Java Back-End Web App Tutorial Part 6: Inheritance in Class Hierarchies

How to deal with class hierarchies in a Java EE back-end web app

Gerd Wagner <G.Wagner@b-tu.de>
Mircea Diaconescu <M.Diaconescu@b-tu.de>

Java Back-End Web App Tutorial Part 6: Inheritance in Class Hierarchies: How to deal with class hierarchies in a Java EE back-end web app

by Gerd Wagner and Mircea Diaconescu

Warning: This tutorial manuscript may still contain errors and may still be incomplete in certain respects. Please report any issue to Gerd Wagner at G.Wagner@b-tu.de.

This tutorial is also available in the following formats: PDF [subtyping-tutorial.pdf]. You may run the example app [http://yew.informatik.tu-cottbus.de/tutorials/java/subtypingapp] from our server, or download it as a ZIP archive file [SubtypingApp.zip]. See also our Web Engineering project page [http://web-engineering.info/].

Publication date 2018-07-03

Table of Contents

List of Figures

Foreword

This tutorial is Part 6 of our series of six tutorials [http://web-engineering.info/JavaJpaJsfApp] about model-based development of front-end web applications with Java by using JPA and JSF frameworks. It shows how to build a web app that manages subtype (inheritance) relationships between object types.

The app supports the four standard data management operations (**C**reate/**R**ead/**U**pdate/**D**elete). It is based on the example used in the other parts, with the object types `Book`, `Person`, `Author`, `Employee` and `Manager`. The other parts are:

- Part 1 [minimal-tutorial.html]: Building a **minimal** app.
- Part 2 [validation-tutorial.html]: Handling **constraint validation**.
- Part 3 [enumeration-tutorial.html]: Dealing with **enumerations**.
- Part 4 [unidirectional-association-tutorial.html]: Managing **unidirectional associations** between books and publishers, assigning a publisher to a book, and between books and authors, assigning authors to a book.
- Part 5 [bidirectional-association-tutorial.html]: Managing **bidirectional associations** between books and publishers and between books and authors, also assigning books to authors and to publishers.

Chapter 1. Subtyping and Inheritance

The concept of a *subtype*, or *subclass*, is a fundamental concept in natural language, mathematics, and informatics. For instance, in English, we say that *a bird is an animal*, or the class of all birds is a *subclass* of the class of all animals. In linguistics, the noun "bird" is a *hyponym* of the noun "animal".

An object type may be specialized by subtypes (for instance, *Bird* is specialized by *Parrot*) or generalized by supertypes (for instance, *Bird* and *Mammal* are generalized by *Animal*). Specialization and generalization are two sides of the same coin.

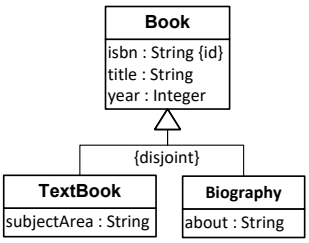
A subtype **inherits** all features from its supertypes. When a subtype inherits attributes, associations and constraints from a supertype, this means that these features need not be explicitly rendered for the subtype in the class diagram, but the reader of the diagram has to know that all features of a supertype also apply to its subtypes.

When an object type has more than one direct supertype, we have a case of **multiple inheritance**, which is common in conceptual modeling, but prohibited in many object-oriented programming languages, such as Java and C#, where subtyping leads to **class hierarchies** with a unique direct supertype for each object type.

1. Introducing Subtypes by Specialization

A new subtype may be introduced by specialization whenever new features of more specific types of objects have to be captured. We illustrate this for our example model where we want to capture text books and biographies as special cases of books. This means that text books and biographies also have an ISBN, a title and a publishing year, but in addition they have further features such as the attribute `subjectArea` for text books and the attribute `about` for biographies. Consequently, we introduce the object types `TextBook` and `Biography` by specializing the object type `Book`, that is, as subtypes of `Book`.

Figure 1.1. The object type `Book` with two subtypes: `TextBook` and `Biography`



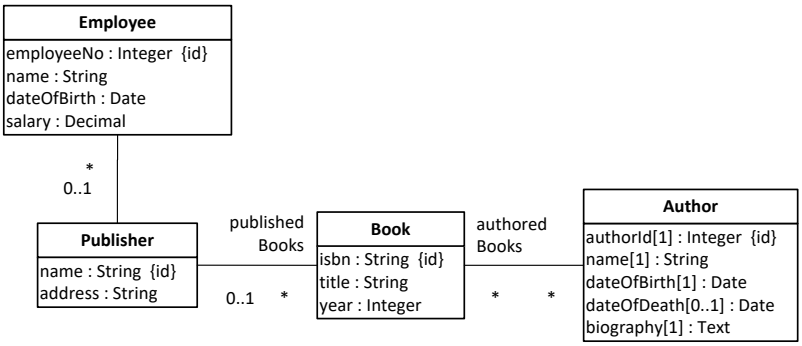
When specializing an object type, we define additional features for the newly added subtype. In many cases, these additional features are more specific properties. For instance, in the case of `TextBook` specializing `Book`, we define the additional attribute `subjectArea`. In some programming languages, such as in Java, it is therefore said that the subtype **extends** the supertype.

However, we can also specialize an object type without defining additional properties (or operations/methods), but by defining additional constraints.

2. Introducing Supertypes by Generalization

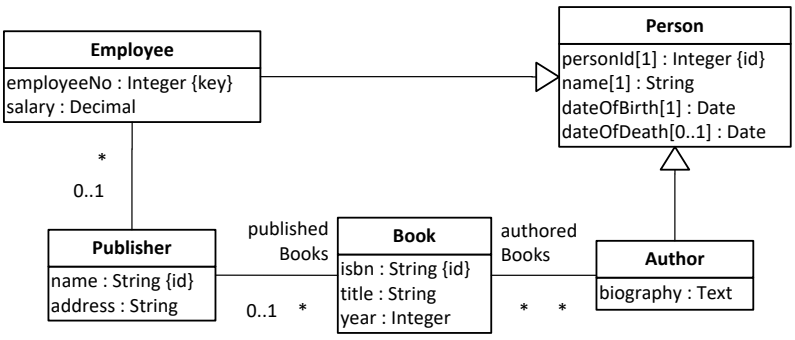
We illustrate generalization with the following example, which extends the information model of Part 4 by adding the object type `Employee` and associating employees with publishers.

Figure 1.2. The object types `Employee` and `Author` share several attributes



After adding the object type `Employee` we notice that `Employee` and `Author` share a number of attributes due to the fact that both employees and authors are people, and *being an employee* as well as *being an author* are **roles** played by people. So, we may generalize these two object types by adding a joint supertype `Person`, as shown in the following diagram.

Figure 1.3. The object types `Employee` and `Author` have been generalized by adding the common supertype `Person`



When generalizing two or more object types, we move (and centralize) a set of features shared by them in the newly added supertype. In the case of `Employee` and `Author`, this set of shared features consists of the attributes `name`, `dateOfBirth` and `dateOfDeath`. In general, shared features may include attributes, associations and constraints.

Notice that since in an information design model, each top-level class needs to have a standard identifier, in the new class `Person` we have declared the standard identifier attribute `personId`, which is inherited by all subclasses. Therefore, we have to reconsider the attributes that had been declared to be standard identifiers in the subclasses before the generalization. In the case of `Employee`, we had declared the attribute `employeeNo` as a standard identifier. Since the employee number is an important business information item, we have to keep this attribute, even if it is no longer the standard identifier. Because it is still an alternative identifier (a "key"), we define a *uniqueness* constraint for it with the constraint keyword `key`.

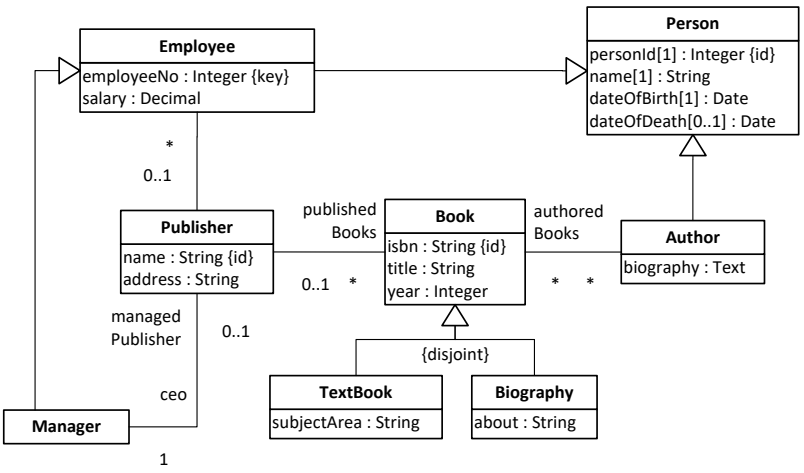
In the case of `Author`, we had declared the attribute `authorId` as a standard identifier. Assuming that this attribute represents a purely technical, rather than business, information item, we dropped it, since it's no longer needed as an identifier for authors. Consequently, we end up with a model which allows to identify employees either by their employee number or by their `personId` value, and to identify authors by their `personId` value.

We consider the following extension of our original example model, shown in Figure 1.4, where we have added two class hierarchies:

1. the disjoint (but incomplete) segmentation of `Book` into `TextBook` and `Biography`,

2. the overlapping and incomplete segmentation of `Person` into `Author` and `Employee`, which is further specialized by `Manager`.

Figure 1.4. The complete class model containing two inheritance hierarchies



3. Intension versus Extension

The **intension** of an object type is given by the set of its features, including attributes, associations, constraints and operations.

The **extension** of an object type is the set of all objects instantiating the object type. The extension of an object type is also called its *population*.

We have the following duality: while all features of a supertype are included in the intensions, or feature sets, of its subtypes (intensional inclusion), all instances of a subtype are included in the extensions, or instance sets, of its supertypes (extensional inclusion). This formal structure has been investigated in formal concept analysis [http://en.wikipedia.org/wiki/Formal_concept_analysis].

Due to the intension/extension duality we can specialize a given type in two different ways:

1. By **extending the type's intension** through adding features in the new subtype (such as adding the attribute `subjectArea` in the subtype `TextBook`).

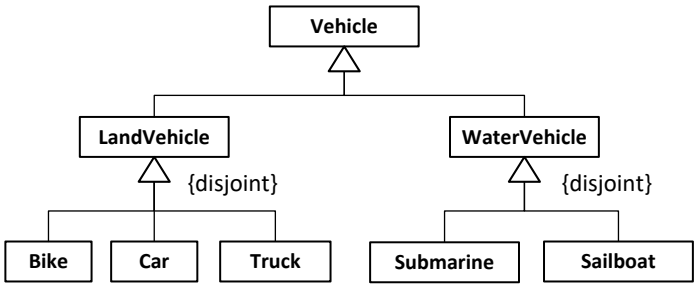
2. By **restricting the type's extension** through adding a constraint (such as defining a subtype `MathTextBook` as a `TextBook` where the attribute `subjectArea` has the specific value "Mathematics").

Typical OO programming languages, such as Java and C#, only support the first possibility (specializing a given type by extending its intension), while XML Schema and SQL99 also support the second possibility (specializing a given type by restricting its extension).

4. Type Hierarchies

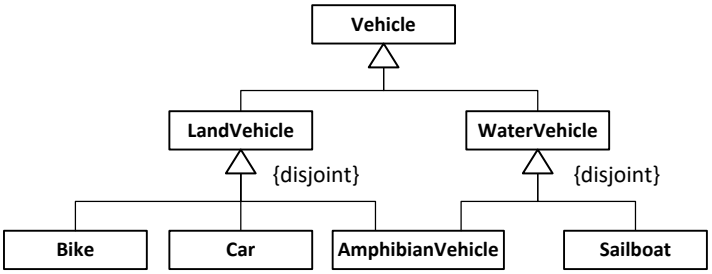
A *type hierarchy* (or *class hierarchy*) consists of two or more types, one of them being the root (or top-level) type, and all others having at least one direct supertype. When all non-root types have a unique direct supertype, the type hierarchy is a **single-inheritance hierarchy**, otherwise it's a **multiple-inheritance hierarchy**. For instance, in Figure 1.5 below, the class `Vehicle` is the root of a single-inheritance hierarchy, while Figure 1.6 shows an example of a multiple-inheritance hierarchy, due to the fact that `AmphibianVehicle` has two direct superclasses: `LandVehicle` and `WaterVehicle`.

Figure 1.5. A class hierarchy having the root class `Vehicle`



The simplest case of a class hierarchy, which has only one level of subtyping, is called a *generalization set* in UML, but may be more naturally called **segmentation**. A segmentation is **complete**, if the union of all subclass extensions is equal to the extension of the superclass (or, in other words, if all instances of the superclass instantiate some subclass). A segmentation is **disjoint**, if all subclasses are pairwise disjoint (or, in other words, if no instance of the superclass instantiates more than one subclass). Otherwise, it is called *overlapping*. A complete and disjoint segmentation is a **partition**.

Figure 1.6. A multiple inheritance hierarchy



In a class diagram, we can express these constraints by annotating the shared generalization arrow with the keywords *complete* and *disjoint* enclosed in braces. For instance, the annotation of a segmentation with {complete, disjoint} indicates that it is a partition. By default, whenever a segmentation does not have any annotation, like the segmentation of `Vehicle` into `LandVehicle` and `WaterVehicle` in Figure 1.6 above, it is {incomplete, overlapping}.

An information model may contain any number of class hierarchies.

5. The *Class Hierarchy Merge* Design Pattern

Consider the simple class hierarchy of the design model in Figure 1.1 above, showing a disjoint segmentation of the class `Book`. In such a case, whenever there is only one level (or there are only a few levels) of subtyping and each subtype has only one (or a few) additional properties, it's an option to re-factor the class model by merging all the additional properties of all subclasses into an expanded version of the root class such that these subclasses can be dropped from the model, leading to a simplified model.

This *Class Hierarchy Merge* design pattern comes in two forms. In its simplest form, the segmentations of the original class hierarchy are **disjoint**, which allows to use a single-valued `category` attribute for representing the specific category of each instance of the root class corresponding to the unique subclass instantiated by it. When the segmentations of the original class hierarchy are not disjoint, that is, when at least one of them is **overlapping**, we need to use a multi-valued `category` attribute for representing the set of types instantiated by an object. In this tutorial, we only discuss the simpler case of *Class Hierarchy Merge* re-factoring for disjoint segmentations, where we take the following re-factoring steps:

1. Add an **enumeration datatype** that contains a corresponding enumeration literal for each segment subclass. In our example, we add the enumeration datatype `BookCategoryEL`.

2. Add a `category` attribute to the root class with this enumeration as its range. The `category` attribute is mandatory [1], if the segmentation is complete, and optional [0..1], otherwise. In our example, we add a `category` attribute with range `BookCategoryEL` to the class `Book`. The `category` attribute is optional because the segmentation of `Book` into `TextBook` and `Biography` is incomplete.

3. Whenever the segmentation is **rigid** (does not allow *dynamic classification*), we designate the `category` attribute as **frozen**, which means that it can only be assigned once by setting its value when creating a new object, but it cannot be changed later.

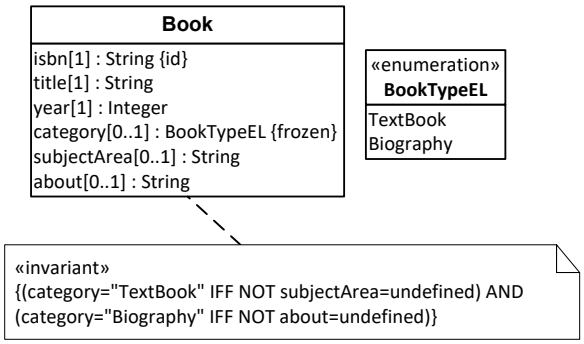
4. Move the properties of the segment subclasses to the root class, and make them **optional**. We call these properties, which are typically listed below the `category` attribute, **segment properties**. In our example, we move the attribute `subjectArea` from `TextBook` and `about` from `Biography` to `Book`, making them *optional*, that is [0..1].

5. Add a constraint (in an invariant box attached to the expanded root class rectangle) enforcing that the optional subclass properties have a value if and only if the instance of the root class instantiates the corresponding category. In our example, this means that an instance of `Book` is of category "TextBook" if and only if its attribute `subjectArea` has a value, and it is of category "Biography" if and only if its attribute `about` has a value.

6. Drop the segment subclasses from the model.

In the case of our example, the result of this design re-factoring is shown in Figure 1.7 below. Notice that the constraint (or "invariant") represents a logical sentence where the logical operator keyword "IFF" stands for the logical equivalence operator "if and only if" and the property condition `prop=undefined` tests if the property `prop` does not have a value.

Figure 1.7. The design model resulting from applying the Class Hierarchy Merge design pattern



6. Subtyping and Inheritance in Computational Languages

Subtyping and inheritance have been supported in *Object-Oriented Programming (OOP)*, in database languages (such as *SQL99*), in the XML schema definition language *XML Schema*, and in other computational languages, in various ways and to different degrees. At its core, subtyping in computational languages is about defining type hierarchies and the inheritance of features: mainly properties and methods in OOP; table columns and constraints in SQL99; elements, attributes and constraints in XML Schema.

In general, it is desirable to have support for **multiple classification** and **multiple inheritance** in type hierarchies. Both language features are closely related and are considered to be advanced features, which may not be needed in many applications or can be dealt with by using workarounds.

Multiple classification means that an object has more than one direct type. This is mainly the case when an object plays multiple roles at the same time, and therefore directly instantiates multiple classes defining these roles. Multiple inheritance is typically also related to role classes. For instance, a student assistant is a person playing both the role of a student and the role of an academic staff member, so a corresponding OOP class `StudentAssistant` inherits from both role classes `Student` and `AcademicStaffMember`. In a similar way, in our example model above, an `AmphibianVehicle` inherits from both role classes `LandVehicle` and `WaterVehicle`.

6.1. Subtyping and Inheritance with OOP Classes

The minimum level of support for subtyping in OOP, as provided, for instance, by Java and C#, allows defining inheritance of properties and methods in single-inheritance hierarchies, which can be inspected with the help of an **is-instance-of** predicate that allows testing if a class is the direct or an indirect type of an object. In addition, it is desirable to be able to inspect inheritance hierarchies with the help of

1. a predefined instance-level property for retrieving **the direct type of an object** (or its *direct types*, if multiple classification is allowed);

2. a predefined type-level property for retrieving **the direct supertype of a type** (or its *direct supertypes*, if multiple inheritance is allowed).

A special case of an OOP language is JavaScript, which does not (yet) have an explicit language element for classes, but only for constructors. Due to its dynamic programming features, JavaScript allows using

various code patterns for implementing classes, subtyping and inheritance (as we discuss in the next section on JavaScript).

6.2. Subtyping and Inheritance with Database Tables

A standard DBMS stores information (objects) in the rows of tables, which have been conceived as set-theoretic relations in classical *relational* database systems. The relational database language SQL is used for defining, populating, updating and querying such databases. But there are also simpler data storage techniques that allow to store data in the form of table rows, but do not support SQL. In particular, key-value storage systems, such as JavaScript's Local Storage API, allow storing a serialization of a **JSON table** (a map of records) as the string value associated with the table name as a key.

While in the classical, and still dominating, version of SQL (SQL92) there is no support for subtyping and inheritance, this has been changed in SQL99. However, the subtyping-related language elements of SQL99 have only been implemented in some DBMS, for instance in the open source DBMS *PostgreSQL*. As a consequence, for making a design model that can be implemented with various frameworks using various SQL DBMSs (including weaker technologies such as *MySQL* and *SQLite*), we cannot use the SQL99 features for subtyping, but have to model inheritance hierarchies in database design models by means of plain tables and foreign key dependencies. This mapping from class hierarchies to relational tables (and back) is the business of ***Object-Relational-Mapping*** frameworks such as Hibernate [http://en.wikipedia.org/wiki/Hibernate_%28Java%29] (or any other JPA [http://en.wikibooks.org/wiki/Java_Persistence/What_is_JPA%3F] Provider) or the Active Record [http://guides.rubyonrails.org/association_basics.html] approach of the Rails [http://rubyonrails.org/] framework.

There are essentially two alternative approaches how to represent a class hierarchy with relational tables:

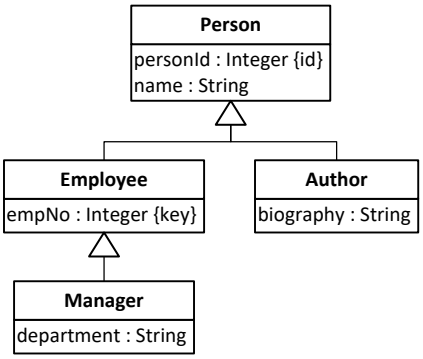
1. **Single Table Inheritance** [http://www.martinfowler.com/eaaCatalog/singleTableInheritance.html] is the simplest approach, where the entire class hierarchy is represented with a single table, containing columns for all attributes of the root class and of all its subclasses, and named after the name of the root class.

2. **Joined Tables Inheritance** [http://en.wikibooks.org/wiki/Java_Persistence/Inheritance#Joined.2C_Multiple_Table_Inheritance] is a more logical approach, where each subclass is represented by a corresponding subtable connected to the supertable via its primary key referencing the primary key of the supertable.

Notice that the *Single Table Inheritance* approach is closely related to the *Class Hierarchy Merge* design pattern discussed in Section 5 above. Whenever this design pattern has already been applied in the design model, or the design model has already been re-factored according to this design pattern, the class hierarchies concerned (their subclasses) have been eliminated in the design, and consequently also in the data model to be coded in the form of class definitions in the app's model layer, so there is no need anymore to map class hierarchies to database tables. Otherwise, when the *Class Hierarchy Merge* design pattern does not get applied, we would get a corresponding class hierarchy in the app's model layer, and we would have to map it to database tables with the help of the *Single Table Inheritance* approach.

We illustrate both the *Single Table Inheritance* approach and the *Joined Tables Inheritance* with the help of two simple examples. The first example is the `Book` class hierarchy, which is shown in Figure 1.1 above. The second example is the class hierarchy of the `Person` roles `Employee`, `Manager` and `Author`, shown in the class diagram in Figure 1.8 below.

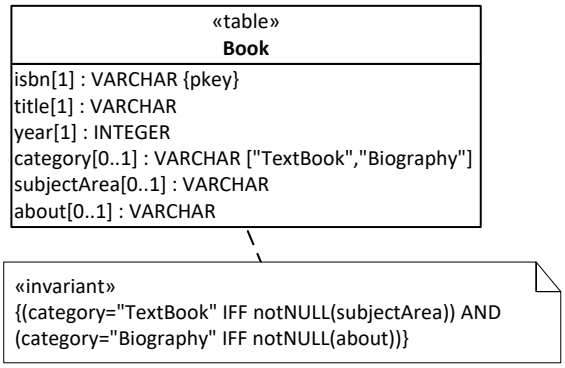
Figure 1.8. A class model with a `Person` roles hierarchy



6.2.1. Single Table Inheritance

Consider the single-level class hierarchy shown in Figure 1.1 above, which is an incomplete disjoint segmentation of the class `Book`, as the design for the model classes of an MVC app. In such a case, whenever we have a model class hierarchy with only one level (or only a few levels) of subtyping and each subtype has only one (or a few) additional properties, it's preferable to use *Single Table Inheritance*, so we model a single table containing columns for all attributes such that the columns representing additional attributes of subclasses are optional, as shown in the SQL table model in Figure 1.9 below.

Figure 1.9. An SQL table model with a single table representing the `Book` class hierarchy



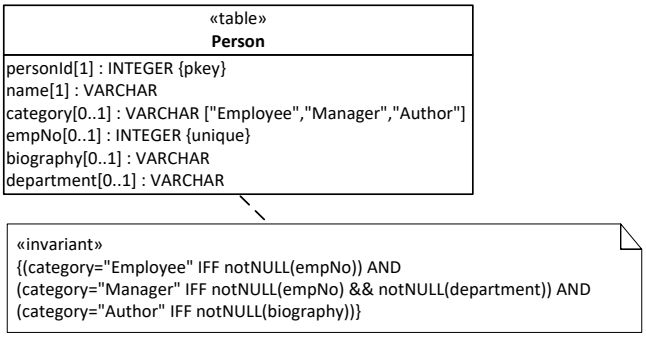
Notice that it is good practice to add a special *discriminator column* for representing the category of each row corresponding to the subclass instantiated by the represented object. Such a column would normally be string-valued, but constrained to one of the names of the subclasses. If the DBMS supports enumerations, it could also be enumeration-valued. We use the name `category` for the discriminator column.

Based on the `category` of a book, we have to enforce that if and only if it is "TextBook", its attribute `subjectArea` has a value, and if and only if it is "Biography", its attribute `about` has a value. This implied constraint is expressed in the invariant box attached to the `Book` table class in the class diagram above, where the logical operator keyword "IFF" represents the logical equivalence operator "if and only if". It needs to be implemented in the database, e.g., with an SQL table CHECK clause or with SQL triggers.

Consider the class hierarchy shown in Figure 1.8 above. With only three additional attributes defined in the subclasses `Employee`, `Manager` and `Author`, this class hierarchy could again be implemented

with the *Single Table Inheritance* approach. In the SQL table model, we can express this as shown in Figure 1.10 below.

Figure 1.10. An SQL table model with a single table representing the `Person` roles hierarchy

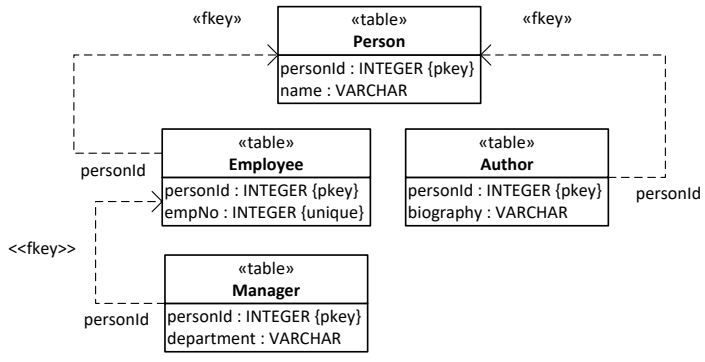


In the case of a multi-level class hierarchy where the subclasses have little in common, the *Single Table Inheritance* approach does not lead to a good representation.

6.2.2. Joined Tables Inheritance

In a more realistic model, the subclasses of `Person` shown in Figure 1.8 above would have many more attributes, so the *Single Table Inheritance* approach would be no longer feasible, and the *Joined Tables Inheritance* approach would be needed. In this approach we get the SQL data model shown in Figure 1.11 below. This SQL table model connects subtables to their supertables by defining their primary key attribute(s) to be at the same time a foreign key referencing their supertable. Notice that foreign keys are visuallized in the form of UML dependency arrows stereotyped with «fkey» and annotated at their source table side with the name of the foreign key column.

Figure 1.11. An SQL table model with the table `Person` as the root of a table hierarchy



The main disadvantage of the *Joined Tables Inheritance* approach is that for querying any subclass *join queries* are required, which may create a performance problem.

Chapter 2. Subtyping in a Java Back-End App

Whenever an app has to manage the data of a larger number of object types, there may be various **subtype** (inheritance) relationships between some of the object types. Handling subtype relationships is an advanced issue in software application engineering. It is often not well supported by application development frameworks.

In this chapter of our tutorial, we explain two case studies based on fragments of the information model of our running example, the *Public Library* app, shown above.

In the first case study, we consider the single-level class hierarchy with root `Book` shown in Figure 1.1, which is an incomplete disjoint segmentation. We use the *Single Table Inheritance* approach for mapping this class hierarchy to a single database table.

In the second case study, we consider the multi-level class hierarchy consisting of the `Person` roles `Employee`, `Manager` and `Author`, shown in Figure 1.8. We use the *Joined Table Inheritance* approach for mapping this class hierarchy to a set of database tables that are related with each other via foreign key dependencies.

In both cases we show

1. how to derive a Java Entity class model,

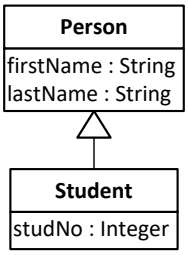
2. how to code the Java Entity class model in the form of Java Entity classes (as model classes),

3. how to write the view and controller code based on the model code.

1. Subtyping in Java

Java provides built-in support for subtyping with its `extends` keyword, but it does not support *multiple inheritance*. Consider the information design model shown in Figure 2.1 below.

Figure 2.1. `Student` is a subclass of `Person`



First we define the superclass `Person`. Then, we define the subclass `Student` and its subtype relationship to `Person` by means of the `extends` keyword:

```
public class Person {
  private String firstName;
  private String lastName;
```

```
  ...
}
public class Student extends Person {
  private int studentNo;

  public Student( String first, String last, int studNo) {
    super( firstName, lastName);
    this.setStudNo( studNo);
  }
  ...
}
```

Notice that in the `Student` class, we define a constructor with all the parameters required to create an instance of `Student`. In this subclass constructor we use `super` to invoke the constructor of the superclass `Person`.

2. Case Study 1: Eliminating a Class Hierarchy with *Single Table Inheritance*

In this example we implement the Book hierarchy shown in Figure 1.1. The Java *Entity class model* is derived from this design model.

2.1. Make the Java Entity class model

We make the *Java Entity class model* in 3 steps:

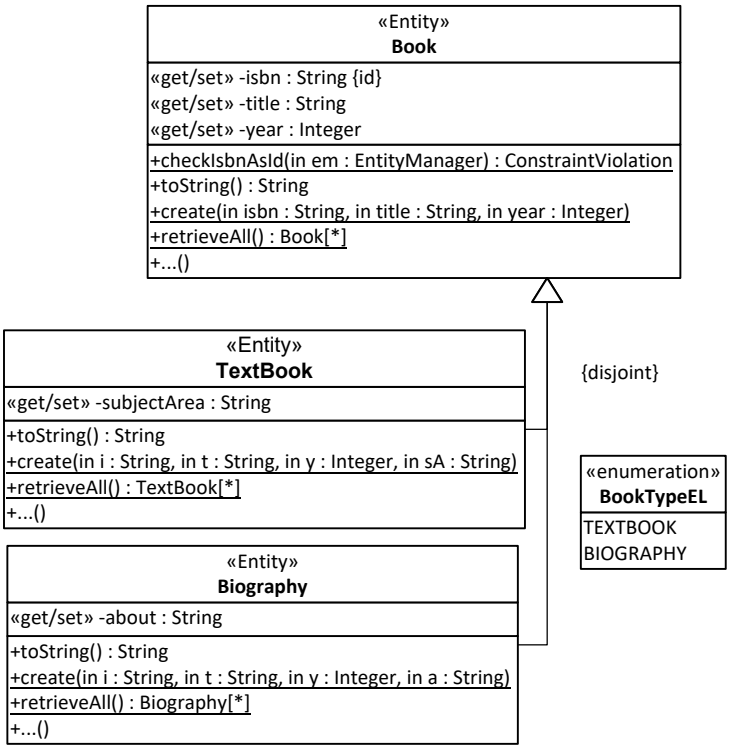
1. For every class in the model, we create a Java Entity class with the corresponding properties and add the **set** and **get** methods corresponding to each property.

2. Turn the plaform-independent datatypes (defined as the ranges of attributes) into Java datatypes.

3. Add the **set** and **get** methods corresponding to each direct property of every class in the hierarchy (subclasses must not define set and get method for properties in superclasses, but only for their direct properties).

This leads to the Java Entity class model shown in Figure 2.2.

Figure 2.2. The Java Entity class model



2.2. New issues

Compared to the validation app [ValidationApp/index.html] discussed in Part 2 of this tutorial, we have to deal with a number of new issues:

1. In the *model code* we have to take care of:

   a. Coding the enumeration type (`BookTypeEL` in our example) used in the views rendering to create a select list for the special case of book. Notice that this enumeration is not really used in the model classes (we discuss further the reasons), but only with the purpose to have the book category (i.e., special type) rendered in the facelets. Read Part 3 enumeration app [EnumerationApp/index.html] for detailed instructions on how to implement enumerations in Java.

   b. Code each class from the `Book` hierarchy using suitable JPA annotations for persistent storage in a corresponding database table, like `books`.

2. In the *UI code* we have to take care of:

   a. Adding a "Special type" column to the display table of the "List all books" use case in `WebContent/views/books/listAll.xhtml`. A book without a special category will have an empty table cell, while for all other books their category will be shown in this cell, along with other segment-specific attribute values.

   b. Adding a "Special type" select control, and corresponding form fields for all segment properties, in the forms of the "Create book" and "Update book" use cases in `WebContent/views/books/create.xhtml` and `WebContent/views/books/update.xhtml`. Segment property form fields are only displayed, and their validation is performed, when a corresponding book category has been selected. Such an approach of rendering specific form fields only on certain conditions is sometimes called "dynamic forms".

2.3. Code the classes of the Java Entity class model

The Java Entity class model can be directly coded for getting the code of the model classes of our Java back-end app.

2.3.1. Summary

1. Code the enumeration type (to be used in the facelets) .
2. Code the model classes and add the corresponding JPA annotations for class hierarchy.

These steps are discussed in more detail in the following sections.

2.3.2. Code the enumeration type `BookTypeEL`

The enumeration type `BookTypeEL` is coded as a Java enum, and we probide enumeration literals as well as assign label (human readable) to each enumeration literal:

```
public enum BookTypeEL {
  TEXTBOOK( "TextBook"), BIOGRAPHY( "Biography");
  private final String label;

  private BookTypeEL( String label) {
    this.label = label;
  }
  public String getLabel() {return this.label;}
}
```

2.3.2.1. Code the model classes

We code the model classes `Book`, `TextBook` and `Biography` in the form of Java Entity classes using suitable JPA annotations:

```
@Entity @Table( name="books")
@Inheritance( strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn( name="category",
    discriminatorType=DiscriminatorType.STRING, length=16)
@DiscriminatorValue( value = "BOOK")
@ManagedBean( name="book") @ViewScoped
public class Book {
  @Id @Column( length=10)
  @NotNull( message="An ISBN is required!")
  @Pattern( regexp="\\b\\d{9}(\\d|X)\\b", message="The ISBN must be a ...!")
  private String isbn;
  @Column( nullable=false) @NotNull( message="A title is required!")
  private String title;
  @Column( nullable=false) @NotNull( message="A year is required!")
  @UpToNextYear
  @Min( value=1459, message="The year must not be before 1459!")
  private Integer year;
  // constructors, set, get and other methods
  ...
```

```
}
```

When using the JPA Single Table Inheritance technique for a class hierarchy, the `@Inheritance` annotation must be set for the superclass (e.g, `Book` in our example). It provides the `strategy` parameter with the value `InheritanceType.SINGLE_TABLE`. The `@DiscriminatorColumn` annotation specifies the name of the table column storing the values that discriminate between different types of subclasses (`TextBook` and `Biography`, in our example). Unfortunately, we cannot use our `BookTypeEL` enumeration for defining the values of the DiscriminatorColumn `name` parameter, because expressions like `BookCategoryEL.TEXTBOOK.name()` are not constant, so a Java compiler exception is thrown. The `@DiscriminatorValue` annotation specifies a unique value to be stored in the discriminator column.

Further, we define the subtypes `TextBook` and `Biography`:

```
@Entity @Table( name="textbooks")
@DiscriminatorValue( value="TEXTBOOK")
@ManagedBean( name="textBook") @ViewScoped
public class TextBook extends Book {
  @Column( nullable=false) @NotNull( message="A subject area value is required"
  String subjectArea;
  // constructors, set, get and other methods
  ...
}
```

`TextBook` and `Biography` are subclasses of `Book`, so we define them with `extends Book`. For each subclass, we add a `@DiscriminatorValue` annotation with a value from the `BookTypeEL` enumeration.

2.4. Database schema for single table class hierarchy

As a result of the `@Inheritance( strategy=InheritanceType.SINGLE_TABLE)` annotation, only one database table is used for representing a class hierarchy. This table contains the columns corresponding to all the properties of all the classes from the hierarchy plus the discriminator column. In our example, the `books` table contains the following columns: `isbn`, `title`, `year`, `subjectArea`, `about` and `category`. The simplified corresponding SQL-DDL code internally used to generate the `books` table for our application is shown below:

```
CREATE TABLE IF NOT EXISTS `books` (
  `ISBN` varchar(10) NOT NULL,
  `TITLE` varchar(255) NOT NULL,
  `YEAR` int(11) NOT NULL,
  `SUBJECTAREA` varchar(255) DEFAULT NULL,
  `ABOUT` varchar(255) DEFAULT NULL,
  `category` varchar(16) DEFAULT NULL,
)
```

2.5. Write the View and Controller Code

The user interface (UI) consists of a start page that allows navigating to the data management pages (in our example, to `WebContent/views/books/index.xhtml`). Such a data management page contains 4 sections: *list books*, *create book*, *update book* and *delete book*.

2.5.1. Summary

We have to take care of handling the category discriminator and the `subjectArea` and `about` segment properties both in the "List all books" use case as well as in the "Create book" and "Update book" use cases by

1. Adding a segment information column ("Special type") to the display table of the "List all books" use case in `WebContent/views/books/listAll.xhtml`.

2. Adding a "Special type" select control, and corresponding form fields for all segment properties, in the forms of the "Create book" and "Update book" use cases in `WebContent/views/books/create.xhtml` and `WebContent/views/books/create.xhtml`. Segment property form fields are only displayed, and their validation occurs only when a corresponding book category has been selected.

2.5.2. Adding a segment information column in "List all books"

We add a "Special type" column to the display table of the "List all books" use case in `books.html`:

```
<h:dataTable value="#{bookController.books}" var="b">
  ...
  <h:column>
    <f:facet name="header">Special type</f:facet>
    #{b.getClass().getSimpleName() != 'Book' ? b.getClass().getSimpleName() : '
  </h:column>
</h:dataTable>
```

A conditional expression is used to check if the Java bean class name is `Book`, in which case we don't show it, or if it is something else (e.g. `TextBook` or `Biography`), and then it is shown. This expression also shows how you can call/use various Java bean methods, not only custom methods.

2.5.3. Adding a "Special type" select control in "Create book" and "Update book"

In both use cases, we need to allow selecting a special category of book ('textbook' or 'biography') with the help of a select control, as shown in the following HTML fragment:

```
<h:panelGrid id="bookPanel" columns="3">
  ...
  <h:outputText value="Special type: " />
  <h:selectOneMenu id="bookType" value="#{viewScope.bookTypeVal}">
    <f:selectItem itemLabel="---"/>
    <f:selectItems value="#{book.typeItems}" />
    <f:ajax event="change" execute="@this"
            render="textBookPanel biographyBookPanel standardBookPanel" />
  </h:selectOneMenu>
  <h:message for="bookType" errorClass="error" />
</h:panelGrid>

<h:panelGroup id="standardBookPanel">
```

```
  <h:commandButton value="Create" rendered="#{viewScope.bookTypeVal == null}"
                   action="#{bookController.add( book.isbn, book.title, book.yea
</h:panelGroup>

<h:panelGroup id="textBookPanel">
  <h:panelGrid rendered="#{viewScope.bookTypeVal == 'TEXTBOOK'}" columns="3">
    <h:outputText value="Subject area: " />
    <h:inputText id="subjectArea" value="#{textBook.subjectArea}"/>
    <h:message for="subjectArea" errorClass="error" />
  </h:panelGrid>
  <h:commandButton value="Create" rendered="#{viewScope.bookTypeVal == 'TEXTBOOK
                   action="#{textBookController.add( book.isbn, book.title, book
</h:panelGroup>

<h:panelGroup id="biographyBookPanel">
  <h:panelGrid rendered="#{viewScope.bookTypeVal == 'BIOGRAPHY'}" columns="3">
    <h:outputText value="About: " />
    <h:inputText id="about" value="#{biographyBook.about}"/>
    <h:message for="about" errorClass="error" />
  </h:panelGrid>
  <h:commandButton value="Create" rendered="#{viewScope.bookTypeVal == 'BIOGRAPH
                   action="#{biographyBookController.add( book.isbn, book.title,
</h:panelGroup>
```

There are a few important remarks on the above view code:

- the `h:selectOneMenu` is used to create a single selection list which is populated with book tiutles by using the `getTypeItems` method of the `Book` class. A more detailed explanation is presented in Part 3 enumeration app [EnumerationApp/index.html].

- it is possible to conditionally render facelet components by using the `rendered` attribute. The JSF EL expression must return `true` or `false`, this making the HTML resulting elements to be part of the HTML DOM or not. Notice that the conditional expressions are evaluated in the server side. This is the method we use to hide or show the input form elements corresponding to various book types (e.g., `TextBook` has a `subjectArea` property while Biography has an `about` property).

- the `render` attribute used with `f:ajax` specifies which of the JSF components are to be updated . This is needed because of the live DOM changes (client side, not server side) which applies after the AJAX call.

- AJAX is used to submit form for reevaluation when the special type select list is changed (something is selected). As a result, it enforces the rendering of the three panels corresponding to three book cases: simple book, text book and biography. Using `execute="@this"` we enforce the re-evaluation of the form at server side, so the resulting HTML DOM structure contains the changes accordiong with the conditions specified by the `rendered` attributes of the various JSF elements. Notice that an JSF element which has a conditional `rendered` expression must be child of another JSF element which is always part of the DOM.

- `h:panelGroup` is used to define a set of elements which are shown or hidden.

- the `action` attribute of the `h:commandButton` can't be used with conditional expressions, therefore we have to create three command buttons, one for each case: create/update a `Book`, a `TextBook` or a `Biography`. The `add` method of the corresponding controller class (i.e., `BookController`, `TextBookController` or `BiographyController` is called).

- since we do not have a corresponding property in the Java bean class(es) for the special type (category), we can use JSF variables to store the value of the single select list, and then use the variable in `rendered` conditions for various elements. Therefore, for the `h:selectOneMenu`, the `value="#{viewScope.bookTypeVal}"` specifies that we use a "on the fly" defined property named `bookTypeVal`, which is part of the view scope internal JSF object(s). We can also define such variable outside the view scope, e.g., `value="#{bookTypeVal}"`, but in this case they are request scoped, so their value is lost after submitting the form, and the `rendered` conditions can't be correctly evaluated.

For a class in the class hierarchy, one corresponding controller class is defined. It contains the specific `add`, `update` and `destroy` methods (see . The shared methods, such as `getAllObjects` are defined by the controller of the top level class (e.g., for our example, this is `BookController`). See minimal app [MinimalApp/index.html] for more details on how to implement the controller class and the corresponding CRUD methods.

The *Update Book* test case is very similar with the *Create Book* test case. The *Delete Book* test case remains unchanged, see See minimal app [MinimalApp/index.html] for more details.

3. Case Study 2: Implementing a Class Hierarchy with *Joined Table Inheritance*

The starting point for our case study is the design model shown in Figure 1.8 above. In the following sections, we show how to eliminate the `Manager` class by using the *Class Hierarchy Merge* design pattern and how to implement the `Person` hierarchy and use *Joined, Multiple Table Inheritance* storage with the help of JPA framework.

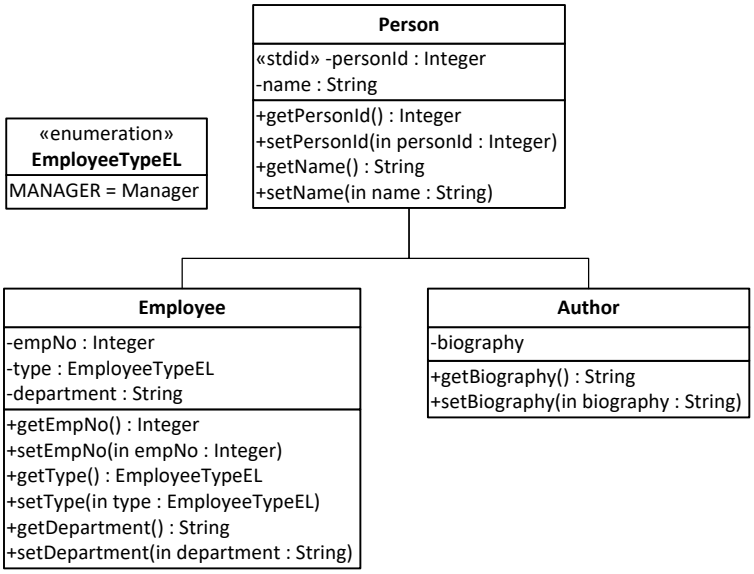
3.1. Make the Java Entity class model

We design the *model classes* of our example app with the help of a Java *Entity class model* that we derive from the *design model* by essentially leaving the generalizatiion arrows as they are and just adding *getters* and *setters* to each class. However, in the case of our example app, it is natural to apply the *Class Hierarchy Merge* design pattern to the segmentation of `Employee` for simplifying the data model by eliminating the `Manager` subclass. This leads to the model shown in Figure 2.3 below. Notice that we have also made two technical design decisions:

1. We have declared the segmentation of `Person` into `Employee` and `Author` to be **complete**, that is, any person is an employee or an author (or both).

2. We have turned `Person` into an **abstract class** (indicated by its name written in italics in the class rectangle), which means that it cannot have direct instances, but only indirect ones via its subclasses `Employee` and `Author`. This technical design decision is compatible with the fact that any `Person` is an `Employee` or an `Author` (or both), and consequently there is no need for any object to instantiate `Person` directly.

Figure 2.3. The Java Entity class model of the Person class hierarchy



3.2. New issues

Compared to the model of our first case study, shown in Figure 2.2 above, we have to define the category relationships between `Employee` and `Person`, as well as between `Author` and `Person`, using the JPA annotation.

In the *UI code* we have to take care of:

1. Adding the views (in the folders `WebContent/views/authors` and `WebContent/views/employees`) and controller classes (`AuthorController` and `EmployeeController`) for the corresponding `Author` and `Employee` model classes.

2. Deal with the `Manager` case, by adding a "Special type" select control, in the forms of the "Create book" and "Update book" use cases in `WebContent/views/books/create.xhtml` and `WebContent/views/books/update.xhtml`. Segment property form fields (i.e., department in our example) are only displayed, and their validation is performed, when a corresponding employee type has been selected.

3.3. Code the model classes of the Java Entity class model

The Java Entity class model shown in Figure 2.3 above is coded by using the JavaBeans `Person`, `Employee` and `Author` as well as for the enmueration type `EmployeeTypeEL`.

3.3.1. Define the category relationships

We define the category relationships between `Employee` and `Person`, as well as between `Author` and `Person`, using the JPA annotations. At first we create the `Person` class as shown below:

```
@Entity
@Inheritance( strategy = InheritanceType.JOINED)
@DiscriminatorColumn( name = "category", discriminatorType = DiscriminatorType.S
@Table( name = "persons")
```

```
public abstract class Person {
  @Id
  @PositiveInteger
  @NotNull( message = "A person ID value is required!")
  private Integer personId;
  @Column( nullable = false)
  @NotNull( message = "A name is required!")
  private String name;

  // constructors, set, get and other methods
}
```

Comparing with the `Book` hierarchy shown in *Test Case 1*, the `@Inheritance` annotations defines now the `strategy = InheritanceType.JOINED`. This means, for every class in the inheritance hierarchy, a database table is used. The `@DiscriminatorColumn( name = "category")` specifies the column in the corresponding table (i.e., `persons`) of the top hierarchy class (i.e., `Person`) which stores the discriminator values used to identify the stored type of each entry (table row).

Notice that the Java class `Person` is declared as being `abstract`, which means it can't be initialized, instead we can and we initialize subclasses derived from it (i.e., `Employee` and `Author`). This also mean that we don't declare a `@DiscriminatorValue` because no direct instance of `Person` is stored in the database table.

Further, we define the `Author` class as follows:

```
@Entity
@DiscriminatorValue( value = "AUTHOR")
@Table( name = "authors")
@ManagedBean( name = "author")
@ViewScoped
public class Author extends Person {
  @NotNull( message = "A biography is required!")
  private String biography;

  // constructors, set, get and other methods
}
```

The `Author` class inherits `Person`, therefore the get and set methods corresponding to `personId` and `name` properties are available. The `@DiscriminatorValue( value = "AUTHOR")` specifies that the column category of the `persons` table stores the value `AUTHOR` for every entry which comes from persisting an `Author` instance.

Last we define the `Employee` class:

```
@Entity
@DiscriminatorValue( value = "EMPLOYEE")
@Table( name = "employees")
@ManagedBean( name = "employee")
@ViewScoped
public class Employee extends Person {
  @Column( nullable = false)
  @NotNull( message = "An employee ID is required!")
  @Min( value = 20000, message = "Employee no. must be greater than 20000!")
  @Max( value = 99999, message = "Employee no. must be lower than 100000!")
```

```
  private Integer empNo;
  @Column( nullable = false, length = 32)
  @Enumerated( EnumType.STRING)
  private EmployeeTypeEL type;
  @Column( nullable = true, length = 64)
  private String department;

  // constructors, set, get and other methods
}
```

Notice the `type` property used to identify the `Employee` type, such as `Manager`. Its values are defined by the `EmployeeTypeEL` enumeration.

3.3.2. Database schema for joined table class hierarchy

As a result of the `@Inheritance( strategy = InheritanceType.JOINED)` annotation, for each class in the inheritance hierarchy, one database table is created. The corresponding simplified SQL-DDL scripts used by JPA to create the `persons`, `authors` and `employees` tables are shown below:

```
CREATE TABLE IF NOT EXISTS `persons` (
  `PERSONID` int(11) NOT NULL,
  `category` varchar(16) DEFAULT NULL,
  `NAME` varchar(255) NOT NULL
);

CREATE TABLE IF NOT EXISTS `authors` (
  `PERSONID` int(11) NOT NULL,
  `BIOGRAPHY` varchar(255) DEFAULT NULL
);
ADD CONSTRAINT `FK_authors_PERSONID` FOREIGN KEY (`PERSONID`) REFERENCES `perso

CREATE TABLE IF NOT EXISTS `employees` (
  `PERSONID` int(11) NOT NULL,
  `DEPARTMENT` varchar(64) DEFAULT NULL,
  `EMPNO` int(11) NOT NULL,
  `TYPE` varchar(32) DEFAULT NULL
);
ADD CONSTRAINT `FK_employees_PERSONID` FOREIGN KEY (`PERSONID`) REFERENCES `pers
```

As we can see, every table contains the direct properties as defined by the corresponding Java bean class. Additionally, the `authors` and `employees` tables are created with a foreign key constraing for the `PERSONID` column refering to to the `PERSONID` column from the `persons` table.

3.4. Write the View and Controller Code

The user interface (UI) is very similar with the one for the Book hierarchy shown earlier in this tutorial. For every Java bean class, we have a controller class which contains the add, update and destroy CRUD methods. The `PersonController` class is defined as abstract and contains the `checkPersonIdAsId` method, which is common to all subclasses. The `AuthorController` and `EmployeeController` inherits the `PersonController`.

For every non-abstract entity class in the inheritance hierarchy we define a set of views corresponding to CRUD operations. For example, in the case of `Author` we have `WebContent/views/authors/`

`{listAll, create, update, destroy}.xhtml` files. In the case of `Employee`, the *List All Employees* test case require to display the *Special type of employee* column:

```
<h:column>
  <f:facet name="header">Special type of employee</f:facet>
  #{e.type != null ? e.type.label.concat( " of ").concat( e.department).concat(
</h:column>
```

It is interesting to notice that within JSF expressions we can' use the + (plus) operator to concatenate Java strings. JSF EL expression allows the `+` operator to be used only with number types. However, we can use the `concat` method available to any `String` object.

The *Create*, *Update* and *Delete* test cases for both cases, `Author` and `Employee` are similar whith what we have learned in this tutorial as well as in the Part 1 to 5 tutorials.

4. Run the App and Get the Code

Running your application is simple. First stop (if already started, otherwise skip this part) your Tomcat/TomEE server by using `bin/shutdown.bat` for Windows OS or `bin/shutdown.sh` for Linux. Next, download and unzip the ZIP archive file [SubtypingApp.zip] containing all the source code of the application and also the ANT script file which you have to edit and modify the `server.folder` property value. Now, execute the following command in your console or terminal: `ant deploy -Dappname=subtypingapp`. Last, start your Tomcat web server (by using `bin/startup.bat` for Windows OS or `bin/startup.sh` for Linux). Please be patient, this can take some time depending on the speed of your computer. It will be ready when the console display the following info: `INFO: Initializing Mojarra [some library versions and paths are shonw here] for context '/subtypingapp'`. Finally, open a web browser and type: `http://localhost:8080/subtypingapp/faces/views/app/index.xhtml`

You may want to download the ZIP archive [lib.jar] containing all the dependency libaries, or run the subtyping app [http://yew.informatik.tu-cottbus.de/tutorials/java/subtypingapp/] directly from our server.