XML-Deskriptor sieht das Mapping einer einbettbaren Klasse und einer Komposition wie folgt aus:

```
<embeddable class="auction.model.Address access-type="FIELD"/>

<entity class="auction.model.User" access="FIELD">
    <attributes>
        ...
        <embedded name="homeAddress">
            <attribute-override name="street">
                <column name="HOME_STREET"/>
            </attribute-override>
            <attribute-override name="zipcode">
                <column name="HOME_ZIPCODE"/>
            </attribute-override>
            <attribute-override name="city">
                <column name="HOME_CITY"/>
            </attribute-override>
        </embedded>
    </attributes>
</entity>
```

Es gibt zwei wichtige Einschränkungen für Klassen, die als Komponenten gemappt sind. Zum einen sind gemeinsame Verweise so wie für alle Wert-Typen nicht möglich. Die Komponente `homeAddress` hat keine eigene Datenbankidentität (Primärschlüssel), und somit kann kein anderes Objekt als die enthaltende Instanz von `User` darauf verweisen.

Zum anderen gibt es keinen eleganten Weg, um eine Null-Referenz für eine `Address` zu repräsentieren. Anstelle einer eleganten Vorgehensweise repräsentiert Hibernate eine Null-Komponente in allen gemappten Spalten der Komponente als Null-Werte. Das bedeutet, wenn Sie ein Komponentenobjekt mit allen Null-Eigenschaftswerten speichern, gibt Hibernate eine Null-Komponente zurück, wenn das besitzende Entity-Objekt aus der Datenbank ausgelesen wird.

Sie finden eine Vielzahl weiterer Komponenten-Mappings (sogar ganze Collections davon) im weiteren Verlauf des Buches.

4.5 Zusammenfassung

In diesem Kapitel haben Sie den wesentlichen Unterschied zwischen Entities und Wert-Typen kennengelernt und wie diese Konzepte die Implementierung Ihres Domain-Modells als persistente Java-Klassen beeinflussen.

Entities sind die grob gewebten Klassen Ihres Systems. Ihre Instanzen haben unabhängige Lebenszyklen und eine eigene Identität, und man kann über viele andere Instanzen auf sie referenzieren. Wert-Typen hingegen hängen von einer bestimmten Entity-Klasse ab. Eine Instanz eines Wert-Typs hat einen Lebenszyklus, der an seine besitzende Entity-Instanz gebunden ist, und auf sie kann nur von einer Entity referenziert werden – sie hat keine individuelle Entity.

Wir haben uns Identität, Objektgleichheit und Datenbankidentität in Java angeschaut und untersucht, woraus gute Primärschlüssel bestehen. Sie haben gelernt, welche Generatoren für die Werte von Primärschlüsseln bei Hibernate eingebaut sind und wie Sie dieses System der Identifikatoren nutzen und erweitern können.

Sie haben auch verschiedene (hauptsächlich optionale) Optionen für Klassen-Mapping kennengelernt und zum Schluss erfahren, wie grundlegende Eigenschaften und Wert-Typ-Komponenten in XML-Mappings und -Annotationen gemappt werden.

In Tabelle 4.2 finden Sie eine Übersicht der Unterschiede zwischen Hibernate und Java Persistence, bezogen auf die in diesem Kapitel angesprochenen Konzepte.

Tabelle 4.2 Vergleich zwischen Hibernate und JPA für Kapitel 4

Hibernate Core	Java Persistence und EJB 3.0
Entity- und Wert-Typ-Klassen sind die wesentlichen Konzepte für die Unterstützung von reichhaltigen und feingranulierten Domain-Modellen.	Die JPA-Spezifikation macht den gleichen Unterschied, doch nennt Wert-Typen „einbettbare Klassen“. Allerdings werden verschachtelte einbettbare Klassen als ein nicht-portierbares Feature betrachtet.
Hibernate unterstützt im Lieferzustand zehn Strategien zu Identifikator-Generierung.	JPA standardisiert eine Untermenge von vier Identifikator-Generatoren, erlaubt aber Vendor Extensions.
Hibernate kann über Felder, Zugriffsmethoden oder mit jeder maßgeschneiderten `PropertyAccessor`-Implementierung auf Eigenschaften zugreifen. Strategien können für eine bestimmte Klasse gemischt werden.	JPA standardisiert den Zugriff auf Eigenschaften über Felder oder Zugriffsmethoden, und ohne Hibernate Extension Annotations können Strategien für eine bestimmte Klasse nicht gemischt werden.
Hibernate unterstützt Formeleigenschaften und datenbankgenerierte Werte.	JPA bietet keines dieser Features, eine Hibernate-Extension ist erforderlich.

Im nächsten Kapitel betrachten wir die Vererbung und wie Hierarchien für Entity-Klassen mit verschiedenen Strategien gemappt werden können. Wir werden auch über das Mapping-Typen-System von Hibernate sprechen, die Konverter für Wert-Typen, die wir in einigen Beispielen gezeigt haben.