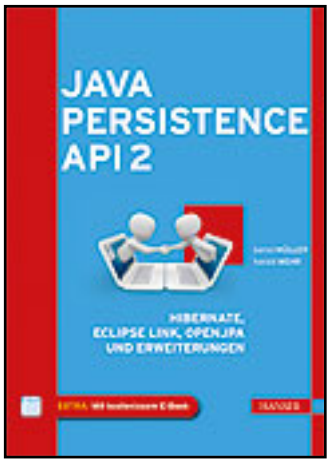
HANSER



Leseprobe

Bernd Müller, Harald Wehr

Java Persistence API 2

Hibernate, EclipseLink, OpenJPA und Erweiterungen

ISBN: 978-3-446-42693-1

Weitere Informationen oder Bestellungen unter

http://www.hanser.de/978-3-446-42693-1

sowie im Buchhandel.

© Carl Hanser Verlag, München

5 Vererbung

In Kapitel 4 haben wir uns mit Beziehungen zwischen Objekten beschäftigt und als Beispiel unter anderem Kunden mit ihren Konten und Buchungen auf diesen Konten betrachtet. Konten existieren aber in verschiedenen Ausprägungen: Sparkonten, Girokonten, Festgeldkonten und andere. In objektorientierten Sprachen werden derartige Ausprägungen in der Regel mit Hilfe von Vererbung modelliert. Dieses Kapitel beschreibt die verschiedenen Möglichkeiten, die JPA zur Realisierung von Vererbungsbeziehungen bereitstellt, und schließt damit die Darstellung des Mappings objektorientierter Konzepte mit JPA ab.

Wie bereits in Kapitel 1 motiviert, ist die Abbildung einer objektorientierten Vererbungshierarchie in relationale Datenbanken eines der zentralen Probleme von OR-Mappern. Heute besteht Konsens darüber, dass prinzipiell drei alternative Realisierungsmöglichkeiten existieren, die Gegenstand dieses Kapitels sind:

- eine einzige Tabelle für eine gesamte Vererbungshierarchie
- eine Tabelle je Unterklasse
- eine Tabelle je konkreter Klasse

Das oben angesprochene Beispiel verschiedener Kontoarten soll nun konkret werden. Wir verwenden eine abstrakte Oberklasse `Konto` sowie die beiden Unterklassen `Sparkonto` und `Girokonto`. Die in Kapitel 4 verwendeten Beziehungen zu Kunden und Buchungen sollen ebenfalls realisiert werden. Die Abbildung 5.1 zeigt das entsprechende UML-Klassenmodell.

5.1 Eine Tabelle für eine Vererbungshierarchie

Wir beginnen mit der einfachsten Alternative, die alle Klassen einer Vererbungshierarchie in eine einzige Tabelle integriert. Abbildung 5.2 zeigt schematisch die entstehende Struktur inklusive zweier Datensätze, wenn die Klassen `Konto`, `Sparkonto` und `Girokonto` in einer Tabelle zusammengefasst werden. Man erkennt von links nach rechts zuerst die sogenannte *Diskriminatorspalte* `disc`. Der Wert dieser Spalte wird verwendet, um den Entity-Typ der Daten dieser Tabellenzeile zu bestimmen. In unserem Beispiel ist die Klasse `Konto` abstrakt, so dass nur zwischen `Sparkonto` und `Girokonto` unterschieden werden muss. Dies kann

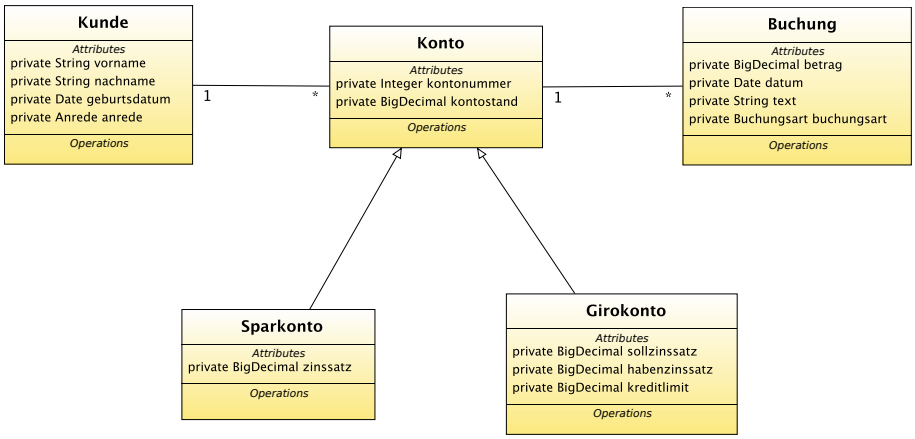


ABBILDUNG 5.1 Vererbungshierarchie der Konten

z. B. durch einen Integer-Wert oder, wie im Beispiel, durch die Strings `"Spar"` und `"Giro"` geschehen. Die nächsten Spalten, `kontonummer` und `kontostand`, sind Properties der Klasse `Konto` und damit in jeder Instanz von `Sparkonto` und `Girokonto` enthalten. Es folgt die Spalte `zinssatz`, das einzige Property der Klasse `Sparkonto`. Die letzten drei Spalten repräsentieren die Properties der Klasse `Girokonto`. In der Darstellung haben wir die Spalte `kunde` unterschlagen, da sie nichts mit Vererbung zu tun hat. Sie fungiert als Fremdschlüssel in die Kundentabelle.

disc	konto-nummer	konto-stand	zinssatz	soll-zinssatz	haben-zinssatz	kreditlimit
			...			
Spar	100000	100,00	0,45			
			...			
Giro	100001	2500,00		14,50	0,20	4000
			...			
	Konto		Sparkonto	Girokonto		

ABBILDUNG 5.2 Tabellenstruktur für die Vererbungshierarchie

Bei den beiden Beispieldatensätzen kann man erkennen, dass die von einer Oberklasse geerbten Properties auch in den Spalten der Unterklassen vorhanden sind, während die Properties der nicht abstrakten Klassen immer nur in einem Datensatz vorhanden und die Spalten der jeweils anderen Unterklasse(n) NULL sind.

Die Umsetzung der Alternative *eine Tabelle für eine Vererbungshierarchie* erfolgt in JPA mit der `@Inheritance`-Annotation und Setzen des Attributs `strategy` auf `SINGLE_TABLE`. Dies ist gleichzeitig der Default-Wert dieses Attributs. Tabelle 5.1 gibt die `@Inheritance`-Annotation komplett wieder.

TABELLE 5.1 `@Inheritance`-Attribute

`@Inheritance(...)`			
Attribut	**Typ**	**Default**	**Beschreibung**
`strategy`	`Inheritance-Type`	`SINGLE_TABLE`	Strategie zur Realisierung der Vererbung. Mögliche Werte: `SINGLE_TABLE`, `JOINED`, `TABLE_PER_SUBCLASS`

Die in der Tabellenstruktur in Abbildung 5.2 dargestellte zusätzliche Diskriminatorspalte wird mit der `@DiscriminatorColumn`-Annotation definiert. Tabelle 5.2 zeigt die Attribute dieser Annotation.

TABELLE 5.2 `@DiscriminatorColumn`-Attribute

`@DiscriminatorColumn(...)`			
Attribut	**Typ**	**Default**	**Beschreibung**
`column-Definition`	`STRING`	—	DDL für Diskriminator-Spalte
`discriminator-Type`	`Discriminator-Type`	`STRING`	Typ des Diskriminators. Mögliche Werte sind `CHAR`, `INTEGER`, `STRING`
`length`	`int`	31	Länge von String-Diskriminatoren
`name`	`String`	`DTYPE`	Spaltenname des Diskriminators

Tipp:

Die Verwendung eines Strings als Diskriminatorwert erleichtert das Verständnis und die Nutzung der Tabelle auf SQL-Ebene.

Das Listing 5.1 gibt die ersten Zeilen der Oberklasse `Konto` wieder. Die Definition der Diskriminatorspalte entspricht dem Beispiel aus Abbildung 5.2. Das Listing ist recht umfangreich, was jedoch nicht der Vererbung geschuldet ist. Beim Generieren des Primärschlüssels wird bei 100 000 begonnen, um möglichst realistische Kontonummern zu erhalten. Die Verwendung von Sequenzen zur Primärschlüsselgenerierung haben wir in Abschnitt 2.1.1 erläutert. Die Beziehungsannotationen werden verwendet, um die 1:n- bzw. die n:1-Beziehung zu Buchung und Kunde zu realisieren.

LISTING 5.1 Klasse `Konto` mit Vererbungsannotationen

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "Disc",
                     discriminatorType = DiscriminatorType.STRING)
public abstract class Konto {

  @Id @GeneratedValue(generator = "KtoSeq",
                      strategy = GenerationType.SEQUENCE)
  @SequenceGenerator(name = "KtoSeq", sequenceName = "KTO_SEQ",
                     allocationSize = 100, initialValue = 100000)
```

```
private Integer kontonummer;
@Column(precision = 10, scale = 2)
private BigDecimal kontostand;

@ManyToOne(optional = false)
@JoinColumn(name = "kunde", nullable = false)
private Kunde kunde;

@OneToMany(mappedBy = "konto", cascade = CascadeType.ALL)
private Set<Buchung> buchungen;
...
```

In den Unterklassen ist lediglich noch der Wert des Diskriminators für diese Klasse anzugeben. Dies erfolgt über die `@DiscriminatorValue`-Annotation, die in Tabelle 5.3 dargestellt ist. Falls diese Annotation nicht verwendet wird, generiert die JPA-Implementierung einen in der Spezifikation nicht näher beschriebenen, herstellerspezifischen Wert. Im Falle eines String-Diskriminators wird der Entity-Name verwendet. Falls Sie die `@DiscriminatorValue`-Annotation verwenden, muss ihr String-Wert dem Diskriminatortyp entsprechen, also z. B. bei `INTEGER` ein ganzzahliger Wert sein.

TABELLE 5.3 `@DiscriminatorValue`-Optionen

`@DiscriminatorValue(...)`			
Option	**Typ**	**Default**	**Beschreibung**
`value`	`String`	—	Wert zur Repräsentation des Entity-Typs. Beim Diskriminatortyp `String` ist der Default der Entity-Name, sonst ein provider-abhängiger Wert.

Im folgenden Code-Ausschnitt entscheiden wir uns bei der Klasse `Sparkonto` für den String `"Spar"`, der auch im Beispiel verwendet wurde. Weil das `Id`-Property bereits in der Oberklasse definiert wurde, wird es in der Unterklasse nicht angegeben. Wir verzichten auf eine Darstellung der Girokonto-Klasse.

```
@Entity
@DiscriminatorValue("Spar")
public class Sparkonto extends Konto {

  @Column(precision = 5, scale = 2)
  private BigDecimal zinssatz;

  ...
```

Das Schema für die drei Tabellen ist relativ offensichtlich. Wir stellen es in Abbildung 5.3 trotzdem dar, damit wir die noch vorzustellenden Alternativen zur Realisierung der Vererbung im Vergleich besser beurteilen können.

Man erkennt in der Tabelle `konto` die Spalten, die wir bereits in Bezug zur Tabelle 5.2 diskutiert haben, sowie die in Tabelle 5.2 unterschlagene Spalte `kunde`, die den Fremdschlüssel zur Kundentabelle repräsentiert.

Das herunterladbare Projekt enthält Datensätze, die in Abbildung 5.4 in der H2-Konsole nach der Abfrage „`Select * from konto`“ dargestellt sind. Die verwendete Vererbungsalternative *eine Tabelle für eine Hierarchie* unterstützt polymorphe Assoziationen vollständig, so

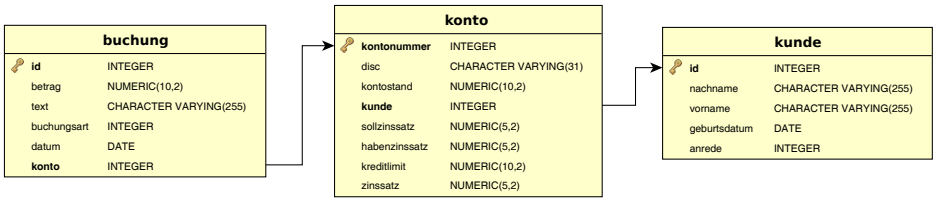


ABBILDUNG 5.3 Tabellenschema für die Vererbungsstrategie `SINGLE_TABLE`

dass ein Kunde mit dem Getter `getKonten()` nach *allen* Konten, sowohl Giro- als auch Sparkonten, gefragt werden kann.

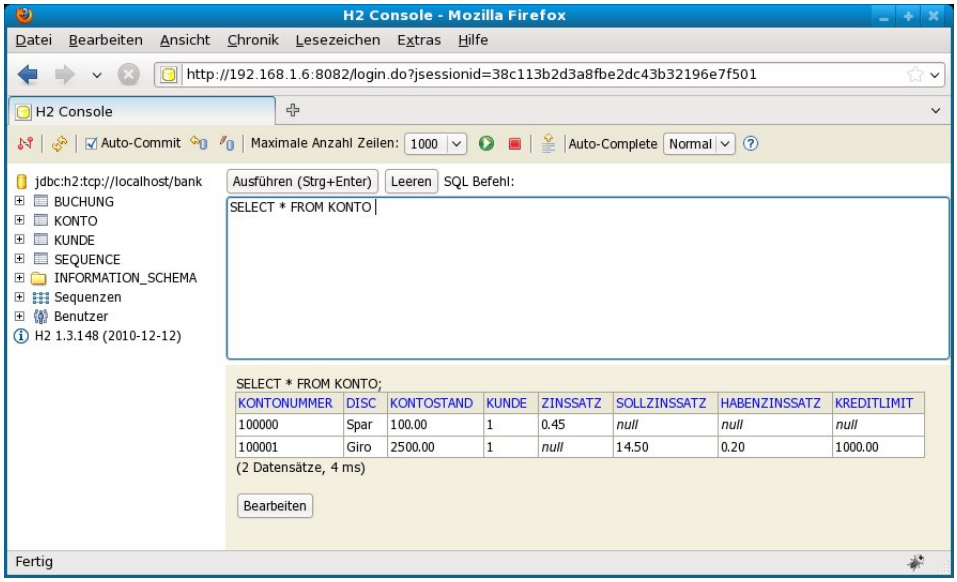


ABBILDUNG 5.4 Beispieldatensätze in der Datenbank

Der folgende Code-Ausschnitt zeigt dies exemplarisch, indem über das Ergebnis dieses Getter-Aufrufs iteriert wird.

```
Kunde k = em.find(Kunde.class, ...);
System.out.println("gelesener Kunde: " + k.getNachname());
for (Konto konto : k.getKonten()) {
  System.out.println("Konto-ID: " + konto.getKontonummer()
                     + ", Klasse: " + konto.getClass().getSimpleName
                         ());
}
```

Die Ausgabe des dargestellten Code-Ausschnitts lautet

```
gelesener Kunde: Mustermann
Konto-ID: 100001, Klasse: Girokonto
Konto-ID: 100000, Klasse: Sparkonto
```

Auch die in Kapitel 7 noch vorzustellenden Datenbankabfragen mit JPQL arbeiten polymorph. Die Abfrage „`Select k from Konto k`“ selektiert alle Instanzen der Entity-Klasse `Konto`. Syntaktisch wird dies im Vorgriff auf Kapitel 7 wie folgt dargestellt.

```
List<Konto> konten = em.createQuery("Select k from Konto k",
                                    Konto.class)
                       .getResultList();
```

Die abstrakte Klasse `Konto` besitzt keine Instanzen. In unserem Beispiel existiert jedoch je eine Instanz der nicht abstrakten Unterklassen. Der folgende Code zeigt die Iteration über das Abfrageergebnis sowie die textuelle Ausgabe.

```
for (Konto konto : konten) {
  System.out.println(konto.getClass().getSimpleName()
     + " mit Kontonummer " + konto.getKontonummer());
}
```

```
Sparkonto mit Kontonummer 100000
Girokonto mit Kontonummer 100001
```

Hinweis:

Soll das Property `habenzinssatz` der Klasse `Girokonto` in `zinssatz` umbenannt werden, so werden die identisch benannten Properties der beiden Unterklassen auf dieselbe Tabellenspalte abgebildet. Ist weiterhin die Existenz zweier Spalten gewünscht, so kann mit der `@Column`-Annotation ein alternativer Spaltenname definiert werden.

Das Paradigma der *Convention over Configuration* wird von JPA vollständig realisiert, wie viele Beispiele des Kapitels 2 exemplarisch gezeigt haben. Im Falle der Vererbung ist die Strategie `SINGLE_TABLE` der Default. Für die Diskriminator-Spalte sowie deren Werte existieren ebenfalls Defaults: Der Spaltenname ist `DTYPE`, der Typ `String` und der Wert der Entity-Name. Es können also alle vererbungsrelevanten JPA-Annotationen des Beispiels entfallen, ohne die Korrektheit und Vollständigkeit des Beispiels zu gefährden. Die folgenden Klassendefinitionen entsprechen dem obigen Beispiel, wobei das Mapping implizit von einer String-Diskriminatorspalte `DTYPE` und den Werten `Sparkonto` und `Girokonto` als einzigem Unterschied ausgeht.

```
@Entity
public abstract class Konto {
  ....
}

@Entity
public class Sparkonto extends Konto {
  ...
}

@Entity
public class Girokonto extends Konto {
  ...
}
```

Achtung:

Die in diesem und den nächsten Abschnitten vorgestellten Möglichkeiten zur Realisierung von Vererbungsbeziehungen werden von OpenJPA nur implementiert, wenn die kompilierten Klassen einem sogenannten Enhancement unterzogen werden. Wir gehen hierauf in Abschnitt 13.2 ein.

Projekt:

Das Projekt *vererbung-single-table* enthält den Code für die erste Vererbungsalternative *eine Tabelle für eine Vererbungshierarchie*, die durch die Vererbungsstrategie `SINGLE_TABLE` realisiert wird.

5.2 Eine Tabelle je Unterklasse

Das zugrunde liegende Problem bei der Abbildung objektorientierter Vererbung auf relationale Datenbanken ist der Umstand, dass objektorientierte Sprachen sowohl die Vererbungsbeziehung als auch eine „normale“ Beziehung, in der objektorientierten Welt als Assoziation bezeichnet, kennen. Man spricht auch von „is-a“- und „has-a“-Beziehungen. Relationale Datenbanken kennen aber lediglich die zweite Beziehungsart. Die Realisierungsalternative *eine Tabelle je Unterklasse* macht sich dies zunutze und bildet die Vererbung auf eine Beziehung zwischen Tabellen ab. Um auf die Daten eines Unterklasseobjekts zugreifen zu können, muss die JPA-Implementierung daher über diese Tabellen joinen.

Wie sieht nun die konkrete Realisierung dieser Alternative aus? Der folgende Code-Ausschnitt gibt die Annotationen der Oberklasse wieder. Als Vererbungsstrategie wird `JOINED`, als Diskriminatorspalte `Disc` verwendet.

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
@DiscriminatorColumn(name = "Disc")
public abstract class Konto {

  @Id @GeneratedValue
  private Integer kontonummer;
  ...
```

Als Diskriminatorwert für die Unterklassen bleiben wir bei den Werten `Spar` und `Giro` des Beispiels. Die Unterklassen benötigen kein mit `@Id` annotiertes Primärschlüssel-Property.

```
@Entity
@DiscriminatorValue("Spar")
public class Sparkonto
       extends Konto {
  ...
}
```

```
@Entity
@DiscriminatorValue("Giro")
public class Girokonto
       extends Konto {
  ...
}
```

Das entsprechende Tabellenschema für die beschriebene Struktur ist in Abbildung 5.5 dargestellt. Für die Vererbungshierarchie existieren die drei Tabellen `Konto`, `Sparkonto` und `Girokonto`, deren Spalten den jeweiligen Properties der Klassen entsprechen. Die Klasse

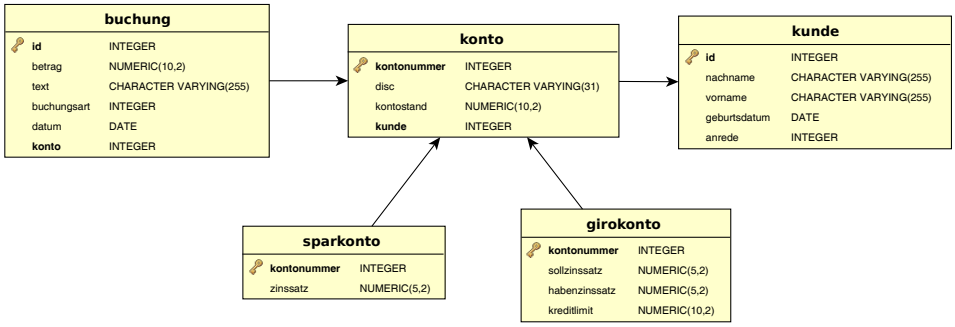


ABBILDUNG 5.5 Tabellenschema für die Vererbungsstrategie `JOINED`

`Konto` enthält das Primärschlüssel-Property `kontonummer`, das sich auch in der Tabelle wiederfindet. Die Klassen `Sparkonto` und `Girokonto` besitzen ein derartiges Property nicht, sehr wohl aber die entsprechenden Tabellen die jeweiligen Spalten. Diese werden für den benötigten Join als Fremdschlüssel verwendet.

Falls die Tabellen schon existieren oder mit abweichenden Bezeichnern generiert werden sollen, kann mit der `@PrimaryKeyJoinColumn`-Annotation ein abweichender Name in der Unterklasse angegeben werden. Dieser wird dann für den Join mit der Tabelle der Oberklasse verwendet. Soll etwa der Fremdschlüssel in der Tabelle `Sparkonto` `Konto_kontonummer` heißen, so lässt sich dies folgendermaßen realisieren:

```
@Entity
@DiscriminatorValue("Spar")
@PrimaryKeyJoinColumn(name = "Konto_kontonummer")
public class Sparkonto extends Konto {
  ...
}
```

Bei der Verwendung der Klassen ergibt sich keine Änderung. Auch diese Vererbungsalternative realisiert polymorphe Abfragen vollständig. Das Beispiel der Abfrage aller Konten eines Kunden sowie die JPQL-Abfrage aller existierender Konten aus Abschnitt 5.1 können problemlos ausgeführt werden und ergeben dieselben Ergebnisse.

Achtung:

Die Vererbungsalternative *eine Tabelle je Unterklasse* verwendet Joins, um die Daten von Unterklasseninstanzen zusammenzustellen. Bei tiefen Vererbungshierarchien kann dies zu Performanzproblemen führen. Die Spezifikation warnt ausdrücklich: „In deep class hierarchies, this may lead to unacceptable performance.“

Die Notwendigkeit zur Verwendung von `@DiscriminatorColumn` in der Oberklasse und `@DiscriminatorValue` in der Unterklasse sowie die Default-Regeln bei Nichtexistenz sind in der Spezifikation (Abschnitt 11.1.10) leider nicht exakt formuliert. EclipseLink als Referenzimplementierung realisiert bei der Nichtexistenz die Default-Regeln, die auch für die Alternative `SINGLE_TABLE` gelten, also `DTYPE` als Spaltenname, Klassennamen als Werte. Hibernate und OpenJPA verzichten vollständig auf die Diskriminatorspalte.

Achtung:

Falls die Annotationen `@DiscriminatorColumn` und `@DiscriminatorValue` nicht verwendet werden, bilden Hibernate und OpenJPA die Oberklasse ohne Diskriminatorspalte auf die entsprechende Tabelle ab, während EclipseLink die Default-Regeln für die beiden Annotationen anwendet. Dies führt zu nicht portablen Anwendungen.

Projekt:

Das Projekt *vererbung-joined* enthält den Code für die zweite Vererbungsalternative *eine Tabelle je Unterklasse*, die durch die Vererbungsstrategie `JOINED` realisiert wird.

5.3 Eine Tabelle je konkreter Klasse

Bei der Alternative *eine Tabelle je konkreter Klasse* wird eine Klassenhierarchie auf eine Tabellenstruktur abgebildet, bei der eine konkrete, nicht abstrakte Klasse durch eine Tabelle repräsentiert wird, die die Properties der Klasse selbst, aber auch die geerbten Properties aller Oberklassen enthält. Abstrakte Klassen werden im Gegensatz zur Alternative aus Abschnitt 5.2 nicht durch Tabellen repräsentiert, sondern sind implizit in den Tabellen der konkreten Unterklassen vorhanden. Der JPA-Provider wird daher mit dem SQL-Vereinigungsoperator `Union` oder getrennten Abfragen je Unterklasse arbeiten, um die Daten für polymorphe Operationen zu ermitteln. Die Umsetzung erfolgt durch die Vererbungsstrategie `TABLE_PER_CLASS`, so dass sich als Code für die Oberklasse `Konto` der folgende Ausschnitt ergibt

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public abstract class Konto {

  @Id @GeneratedValue(strategy = GenerationType.SEQUENCE)
  private Integer kontonummer;
  ...
```

In den konkreten Unterklassen wird keine spezielle Syntax benötigt, so dass wir auf eine Darstellung verzichten.

Hinweis:

Die beschriebene Alternative *eine Tabelle je konkreter Klasse* ist in der JPA-Spezifikation sowohl in der Version 1.0 als auch in der Version 2.0 als optional gekennzeichnet. Die bedeutet, dass ein JPA-Provider diese Alternative anbieten kann, aber nicht muss, um spezifikationskonform zu sein. Die drei von uns verwendeten JPA-Provider realisieren diese Alternative, wenn auch in unterschiedlichem Umfang.

Die Verwendung der Klassen erfolgt ohne Einschränkungen, soweit die Vererbungshierarchie isoliert betrachtet wird. Werden Assoziationen auf Entity-Klassen verwendet, die keine Blät-

ter der Vererbungshierarchie sind, kommt es zu Problemen bzw. Einschänkungen. Bei unserem zugrunde liegenden Beispiel ist genau dies der Fall. In Abbildung 5.1 auf Seite 134 ist die abstrakte Oberklasse der Vererbungshierarchie sowohl an einer n:1- als auch an einer 1:n-Beziehung beteiligt. Problematisch ist hier die 1:n-Beziehung, da diese auf Datenbankebene durch einen Fremdschlüssel in der Tabelle `Buchung` zu realisieren ist mit der Tabelle `Konto` als Ziel. Diese existiert jedoch nicht, sondern nur die Tabellen `Sparkonto` und `Girokonto`. Zur Verdeutlichung dient die Darstellung des Datenbankschemas in Abbildung 5.6.

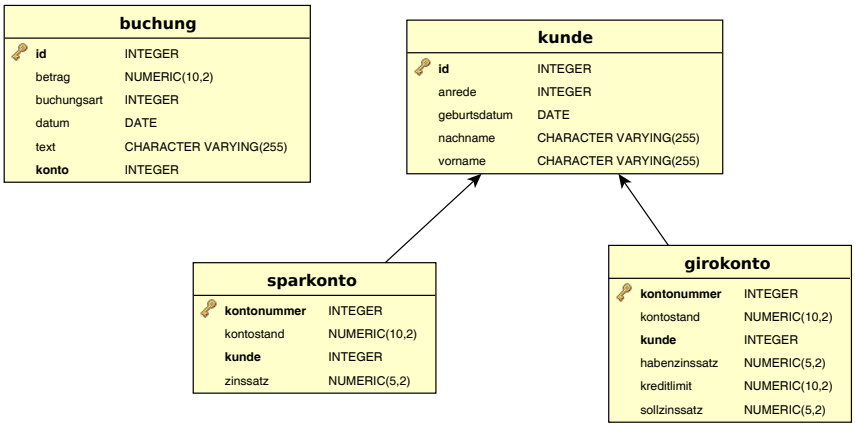


ABBILDUNG 5.6 Tabellenschema für die Vererbungsstrategie `TABLE_PER_CLASS`

Man erkennt, dass *keine* Fremdschlüsselbeziehung von der Tabelle `Buchung` zu den beiden Kontentabellen existiert und auch nicht existieren kann, da SQL derartige Fremdschlüsselbeziehungen nicht unterstützt. Das Kritische an dieser Realisierungsalternative ist also, dass der JPA-Provider einerseits durch die entsprechenden Annotationen prinzipiell über die Informationen bezüglich der Annotationen verfügt, um zu korrekten Ergebnissen zu kommen, andererseits aber andere Anwendungen, eventuell mit anderen Programmiersprachen realisiert, diese Informationen nicht besitzen. Außerdem kann das Datenbanksystem die Integrität der Anwendungsdaten nicht garantieren, da die entsprechenden Fremdschlüssel-Constraints nicht existieren.

EclipseLink und Hibernate kommen mit diesem Problem zurecht, OpenJPA nicht: Für das gegebene Beispiel mit bidirektionalen 1:n- und n:1-Beziehungen können EclipseLink und Hibernate auf das in Abbildung 5.6 dargestellte Schema mappen, OpenJPA nicht. Details über die von OpenJPA nicht unterstützten Assoziationsarten bei der Verwendung von `TABLE_PER_CLASS` sind im OpenJPA-Handbuch dokumentiert.

Achtung:

OpenJPA verbietet die Verwendung von 1:1- und n:1-Beziehungen zu Entities einer `TABLE_PER_CLASS`-Hierarchie, die keine Blätter der Hierarchie sind.

Um die Eindeutigkeit der Primärschlüssel in den konkreten Unterklassen garantieren zu können, müssen aber auch EclipseLink und Hibernate eine gewisse Einschränkung erzwingen:

Die Generierungsstrategie für das Primärschlüssel-Property für das Assoziationsziel, in unserem Beispiel die Klasse `Konto`, muss `TABLE` oder `SEQUENCE` sein. Der JPA-Provider muss garantieren können, dass Spar- und Girokonten verschiedene Primärschlüssel benutzen, da im Property `konto` der Klasse `Buchung` ein eindeutiger Wert verwendet werden muss. Dies ist mit `IDENTITY` jedoch nicht zu garantieren.

Interessant bei dieser Vererbungsalternative ist die Umsetzung von Anfragen, die in der Spezifikation bereits mit `Union` oder separaten `Select`-Abfragen pro Unterklasse vorgeschlagen wird. EclipseLink und OpenJPA erzeugen für unser Beispiel bei der JPQL-Abfrage „`Select k from Konto k`“ zwei einzelne `Select`-Abfragen während Hibernate insgesamt drei verschachtelte `Selects` mit `Union` erzeugt.

Projekt:

Das Projekt *vererbung-table-per-class* enthält den Code für die dritte Vererbungsalternative *eine Tabelle je konkreter Klasse*, die durch die Vererbungsstrategie `TABLE_PER_CLASS` realisiert wird.

5.4 Vergleich der Vererbungsstrategien

Wir haben in den letzten drei Abschnitten die in JPA vorgesehenen Möglichkeiten zur Realisierung von Vererbungshierarchien vorgestellt und zum Teil bereits deren Vor- und Nachteile genannt. Hier sollen nun noch einmal die Vor- und Nachteile explizit und vollständig aufgezählt werden, um dem Leser einen Vergleich der Vererbungsstrategien zu ermöglichen. Es gibt keine generell zu bevorzugende Alternative, und es sollten bei jeder zu realisierenden Vererbungshierarchie erneut die Vor- und Nachteile in der konkreten Aufgabenstellung abgewogen und eine Entscheidung für eine der drei Alternativen getroffen werden. Im Abschnitt 5.6 stellen wir anhand eines einfachen Beispiels die Kombination von Vererbungsstrategien vor. Dies erfolgt aus Gründen der Vollständigkeit. Im Allgemeinen ist von der Kombination abzuraten, da sie zu komplexeren, schlechter wartbaren und somit unerwünschten Strukturen führt.

5.4.1 Eine Tabelle für eine Vererbungshierarchie

Die Strategie `SINGLE_TABLE` ist der Default bei den Vererbungsstrategien. Durch die von JPA praktizierte *Convention over Configuration* legt dies eine gewisse Präferenz des Spezifikationsgremiums für diese Strategie nahe.

Vorteile `SINGLE_TABLE`:

- Volle Unterstützung objektorientierter Polymorphie
- Sehr performante Abfragen möglich, da keine Joins
- Einfaches konzeptionelles Modell
- Einfache Syntax. Im Minimalfall kein zusätzliches JPA-Mapping benötigt

Nachteile SINGLE_TABLE:

- NOT-NULL-Constraints nicht möglich
- Minimaler Speicherbedarf für die NULL-Werte
- Datenbank-Schema nicht in Normalform, da funktionale Abhängigkeiten

5.4.2 Eine Tabelle je Unterklasse

Die Strategie JOINED entspricht der objektorientierten Denkweise, da es eine 1:1-Entsprechung zwischen einer Klasse und einer Tabelle gibt.

Vorteile JOINED:

- Volle Unterstützung objektorientierter Polymorphie
- 1:1-Entsprechung zwischen Klasse und Tabelle
- Tabellen in Normalform

Nachteile JOINED:

- Abfragen verwenden Joins und sind daher aufwendig. Der Join findet jedoch über einen singulären Fremdschlüssel statt, so dass der Aufwand begrenzt ist.

5.4.3 Eine Tabelle je konkreter Klasse

Die Strategie TABLE_PER_CLASS ist optional, muss also von einem JPA-Provider nicht implementiert werden, so dass sich Portabilitätsprobleme ergeben können. Die drei von uns verwendeten Provider stellen die Strategie zur Verfügung.

Vorteile TABLE_PER_CLASS:

- Sehr performante Abfragen möglich, da keine Joins
- Einfache Verwendung in anderen Programmiersprachen
- Keine Änderung an bestehenden Tabellen beim Einfügen von Blättern in der Vererbungshierarchie

Nachteile TABLE_PER_CLASS:

- Aufwendige Abfrage durch mehrere Select- und/oder Union-Anweisungen
- Schwache objektorientierte Polymorphie
- Schemaredundanz in Unterklassen, daher nicht in Normalform
- Keine expliziten Fremdschlüssel auf Datenbankebene
- Verwendung von Beziehungen eventuell eingeschränkt
- In der Spezifikation als optional gekennzeichnet; daher eventuell nicht portabel

5.5 Mapping von Oberklassen

Entities können von Oberklassen erben, die persistente Daten und Mapping-Informationen bereitstellen, selbst aber keine Entities sind. Eine solche Oberklasse wird *Mapped Super-*

class genannt und mit `@MappedSuperclass` annotiert. Die Klasse dient als Verteiler von Mapping-Informationen in die Unterklassen, darf aber selbst nicht als Entity verwendet werden: Die Klasse lässt sich nicht in Entity-Manager-Methoden und JPQL-Abfragen verwenden. Sie ist auch nicht als Ziel von Beziehungen erlaubt.

Im folgenden Beispiel werden Firmen- und Privatkunden und deren Adressen verwaltet. Dazu wird die Klasse `Kunde` als gemappte Oberklasse mit `@MappedSuperclass` annotiert. Die Klasse `Adresse` ist eine gewöhnliche Entity-Klasse.

```
@MappedSuperclass
public abstract class Kunde {

  @Id @GeneratedValue
  private Integer id;
  @Version
  private Integer version;
  @ManyToOne
  @JoinColumn(name = "adresse")
  private Adresse adresse;
  ...
```

```
@Entity
public class Adresse {

  @Id @GeneratedValue
  private Integer id;
  private String strasse;
  private String hausnummer;
  private String plz;
  private String ort;
  ...
```

Die Beziehung zwischen Kunden und Adressen darf nicht bidirektional sein, da gemappte Oberklassen nicht als Ziel einer Beziehung erlaubt sind. Firmen- und Privatkunden werden ohne zusätzliche vererbungsspezifische Annotationen als Unterklasse definiert.

```
@Entity
public class Privatkunde
       extends Kunde {

  private String vorname;
  private String nachname;
  @Temporal(TemporalType.DATE)
  private Date geburtsdatum;
  private Anrede anrede;
  ...
```

```
@Entity
public class Firmenkunde
       extends Kunde {

  private String
      firmenname;
  ...
```

Die gemappte Oberklasse kann abstrakt sein, muss es aber nicht. Für den abstrakten Fall entspricht das Mapping der Vererbungsalternative `TABLE_PER_CLASS`, da die Properties in die Unterklassen wandern. Der Unterschied liegt in der nicht vorhandenen Tabelle für die Oberklasse `Kunde`. Abbildung 5.7 zeigt das entsprechende Schema. Die bei `TABLE_PER_CLASS` fehlenden Fremdschlüssel-Constraints sind vorhanden.

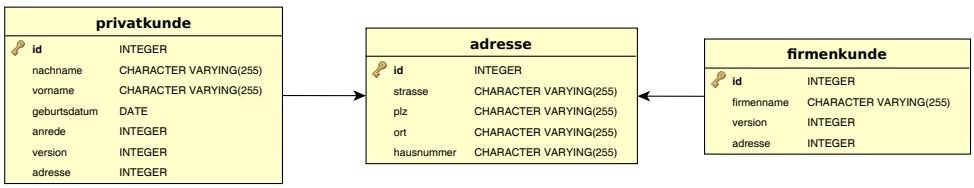


ABBILDUNG 5.7 Tabellenschema für das Beispiel mit `@MappedSuperclass`

Hinweis:

OpenJPA erlaubt die Verwendung von gemappten Oberklassen in Queries und Entity-Manager-Methoden. Das Ergebnis enthält entsprechende Unterklasseninstanzen.

Wir haben im Beispiel die gemappte Oberklasse fachlich verwendet und Kunden sowie Firmen- und Privatkunden modelliert. Die Verwendung von `@MappedSuperclass` ist technisch motiviert: Sollen bestimmte Entities rein technisch motivierte persistente Daten erhalten, so scheidet hierfür ein Entity als Oberklasse aus, da es keine Fachlichkeit (keine fachlichen Daten) enthält. Eine gemappte Oberklasse kann diese technisch motivierten Daten jedoch einfach an alle Unterklassen vererben. Ein Beispiel hierfür sind etwa Primärschlüssel (`@Id`) und Versionsproperty (`@Version`), wie wir sie bereits im Beispiel verwendet haben, sowie ein Zeitstempel für den initialen Eintrag in der Datenbank und ein Zeitstempel für die letzte Änderung. Für derartige Erweiterungen in Richtung historische Datenverarbeitung existieren jedoch auch dedizierte Frameworks. Eines davon, Envers, stellen wir in Abschnitt 11.4 vor.

Projekt:

Den Programm-Code für gemappte Oberklassen finden Sie im Projekt *vererbung-mapped-superclass*.

5.6 Kombination von Vererbungsstrategien

In den Abschnitten 5.1, 5.2 und 5.3 wurde die `@Inheritance`-Annotation in ihren drei alternativen Verwendungen für die jeweils komplette Vererbungshierarchie der Beispiele eingesetzt. Die `@MappedSuperclass`-Annotation im Abschnitt 5.5 wurde verwendet, um Objektzustände ohne `@Entity` persistent zu machen. Die Kombination all dieser Annotationen und die Verwendung nicht persistenter Klassen innerhalb ein und derselben Vererbungshierarchie ist jedoch auch möglich. Die JPA-Spezifikation erwähnt explizit, dass der Einsatz verschiedener Vererbungsstrategien innerhalb einer Entity-Hierarchie durch einen JPA-Provider *nicht* unterstützt werden muss und daher nicht portabel ist. Wir raten Ihnen von einer Kombination verschiedener Vererbungsstrategien innerhalb einer Hierarchie ab, da die Tabellenstrukturen, aber auch die Handhabung auf Java-Seite naturgemäß komplexer werden. Nichtsdestotrotz ist die Kombination möglich, und wir wollen sie an dieser Stelle exemplarisch erläutern.

Hinweis:

Die Kombination verschiedener Vererbungsstrategien innerhalb einer Hierarchie muss von einem JPA-Provider nicht unterstützt werden. Wir raten daher im allgemeinen Fall von einer derartigen Kombination ab. Falls entsprechende Gründe für eine Kombination sprechen, raten wir zur ausschließlichen Verwendung von `SINGLE_TABLE` und `JOINED`.

Um die Vererbungsstrategien `SINGLE_TABLE` und `JOINED` einsetzen zu können, müssen wir unsere Vererbungshierarchie vergrößern. Abbildung 5.8 zeigt die neue Hierarchie, in die ein Kontokorrent- und ein Tagesgeldkonto integriert wurden. Falls die Fachlichkeit nicht der Realität entspricht, bitten wir den bankkaufmännisch gebildeten Leser um Nachsicht.

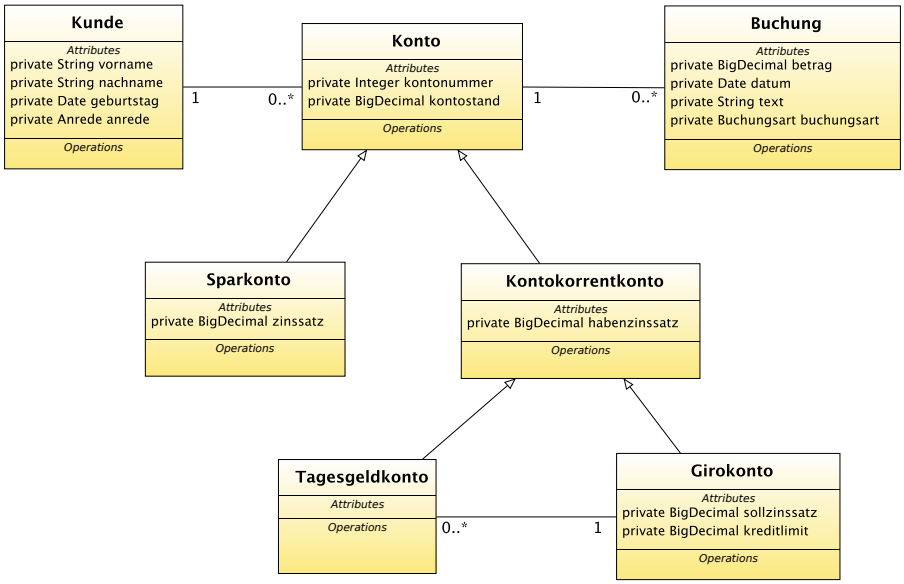


ABBILDUNG 5.8 Klassenhierarchie für kombinierte Vererbungsstrategie

Die Klassen `Konto` und `Kontokorrentkonto` sind abstrakt und jeweils die Wurzel einer entsprechenden Vererbungsstrategie für den darunter befindlichen Teilbaum. Die verwendeten Strategien sind `JOINED` für das Entity `Konto` und `SINGLE_TABLE` für das Entity `Kontokorrentkonto`.

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class Konto {

  @Id @GeneratedValue
  private Integer kontonummer;
  ...
}
```

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
public abstract class Kontokorrentkonto extends Konto {
  ...
}
```

Das zugrunde liegende Tabellenschema ist in Abbildung 5.9 dargestellt. Bemerkenswert bei diesem Schema ist eine doppelte Fremdschlüsselbeziehung von der Tabelle `Kontokorrentkonto` zur Tabelle `Konto`. Eine Fremdschlüsselbeziehung ist der Vererbungsstrategie `JOINED` geschuldet, damit Instanzen von `Konto`-Unterklassen mit den entsprechenden Daten von `Konto` gejoint werden können. Die zweite Fremdschlüsselbeziehung repräsentiert die n:1-Beziehung eines Tagegeldkontos zu seinem Referenzkonto, einem Girokonto. Der folgende Code-Ausschnitt skizziert dies.

```
@Entity
public class Tagesgeldkonto extends Kontokorrentkonto {

  @ManyToOne
  private Girokonto referenzkonto;
  ...
}
```

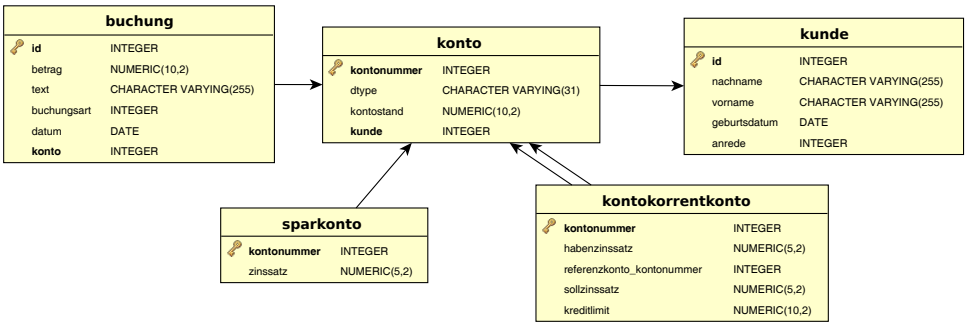


ABBILDUNG 5.9 Tabellenschema für kombinierte Vererbungsstrategie

Die in Abbildung 5.8 dargestellten Klassen sind durch den Verzicht auf die Vererbungsstrategie `TABLE_PER_CLASS` in vollem Umfang polymorph verwendbar: Über einen Kunden kann zu seinen Konten navigiert werden, und JPQL-Anfragen für Konten liefern Instanzen sämtlicher Unterklassen. Beide Anwendungsfälle sind als Beispiele im herunterladbaren Projekt enthalten.

Projekt:

Den Programm-Code für die Kombination von Vererbungsstrategien finden Sie im Projekt *vererbung-mixed*.