Component and Inheritance Mapping

Originals of Slides and Source Code for Examples:
http://courses.coreservlets.com/Course-Materials/hibernate.html

Customized Java EE Training: http://courses.coreservlets.com/
Servlets, JSP, Struts, JSF/MyFaces/Facelets, Ajax, GWT, Spring, Hibernate/JPA, Java 5 & 6.
Developed and taught by well-known author and developer. At public venues or onsite at your location.

For live Spring & Hibernate training, see courses at http://courses.coreservlets.com/.

Taught by the experts that brought you this tutorial.
Available at public venues, or customized versions can be held on-site at your organization.

* Courses developed and taught by Marty Hall
  – Java 5, Java 6, intermediate/beginning servlets/JSP, advanced servlets/JSP, Struts, JSF, Ajax, GWT, custom mix of topics
* Courses developed and taught by coreservlets.com experts (edited by Marty)
  – Spring, Hibernate/JPA, EJB3, Ruby/Rails

Contact hall@coreservlets.com for details
Topics in This Section

- Understand the difference between Components and Entities
- Walk through some uses of Components, and their mapping
- Look at different strategies and implementations for realizing inheritance

Component

- Refers to the UML modeling term of composition
- Does not exist on its own; dependent on a parent object
  - Does not have a table or identifier in the database
  - Only associated to a single parent class
- Commonly referred to as a “has a” relationship
  - An AccountOwner has an Address
Entity

- 1st class citizen -- lives on its own
- Has its own table and identifier
- Can be made up of multiple components
- Can be related/associated to other entities

Entity/Component Example

- ‘AccountOwner’ is an Entity
  - Can lives on its own
  - Has its own id
- ‘Address’ is a component of ‘AccountOwner’
  - Non-existent without AccountOwner
  - No id of its own.

<table>
<thead>
<tr>
<th>ACCOUNT_OWNER</th>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCOUNT_OWNER_ID</td>
<td>NUMBER</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>LAST_NAME</td>
<td>VARCHAR(50)</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>FIRST_NAME</td>
<td>VARCHAR(50)</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>SOCIAL_SECURITY_NUMBER</td>
<td>VARCHAR(20)</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>STREET_ADDRESS</td>
<td>VARCHAR(50)</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>CITY</td>
<td>VARCHAR(50)</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>STAIL</td>
<td>VARCHAR(20)</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>ZIP_CODE</td>
<td>VARCHAR(20)</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>HOME_PHONE</td>
<td>VARCHAR(20)</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>CELL_PHONE</td>
<td>VARCHAR(20)</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>

```java
AccountOwner
- accountOwner: long
- lastName: String
- firstName: String
- socialSecurityNumber: String
- address: Address
- homePhone: String
- cellPhone: String

Address
- city: String
- streetAddress: String
- zipCode: String
- accountOwner: AccountOwner
```
Using Components

1. **Determine your domain model**
   - Which objects are best suited to be a component?

2. **Create your database tables**
   - Model your components accordingly within entity tables

3. **Create your Java classes**
   - One for each entity, and one for each component
   - Setup your components within the entity classes

4. **Write the Hibernate mapping file for the entity, using the embedded component tag within it to identify the component class.**

---

Mapping a Component

```xml
<class name="courses.hibernate.vo.AccountOwner"
  table="ACCOUNT_OWNER">
  <id name="accountOwnerId" column="ACCOUNT_OWNER_ID">
    <generator class="native" />
  </id>
  <property name="lastName" column="LAST_NAME" type="string" />
  <property name="firstName" column="FIRST_NAME" type="string" />
  <property name="socialSecurityNumber" column="SOCIAL_SECURITY_NUMBER" type="string" />
  <component name="address" class="courses.hibernate.vo.Address">
    <parent name="accountOwner"/>
    <property name="streetAddress" column="STREET_ADDRESS" type="string" />
    <property name="city" column="CITY" type="string" />
    <property name="state" column="STATE" type="string" />
    <property name="zipCode" column="ZIP_CODE" type="string" />
  </component>
  <property name="homePhone" column="HOME_PHONE" type="string" />
  <property name="cellPhone" column="CELL_PHONE" type="string" />
</class>
```
Nested Components

- Component ‘Address’ has nested component ‘ZipCode’
  - Example: 12222-1234

Mapping Nested Components

```xml
<class name="courses.hibernate.vo.AccountOwner"
  table="ACCOUNT_OWNER">
  ...
  <component name="address" class="courses.hibernate.vo.Address">
    <parent name="accountOwner"/>
    <property name="streetAddress" column="STREET_ADDRESS" type="string"/>
    <property name="city" column="CITY" type="string"/>
    <property name="state" column="STATE" type="string"/>
    <component name="zipCode" class="courses.hibernate.vo.ZipCode">
      <property name="zip" column="ZIP_CODE" type="string"/>
      <property name="plus4" column="ZIP_PLUS_FOUR" type="string"/>
    </component>
  </component>
  ...
</class>
```
Collection of Components

```xml
<class name="courses.hibernate.vo.Account" table="ACCOUNT">
  ...
  <set name="accountOwnerAddresses" table="ACCOUNT_ACCOUNT_OWNER">
    <key column="ACCOUNT_ID" />
    <composite-element class="courses.hibernate.vo.Address">
      <property name="streetAddress" type="string"
        formula="(SELECT AO.STREET_ADDRESS FROM ACCOUNT_OWNER AO
                 WHERE AO.ACCOUNT_OWNER_ID = ACCOUNT_OWNER_ID)" />

      <property name="city" type="string"
        formula="(SELECT AO.CITY FROM ACCOUNT_OWNER AO WHERE
                 AO.ACCOUNT_OWNER_ID = ACCOUNT_OWNER_ID)" />

      <property name="state" type="string"
        formula="(SELECT AO.STATE FROM ACCOUNT_OWNER AO WHERE
                 AO.ACCOUNT_OWNER_ID = ACCOUNT_OWNER_ID)" />
    </composite-element>
  </set>
  ...
</class>
```

Component as Entity ID

- **Done with or without a separate Java class**
  - If using Java class
    - Must implement Comparable and Serializable
    - Must define ‘class’ attribute in mapping file
- **If composite key contains identifiers from associations:**
  - Set the id value when you set the corresponding association on the main object
  - Set insert=‘false’ and update=‘false’ on the association on the main object
Component as Entity ID

- Example: EBill uses EBillId component as primary key
  - EBillId component class contains attributes required to uniquely identify the EBill

**EBillId Class**

```java
public class EBillId implements Comparable<EBillId>, Serializable {
    private long accountId;
    private long ebillerId;
    private Date dueDate;
    ...
    // getters and setters
    ...
}
```

Component as Entity ID

- Example: EBill uses EBillId component as primary key
  - In EBill entity class, EBillId values are set as appropriate setters are called.

**EBill Class**

```java
public class EBill {
    private EBillId ebillId = new EBillId();
    private EBiller ebiller;
    ...
    protected void setEbiller(EBiller ebiller) {
        this.ebiller = ebiller;
        ebillId.setEbillerId(ebiller.getEbillerId());
        ...
    }
    ...
}
```
Component as Entity ID

Example: EBill uses EBillId component as primary key

EBill Mapping File

```xml
<class name="courses.hibernate.vo.EBill" table="EBILL">
  <composite-id name="ebillId" class="courses.hibernate.vo.EBillId">
    <key-property name="accountId" column="ACCOUNT_ID" type="long"/>
    <key-property name="ebillerId" column="EBILLER_ID" type="long"/>
    <key-property name="dueDate" column="DUE_DATE" type="timestamp"/>
  </composite-id>
  ...
  <!-- Must have insert="false" update="false" because ids for the objects are part of the composite key. The relationships are managed via the composite key elements rather than the M:1 relationship. -->
  <many-to-one name="ebiller" column="EBILLER_ID" class="courses.hibernate.vo.EBiller" access="field" insert="false" update="false"/>
  ...
</class>
```

Component as Entity ID Caveat

- Using a Component as an Entity ID and not using a separate class to represent it can be troublesome
- You can not simply pass in the attributes which make up the key into a session.get() method
  - session.get(EBill.class, int x, int y)
  - No such api on Session object
- Need to write a separate Hibernate query to bring back the object
Inheritance

- Allows for logical affiliation of classes with common state and/or behavior
- Commonly referred to as an “is a” relationship
  - A Cat is a Animal
  - A Dog is a Animal
- Polymorphism
  - Dynamic realization of subclass behavior while treating the object as an instance of its superclass
public class Animal {
    public String speak() {
        return "Generic Hello";
    }
}

public class Cat extends Animal {
    public String speak() {
        return "Meow";
    }
}

public class Dog extends Animal {
    public String speak() {
        return "Woof";
    }
}

public class Farm {
    List<Animals> animals = someDAO.getAnimals();
    for (int index=0; index < animals.size(); index++) {
        Animal someAnimal = animals.get(index);
        System.out.println(someAnimal.speak());
    }
}

Even though the variable type in our List is 'Animal'...

Assuming the list of animals returned included (in order) a
cat, then a dog, the output would be:

Meow
Woof

... polymorphism allows each instance
to make use of it’s own implementation
Inheritance Realization

• Easy to do in an object oriented language
  – Java: Use ‘extends’ or ‘implements’ keyword
• Not so easy to do in a relational database
  – Tables do not ‘extend’ from each other
  – Part of the ‘impedance mismatch’ problem
• How do we get around this?
  – Four approaches through Hibernate
    • Hibernate implicit polymorphism
    • Table-per-concrete class
    • Table-per-class-hierarchy
    • Table-per-subclass

Modeling Inheritance

1. Determine your domain model
   • What objects have hierarchical relationships?
2. Choose your inheritance strategy
   • Hibernate implicit polymorphism
   • Table-per-concrete class
   • Table-per-class-hierarchy
   • Table-per-subclass
3. Create your database tables based on the chosen strategy
4. Code your Java objects, using ‘extends’ or ‘implements’
5. Write your Hibernate mapping file using the appropriate subclass tags
Implicit Polymorphism

- **Database**
  - *One* database table *per* concrete class

- **Hibernate Mapping Files**
  - *Separate mapping files for each inherited class*
    - Like normal, including any inherited properties

- **Default Behavior**
  - Out of the box Hibernate will automatically recognize any Java inheritance associations

---

**Implicit Polymorphism**

- One table per concrete class

<table>
<thead>
<tr>
<th>CHECKING_ACCOUNT</th>
<th>SAVINGS_ACCOUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Column Name</td>
<td>Data Type</td>
</tr>
<tr>
<td>CHECKING_ACCOUNT_ID</td>
<td>NUMBER</td>
</tr>
<tr>
<td>CREATION_DATE</td>
<td>TIMESTAMP(6)</td>
</tr>
<tr>
<td>BALANCE</td>
<td>NUMBER(10,2)</td>
</tr>
<tr>
<td>CHECK_STYLE</td>
<td>VARCHAR(50)</td>
</tr>
</tbody>
</table>

<<Abstract Class>>

Account
- creatorDate : Date
- balance : double

Checking Account
- checkingAccountId : long
- checkStyle : String

Savings Account
- savingsAccountId : long
- InterestRate : double
Implicit Polymorphism

- CheckingAccount mapping file

```xml
<class name="courses.hibernate.vo.CheckingAccount" table="CHECKING_ACCOUNT">
  <id name="checkingAccountId" column="CHECKING_ACCOUNT_ID">
    <generator class="native"/>
  </id>
  
  <property name="creationDate" column="CREATION_DATE" type="timestamp" update="false"/>
  
  <property name="balance" column="BALANCE" type="double"/>
  
  <property name="checkStyle" column="CHECK_STYLE" type="string"/>
</class>
```

- SavingsAccount mapping file

```xml
<class name="courses.hibernate.vo.SavingsAccount" table="SAVINGS_ACCOUNT">
  <id name="savingsAccountId" column="SAVINGS_ACCOUNT_ID">
    <generator class="native"/>
  </id>
  
  <property name="creationDate" column="CREATION_DATE" type="timestamp" update="false"/>
  
  <property name="balance" column="BALANCE" type="double"/>
  
  <property name="interestRate" column="INTEREST_RATE" type="double"/>
</class>
```
Implicit Polymorphism

• **Advantages**
  – Get it for free. Hibernate automatically scans classes on startup (including inheritance); No additional configuration/mapping needed
  – Not a lot of nullable columns *(good for integrity)*
  – Queries against individual types are fast and simple

• **Disadvantages**
  – Makes handling relationships difficult
    • One-to-many relationships typically require a foreign key, but inherited associations can’t key to both tables; Databases don’t support it *(Example: AccountOwner:Account)*
  – Polymorphic queries are process intensive
    • For queries against the superclass, Hibernate executes multiple queries
      – `SELECT * FROM CHECKING_ACCOUNT WHERE ACCOUNT_OWNER_ID=?`
      – `SELECT * FROM SAVINGS_ACCOUNT WHERE ACCOUNT_OWNER_ID=?`
  – Database schema evolution more complex
    • Need to add the same column to multiple tables
    • Integrity constraints might have to span multiple tables
Table-per-concrete class

- **Database**
  - *One* database table *per* concrete class

- **Hibernate Mapping**
  - *Single mapping file*
    - Based on superclass
    - Includes ‘union-subclass’ definitions for inherited classes

---

**Table-per-concrete class**

- **One table per concrete class**

<table>
<thead>
<tr>
<th>Checking Account</th>
<th>SAVINGS_ACCOUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Column Name</td>
<td>Data Type</td>
</tr>
<tr>
<td>ACCOUNT_ID</td>
<td>NUMBER</td>
</tr>
<tr>
<td>CREATION_DATE</td>
<td>TIMESTAMP(6)</td>
</tr>
<tr>
<td>BALANCE</td>
<td>NUMBER(10,2)</td>
</tr>
<tr>
<td>CHECK_STYLE</td>
<td>VARCHAR2(60)</td>
</tr>
</tbody>
</table>
Table-per-concrete class

- **Account Mapping File**
  
  ```xml
  <class name="Account" abstract="true">
    <id name="accountId" column="ACCOUNT_ID" type="long">
      <generator class="native"/>
    </id>

    <property name="creationDate" column="CREATION_DATE" type="timestamp"/>
    <property name="balance" column="BALANCE" type="double"/>

    <union-subclass name="courses.hibernate.vo.SavingsAccount" table="SAVINGS_ACCOUNT">
      <property name="interestRate" column="INTEREST_RATE" type="double"/>
    </union-subclass>

    <union-subclass name="courses.hibernate.vo.CheckingAccount" table="CHECKING_ACCOUNT">
      <property name="checkStyle" column="CHECK_STYLE" type="string"/>
    </union-subclass>
  </class>
  ```

- **Advantages**
  - Shared mapping of common elements
    - Shared database id
  - Not a lot of nullable columns (*good for integrity*)
  - Queries against individual types are fast and simple
  - Less SQL statements generated with use of ‘Union’ for polymorphic queries

- **Disadvantages**
  - Still have difficulty with relationships
    - Foreign keying to two tables not possible
  - Database schema evolution still more complex
    - Need to add the same column to multiple tables
    - Integrity constraints might have to span multiple tables
**Table-per-class-hierarchy**

- **Database**
  - *One* database table for *all* subclasses
  - Denormalized table has columns for all attributes

- **Hibernate Mapping**
  - *Single mapping file* still based on superclass
  - Includes ‘subclass’ definitions for inherited classes
  - Use ‘discriminator’ column/field to identity concrete type

---

**Table-per-class-hierarchy**

- **One table for all inherited classes**

```
ACCOUNT

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Data Type</th>
<th>Nullable</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCOUNT_ID</td>
<td>NUMBER</td>
<td>No</td>
</tr>
<tr>
<td>CREATION_DATE</td>
<td>TIMESTAMP(6)</td>
<td>No</td>
</tr>
<tr>
<td>BALANCE</td>
<td>NUMBER(10,2)</td>
<td>No</td>
</tr>
<tr>
<td>ACCOUNT_TYPE</td>
<td>VARCHAR2(1)</td>
<td>No</td>
</tr>
<tr>
<td>CHECK_STYLE</td>
<td>VARCHAR2(50)</td>
<td>Yes</td>
</tr>
<tr>
<td>INTEREST_RATE</td>
<td>NUMBER(10,2)</td>
<td>Yes</td>
</tr>
</tbody>
</table>
```

---

Account Types:
- Checking Account
  - checkStyle : String
- Savings Account
  - interestRate : double
Table-per-class-hierarchy

• Account Mapping File

```xml
<class name="Account" table="ACCOUNT" abstract="true">
  <id name="accountId" column="ACCOUNT_ID" type="long"
     generator="native"/>
  <discriminator column="ACCOUNT_TYPE" type="string"/>
  <property name="creationDate" column="CREATION_DATE"
    type="timestamp"/>
  <property name="balance" column="BALANCE" type="double"/>

  <subclass name="courses.hibernate.vo.SavingsAccount"
    discriminator-value="S">
    <property name="interestRate" column="INTEREST_RATE"/>
  </subclass>

  <subclass name="courses.hibernate.vo.CheckingAccount"
    discriminator-value="C">
    <property name="checkStyle" column="CHECK_STYLE"/>
  </subclass>
</class>
```

Table-per-class-hierarchy

• Advantages
  – Simple
  – Fast reads/writes, even across types

• Disadvantages
  – Lots of nullable columns
    • Possible data integrity concern
  – Denormalized table generally considered bad database design
Table-per-subclass

- **Database**
  - *One* database table for the superclass **AND** one *per* subclass
  - Shared columns in superclass table
  - Subclass tables have their object-specific columns

- **Hibernate Mapping File**
  - *Single mapping file* based on the superclass
  - Includes *joined-subclass* definitions for inherited classes

**Example: Table-per-subclass**

- Every class that has persistent properties has its own table
  - Each table contains a primary key, and non-inherited properties
  - Inheritance is realized through foreign keys
Table-per-subclass

• Account Mapping File

```xml
<class name="Account" table="ACCOUNT" abstract="true">
  <id name="accountId" column="ACCOUNT_ID" type="long">
    <generator class="native"/>
  </id>
  <property name="creationDate" column="CREATION_DATE" type="timestamp"/>
  <property name="balance" column="BALANCE" type="double"/>
  <joined-subclass name="courses.hibernate.vo.SavingsAccount" table="SAVINGS_ACCOUNT">
    <key column="SAVINGS_ACCOUNT_ID"/>
    <property name="interestRate" column="INTEREST_RATE" type="double"/>
  </joined-subclass>
  <joined-subclass name="courses.hibernate.vo.CheckingAccount" table="CHECKING_ACCOUNT">
    <key column="CHECKING_ACCOUNT_ID"/>
    <property name="checkStyle" column="CHECK_STYLE" type="string"/>
  </joined-subclass>
</class>
```

Table-per-subclass

• Advantages
  – Normalized schema
    • Schema evolution and integrity are straightforward
  – Reduced number of SQL statements produced
    • Hibernate uses inner joins for subclass queries, outer joins for polymorphic ones

• Disadvantages
  – Can have poor performance for complex systems
    • Requires many joins or sequential reads for queries
When to use Which

• Leave implicit polymorphism for queries against interfaces (based on behavior, not different attributes)

• If you rarely require polymorphic queries, lean towards table-per-concrete-class.

• If polymorphic behavior is required, AND subclasses have only a few distinct properties, try table-per-class-hierarchy

• If polymorphic AND many distinct properties, look at table-per-subclass or table-per-concrete-class, weighing the cost of joins versus unions.

Wrap-up

Customized Java EE Training: http://courses.coreservlets.com/
Servlets, JSP, Struts, JSF/MyFaces/Facelets, Ajax, GWT, Spring, Hibernate/JPA, Java 5 & 6.
Developed and taught by well-known author and developer. At public venues or onsite at your location.
Summary

• In this lecture, we:
  – Saw how Components are different from Entities
    • Entities have their own IDs
    • Component are dependant on Entities
  – Learned how to use Components and Entities to represent
    • Subsets of logical data,
    • Identifiers for Entities,
  – Talked about the different methods of modeling
    inheritance relationships through
    Java/Database/Hibernate, and where each one is best
    suited.
    • Hibernate implicit polymorphism
    • Table-per-concrete class
    • Table-per-class-hierarchy
    • Table-per-subclass

Preview of Next Sections

• Stages of an object’s lifecycle within Hibernate
• Closer look at the Hibernate persistence process
• Session Management
Questions?

Customized Java EE Training: http://courses.coreservlets.com/
Servlets, JSP, Struts, JSF/MyFaces/Facelets, Ajax, GWT, Spring, Hibernate/JPA, Java 5 & 6.
Developed and taught by well-known author and developer. At public venues or onsite at your location.