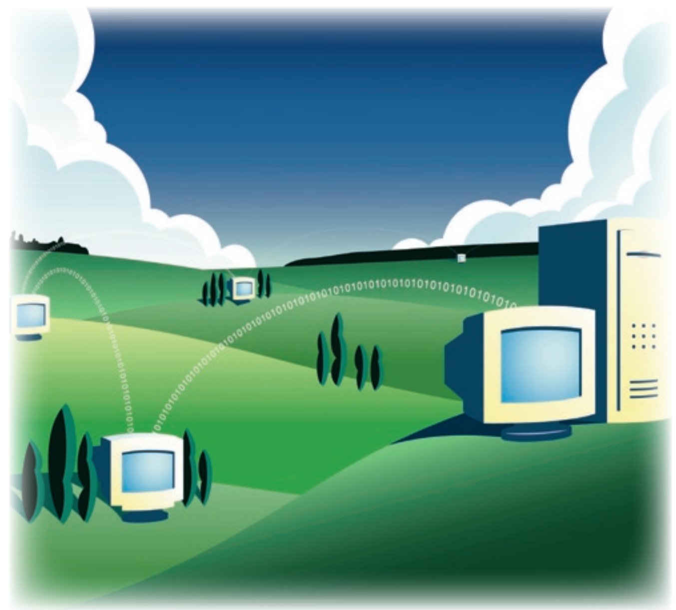
Die Erben-Generation

Persistenz und Vererbung in Hibernate Teil 1: Prinzipien

Dirk Mascher

Mit Hibernate 3 steht eine leistungsfähige, frei verfügbare, quelloffene und in vielen Projekten erfolgreich eingesetzte Technologie zur Abbildung von Objekten und Assoziationen auf Tabellen und Relationen (ORM) zur Verfügung. Die genannten Vorzüge sowie die funktionale Vollständigkeit sind ausschlaggebend für die immer größere Beliebtheit von Hibernate. Zudem wird das Persistenzmodell der EJB-Spezifikation 3.0 in vielen Punkten große Ähnlichkeiten zu Hibernate 3 haben. Gründe genug einige der Fähigkeiten von Hibernate im Rahmen dieses zweiteiligen Artikels tiefer zu beleuchten. Was bietet sich zu diesem Zweck besser an, als die Abbildung von Vererbungshierarchien mittels Hibernate vorzustellen?



Die Abbildung von Vererbungshierarchien in Java-Enterprise-Anwendungen ist (bis zum Zeitpunkt der Entstehung dieses Artikels) immer noch ein Wunsch, der sich mit reinen J2EE-Anwendungen nur schwer bzw. gar nicht realisieren lässt. Hauptursache für dieses Manko sind die beschränkten Möglichkeiten der EJB-2.1-Spezifikation. CMP (container managed persistence) 2.1 bietet keine Möglichkeit, ein objektorientiertes Domänen-Modell direkt auf eine relationale Datenbank abzubilden.

In der Konsequenz sind in den letzten Jahren viele J2EE-Anwendungen entstanden, die zwar in einer objektorientierten Sprache geschrieben wurden, aber im Wesentlichen aus prozeduralem Code bestehen. Dabei ist doch gerade Objektorientierung ein Garant dafür, die Komplexität größerer Anwendungen in den Griff zu bekommen. Eine Möglichkeit, dem Ziel von *Domain-Driven Design* näher zu kommen, besteht darin, in J2EE-Anwendungen auf CMP komplett zu verzichten und die Persistenz mit alternativen Technologien zu realisieren. Als eine dieser alternativen Technologien bietet sich Hibernate an.

Das Herzstück einer Anwendung sollte immer ein sauberes Domänen-Modell sein. Ein wichtiges Mittel, um das Domänen-Modell flexibel und erweiterbar zu halten, stellt Vererbung dar. Mit Hibernate ist es möglich, Klassen des Domänen-Modells direkt auf die eingesetzte relationale Datenbank abzubilden. Man kann also von allen Vorteilen, die Vererbung bietet, uneingeschränkt profitieren. Hinzu kommen in diesem Zusammenhang noch sehr mächtige Features, wie beispielsweise die Möglichkeit polymorphe Abfragen über die *Hibernate Query Language* (HQL) zu formulieren. Dazu später mehr.

Im ersten Teil dieses zweiteiligen Artikels wird zuerst der theoretische Hintergrund für die Abbildung von Vererbungshierarchien auf ein relationales Schema beleuchtet und anschließend anhand eines einfachen Domänen-Modells gezeigt, wie konkrete Abbildungen mit Hibernate aussehen können.

Im zweiten Teil werden die Hibernate-Mappings um eine polymorphe Assoziation ergänzt. Anhand eines Beispiels wird außerdem erläutert, was polymorphe Abfragen sind, wie diese in HQL formuliert und über die Hibernate API abgesetzt werden. In diesem Zusammenhang wird auch auf die Hibernate-Console als nützliches Werkzeug für das Prototyping von HQL-Abfragen eingegangen.

Theoretischer Hintergrund

Martin Fowler beschreibt in [Fow03] drei prinzipielle Vorgehensweisen, eine Klassenhierarchie auf ein relationales Schema abzubilden, und gibt diesen Mustern die folgenden Namen:

- *Single Table Inheritance:* In diesem Fall werden alle Klassen einer Klassenhierarchie, d. h. Basisklassen und konkrete Ableitungen, auf dieselbe Datenbank-Tabelle abgebildet. Diese eine Datenbank-Tabelle enthält also Spalten, die der Vereinigungsmenge aller Felder aller Klassen aus der Hierarchie entsprechen. Zusätzlich wird eine so genannte Diskriminator-Spalte definiert. Diese ist notwendig, um aus Records in der Datenbank wieder Objekte des richtigen Typs instanziieren zu können.
- *Class Table Inheritance:* Hier werden alle Klassen einer Hierarchie auf verschiedene Tabellen abgebildet. Die Felder einer Klasse aus der Hierarchie werden direkt auf Spalten in der zugehörigen Tabelle abgebildet. Die Verlinkung der Klassen kann dadurch realisiert werden, dass in den Records eines abgebildeten Objekts in allen Tabellen derselbe Primärschlüssel verwendet wird. Konkret bedeutet dies, dass es zu einem Eintrag in einer Tabelle, die einer Ableitungsklasse entspricht, genau einen verknüpften Eintrag mit demselben Primärschlüssel in der Tabelle geben muss, die der Basisklasse entspricht.
- *Concrete Table Inheritance:* Bei dieser Strategie werden nur die finalen Ableitungen einer Klassenhierarchie auf jeweils eine Datenbank-Tabelle abgebildet. Jede Tabelle enthält also die Spalten der zugehörigen finalen Ableitungsklasse und alle Spalten, die allen geerbten Feldern entsprechen.

Alle drei vorgestellten Muster für die Abbildung von Vererbungshierarchien haben ihre Vor- und Nachteile. Ein funktional vollständiges ORM-Framework sollte idealerweise die Verwendung aller drei Strategien erlauben, sodass man die für einen speziellen Anwendungsfall günstigste Strategie auswählen kann. Hibernate implementiert alle vorgestellten Strategien. Die Strategie „Concrete Table Inheritance“ wird allerdings erst ab Version 3 richtig unterstützt.

Domänen-Modell Online-Shop

Um das Ganze etwas anschaulicher zu gestalten, soll im Folgenden ein Ausschnitt aus einem stark vereinfachten Domänen-

Modell (s. Abb. 1) eines fiktiven Online-Shops mittels Hibernate auf eine relationale Datenbank abgebildet werden. In unserem Online-Shop gibt es registrierte Kunden. Jeder Kunde kann mehrere Zahlungsinformationen hinterlegen und bei Bezahlvorgängen eine davon auswählen. Eine Zahlungsinformation ist dabei erst einmal etwas Abstraktes. Konkrete Ausprägungen können z. B. Kreditkarten oder Bankverbindungen sein.

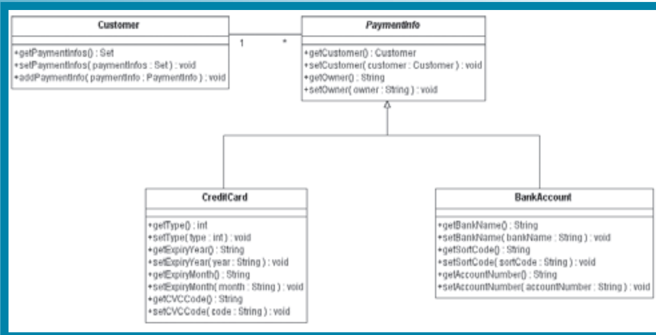


Abb. 1: Domänen-Modell zum Beispiel Online-Shop

Hibernate-Strategien

Wir konzentrieren uns zunächst auf die `PaymentInfo`-Klassenhierarchie unseres Online-Shops und wollen diese mit den drei vorgestellten Strategien auf ein relationales Modell abbilden. Auf die Abbildung der `Customer`-Klasse und der „1"-Seite der bidirektionalen Assoziation zwischen `Customer` und `PaymentInfo` wird im zweiten Teil dieses Artikels eingegangen.

Hibernate-Strategie: Table per Class Hierarchy

Beginnen wir mit der Strategie „Single Table Inheritance" bzw. „Table per Class Hierarchy", wie die Strategie in der Hibernate-Referenz-Dokumentation auch genannt wird [HIB]. Bevor wir auf das eigentliche Mapping eingehen, werfen wir zuerst einen Blick auf das resultierende Schema (s. Abb. 2).

Die komplette `PaymentInfo`-Vererbungshierarchie wird über eine einzige Tabelle `PAYMENT_INFO` abgebildet. Die spezielle Diskriminator-Spalte `PAYMENT_INFO_TYPE` verwendet Hibernate intern, um bei Abfragen Objekte des richtigen Typs, d. h. der richtigen Ableitungsklasse, erzeugen zu können. Das für diese Abbildung notwendige Mapping-Dokument ist in Listing 1 dargestellt.

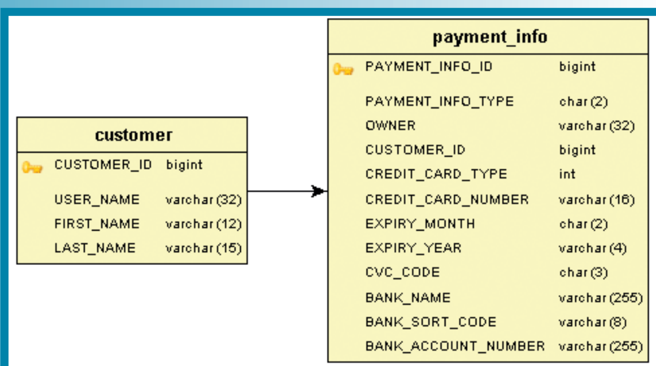


Abb. 2: Datenbank-Schema für die Strategie „Single Table Inheritance" bzw. „Table per Class Hierarchy"

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping
        PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
         "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
  <class name="samples.hibernate.inheritance.model.PaymentInfo"
         table="PAYMENT_INFO"
         abstract="true">
    <id name="id" type="long" column="PAYMENT_INFO_ID">
      <generator class="native" />
    </id>
    <discriminator column="PAYMENT_INFO_TYPE"
                   type="string" length="2"/>
    <property name="owner" column="OWNER"
              type="string" not-null="true"/>
    <many-to-one name="customer"
                 class="samples.hibernate.inheritance.model.Customer"
                 column="CUSTOMER_ID" />
      <subclass name="samples.hibernate.inheritance.model.CreditCard"
                discriminator-value="CC">
        <property name="type" column="CREDIT_CARD_TYPE" type="int" />
        <property name="number" column="CREDIT_CARD_NUMBER"
                  type="string" />
        <property name="expMonth" column="EXPIRY_MONTH"
                  type="string" />
        <property name="expYear" column="EXPIRY_YEAR" type="string" />
        <property name="cvcCode" column="CVC_CODE" type="string" />
      </subclass>
      <subclass name="samples.hibernate.inheritance.model.BankAccount"
                discriminator-value="BA">
        <property name="bankName" column="BANK_NAME" type="string" />
        <property name="sortCode" column="BANK_SORT_CODE" type="string" />
        <property name="accountNumber" column="BANK_ACCOUNT_NUMBER"
                  type="string" />
      </subclass>
  </class>
</hibernate-mapping>
```

Listing 1: PaymentInfo.hbm.xml für Hibernates „Table per Class Hierarchy"-Strategie

Über das `discriminator`-Tag wird die Diskriminator-Spalte definiert. Die Definition der Ableitungen erfolgt über `<subclass>`-Tags. Ein `<subclass>`-Tag muss dabei über das Attribut `discriminator-value` einen Wert für die Diskriminator-Spalte definieren. In Listing 1 wird beispielsweise der Wert „CC" für die Subklasse `CreditCard` definiert. Dies bedeutet, dass alle von Hibernate in der Tabelle `PAYMENT_INFO` persistierten `CreditCard`-Objekte in der Spalte `PAYMENT_INFO_TYPE` mit dem Wert „CC" markiert werden.

Ein großer Vorteil dieser Strategie liegt in der sehr guten Performance, da für alle `PaymentInfo`-Abfragen lediglich auf eine Tabelle zugegriffen werden muss. Ein offensichtlicher Nachteil der Strategie liegt darin, dass für die Spalten, die den Ableitungsklassen entsprechen, keine *not-null Constraints* definiert werden können – auch wenn dies beispielsweise für das Feld Kreditkartennummer eigentlich wünschens-

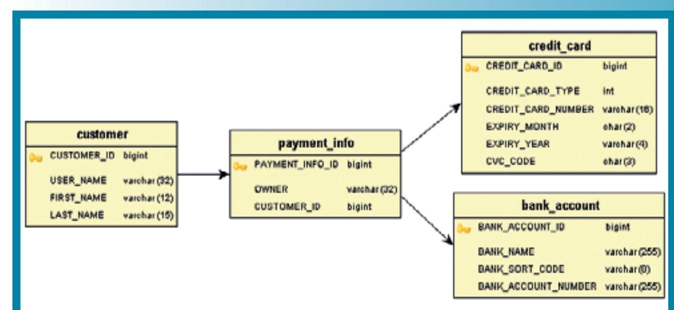


Abb. 3: Datenbank-Schema für die Strategie „Class Table Inheritance" bzw. „Table per Subclass"

wert wäre. *Not-null Constraints* können hier ausschließlich für Spalten, die der Basisklasse entsprechen, definiert werden. In der Praxis ist dies jedoch ein geringer Preis, den man gerne gewillt ist zu bezahlen, um von der guten Performance dieser Strategie zu profitieren.

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping
        PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
     "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
  <class name="samples.hibernate.inheritance.model.PaymentInfo"
         table="PAYMENT_INFO"
         abstract="true">
    <id name="id" type="long" column="PAYMENT_INFO_ID">
      <generator class="native" />
    </id>
    <property name="owner" column="OWNER"
              type="string" not-null="true" />
    <many-to-one name="customer"
                 class="samples.hibernate.inheritance.model.Customer"
                 column="CUSTOMER_ID" />
    <joined-subclass
        name="samples.hibernate.inheritance.model.CreditCard"
        table="CREDIT_CARD">
      <key column="CREDIT_CARD_ID" />
      <property name="type" column="CREDIT_CARD_TYPE" type="int" />
      <property name="number" column="CREDIT_CARD_NUMBER"
                type="string" not-null="true"/>
      <property name="expMonth" column="EXPIRY_MONTH" type="string" />
      <property name="expYear" column="EXPIRY_YEAR" type="string" />
      <property name="cvcCode" column="CVC_CODE" type="string" />
    </joined-subclass>
    <joined-subclass
        name="samples.hibernate.inheritance.model.BankAccount"
        table="BANK_ACCOUNT">
      <key column="BANK_ACCOUNT_ID" />
      <property name="bankName" column="BANK_NAME" type="string" />
      <property name="sortCode" column="BANK_SORT_CODE"
                type="string" />
      <property name="accountNumber" column="BANK_ACCOUNT_NUMBER"
                type="string" />
    </joined-subclass>
  </class>
</hibernate-mapping>
```

Listing 2: PaymentInfo.hbm.xml für Hibernates „Table per Subclass"-Strategie

Hibernate-Strategie: Table per Subclass

Die von Martin Fowler mit „Class Table Inheritance" benannte Strategie wird in der Hibernate-Dokumentation als „Table per Subclass" bezeichnet. Das resultierende Schema dieser Strategie in unserem Beispiel ist in Abbildung 3 dargestellt. Das zugehörige Mapping-Dokument zur Abbildung der `PaymentInfo`-Hierarchie in diesem Fall zeigt Listing 2.

Ableitungen werden bei dieser Variante in der Mapping-Datei mit dem `joined-subclass`-Tag definiert. Als Attribut muss dabei der Name der zugehörigen Tabelle angegeben werden. Alle gemeinsamen Properties werden über die Tabelle `PAYMENT_INFO` abgebildet. Um ein konkretes `CreditCard`-Objekt aus der Datenbank heraus zu instanziieren, muss Hibernate in diesem Beispiel also Daten aus den Tabellen `PAYMENT_INFO` und `CREDIT_CARD` lesen. Um die jeweils zugehörigen Daten zu finden, ist eine Verknüpfung der Tabellen notwendig. Diese wird über die Primärschlüssel realisiert. Zu jedem Eintrag in der Tabelle `CREDIT_CARD` gibt es genau einen passenden Eintrag in der Tabelle `PAYMENT_INFO`.

Obwohl das relationale Schema in diesem Fall keine Redundanzen und Anomalien aufweist und obwohl in diesem Fall auch für die Spalten der Subklassen *not-null Constraints* definierbar sind (siehe Beispiel Kreditkartennummer `CREDIT_CARD_NUMBER` in Listing 2), ist diese Variante nur für kleinere Klassenhierarchien (Tiefe 2) empfehlenswert, da intern sonst zu viele Join-Operationen – selbst bei eigentlich einfachen Abfragen – generiert werden müssten.

Hibernate-Stategie: Table per Concrete Class

Die Strategie „Concrete Table Inheritance", die in der Version 3.0 von Hibernate neu hinzugekommen ist, wird in der Referenz-Dokumentation mit „Table per Concrete Class" bezeichnet. Das zugehörige Schema zur Abbildung des Domänen-Modells unseres fiktiven Online-Shops zeigt Abbildung 4.

Alle Spalten, die den Properties der `PaymentInfo`-Basisklasse entsprechen, werden jeweils über die Tabellen `CREDIT_CARD` und `BANK_ACCOUNT` abgebildet. Das für diese Abbildung notwendige Mapping-Dokument zeigt Listing 3.

Die Abbildung der Subklassen erfolgt hier über `union-subclass`-Tags. Als Attribut muss für das `union-subclass`-Tag immer der Name der Tabelle angegeben werden, auf die die konkrete Ableitungsklasse abgebildet werden soll. Interessant ist in diesem Fall, dass das übergeordnete `class`-Tag für die Definition der abstrakten Basisklasse keinen Wert für das `table`-Attribut definiert. Das `id`-Tag direkt unterhalb des `class`-Tags für die Basisklasse beschreibt die Regeln für die Abbildung bzw. Erzeugung von Primärschlüsseln aller Tabellen, die über die `union-sublcass`-Tags definiert sind.

Ein Vorteil dieses Mappings ist, dass Abfragen auf konkrete Objekte sehr effizient möglich sind, da hier (wie bei der „Table per Class Hierarchy"-Strategie) nur ein SQL-Statement auf eine Tabelle notwendig ist. Anders sieht es jedoch aus, wenn polymorphe Abfragen über die *Hibernate Query Language* HQL in SQL-Select-Statements transformiert werden müssen. In diesem Fall muss Hibernate SQL-UNION-Statements generieren. Ein weiterer Nachteil dieser Mapping-Strategie liegt in der Redundanz im relationalen Modell. Bei einer Erweiterung der Properties der Basisklasse müssen in der Datenbank alle Tabellen, die den Subklassen entsprechen, angepasst werden.

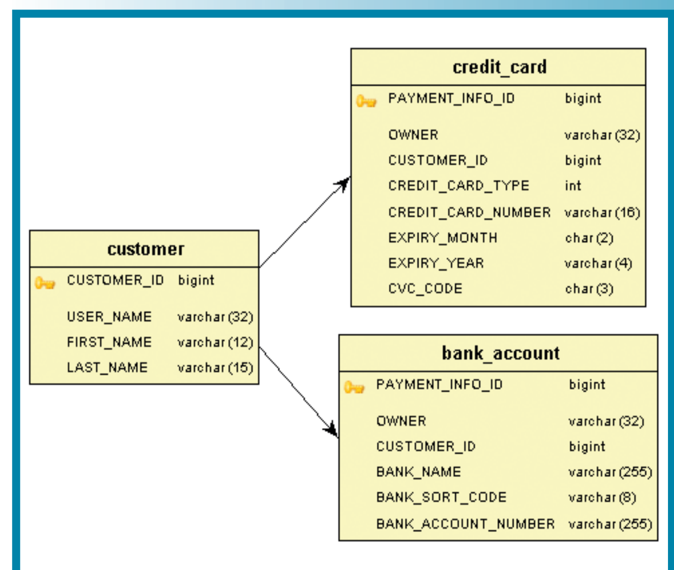


Abb. 4: Datenbank-Schema für die Strategie „Concrete Table Inheritance" bzw. „Table per Concrete Class"

```xml
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping
          PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
     "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
  <class name="samples.hibernate.inheritance.model.PaymentInfo"
         abstract="true">
    <id name="id" type="long" column="PAYMENT_INFO_ID">
      <generator class="hilo" />
    </id>
    <property name="owner" column="OWNER"
              type="string" not-null="true" />
    <many-to-one name="customer"
                 class="samples.hibernate.inheritance.model.Customer"
                 column="CUSTOMER_ID" />
    <union-subclass
        name="samples.hibernate.inheritance.model.CreditCard"
        table="CREDIT_CARD">
      <property name="type" column="CREDIT_CARD_TYPE" type="int" />
      <property name="number" column="CREDIT_CARD_NUMBER"
                type="string"/>
      <property name="expMonth" column="EXPIRY_MONTH" type="string" />
      <property name="expYear" column="EXPIRY_YEAR" type="string" />
      <property name="cvcCode" column="CVC_CODE" type="string" />
    </union-subclass>
    <union-subclass
        name="samples.hibernate.inheritance.model.BankAccount"
        table="BANK_ACCOUNT">
      <property name="bankName" column="BANK_NAME" type="string"/>
      <property name="sortCode" column="BANK_SORT_CODE"
                type="string" />
      <property name="accountNumber" column="BANK_ACCOUNT_NUMBER"
                type="string"/>
    </union-subclass>
  </class>
</hibernate-mapping>
```

Listing 3: PaymentInfo.hbm.xml für Hibernates „Table per Concrete Class"-Strategie

Zusammenfassung und Ausblick

Im diesem ersten Teil wurden die prinzipiellen Möglichkeiten aufgezeigt, wie sich Vererbungshierarchien auf ein relationales Schema abbilden lassen. Martin Fowler hat diese in Form von Mustern in [Fow03] zusammengefasst. Anhand eines einfachen Domänen-Modells eines beispielhaften Online-Shops wurden diese Muster mit der jeweiligen in Hibernate 3 vorgesehenen Strategie umgesetzt.

Im abschließenden zweiten Teil des Artikels werden die Hibernate-Mappings um das Mapping für die `Customer`-Klasse erweitert. Das Interessante an diesem Mapping wird die polymorphe Assoziation zur abstrakten `PaymentInfo`-Klasse sein. Neben polymorphen Assoziationen wird im zweiten Teil außerdem auf polymorphe Abfragen und implizite Joins eingegangen. Des Weiteren wird die Hibernate-Console als nützliches Tool für das Prototyping von HQL-Abfragen vorgestellt.

Literatur und Links

[Bet05] U. Bettag, Persistenz mit Hibernate, in: JavaSPEKTRUM, Sonderheft zur CeBit, 2005

[Fow03] M. Fowler, Patterns of Enterprise Application Architecture, Addison-Wesley, 2003, **http://www.martinfowler.com/eaaCatalog/**

[HIB] Hibernate, **http://www.hibernate.org/**

[JSR220] Java Specification Request 220: Enterprise JavaBeans 3.0, **http://www.jcp.org/en/jsr/detail?id=220**

Dirk Mascher ist selbständiger Berater im Java- und J2EE-Umfeld. Er hat in den letzten Jahren viele größere J2EE-Projekte zum Erfolg geführt. Ein Schlüssel dieses Erfolgs lag u. a. darin, zu einzelnen, problematischen J2EE-Technologien Alternativen zu finden. Eine dieser Alternativen ist Hibernate, das Dirk Mascher bereits in mehreren Projekten erfolgreich einsetzen konnte. Neben seiner Projekttätigkeit hat Dirk Mascher in den letzten Jahren mehrere Seminare entwickelt, u. a. auch zu dem Thema Hibernate. Dirk Mascher ist zertifizierter JBoss Consultant.
E-Mail: dirk.mascher@gmx.de.