JavaScript Front-End Web App Tutorial Part 6: Inheritance in Class Hierarchies

An advanced tutorial about developing front-end web applications with class hierarchies, using plain JavaScript

Gerd Wagner <G.Wagner@b-tu.de>

JavaScript Front-End Web App Tutorial Part 6: Inheritance in Class Hierarchies: An advanced tutorial about developing front-end web applications with class hierarchies, using plain JavaScript

by Gerd Wagner

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 [SubtypingApp/index.html] 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 2015-11-12

Table of Contents

List of Figures

Foreword

This tutorial is Part 6 of our series of six tutorials [http://web-engineering.info/JsFrontendApp] about model-based development of front-end web applications with plain JavaScript. 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**, such as the 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**, such as the associations between books and publishers and between books and authors, also assigning books to authors and to publishers.

You may also want to take a look at our open access book Building Front-End Web Apps with Plain JavaScript [http://web-engineering.info/JsFrontendApp-Book], which includes all parts of the tutorial in one document, dealing with multiple object types ("books", "publishers" and "authors") and taking care of constraint validation, associations and subtypes/inheritance.

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.

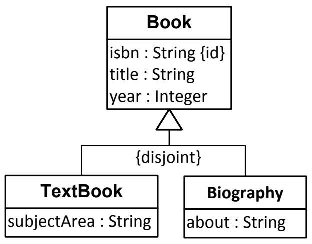
A subtype **inherits** all features from its supertypes. When a subtype inherits attributes, associations and constraints from a supertype, this means that these features need not be explicitly rendered for the subtype in the class diagram, but the reader of the diagram has to know that all features 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` is specialized by two subtypes: `TextBook` and `Biography`



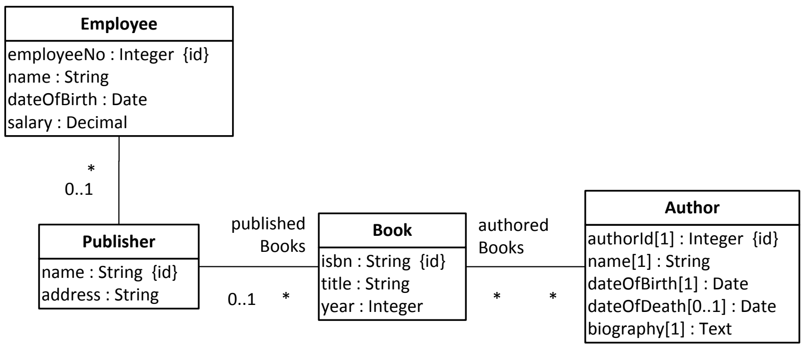
When specializing an object type, we define additional features for the newly added subtype. In many cases, these additional features are more specific properties. For instance, in the case of `TextBook` specializing `Book`, we define the additional attribute `subjectArea`. In some programming languages, 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

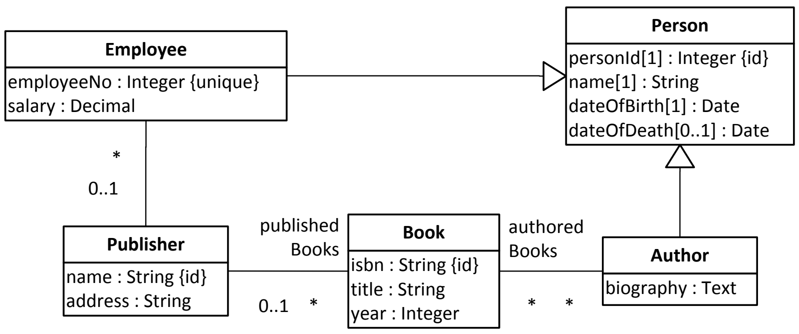
We illustrate generalization with the following example, which extends the information model of Part 4 by adding the object type `Employee` and associating employees with publishers.

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`



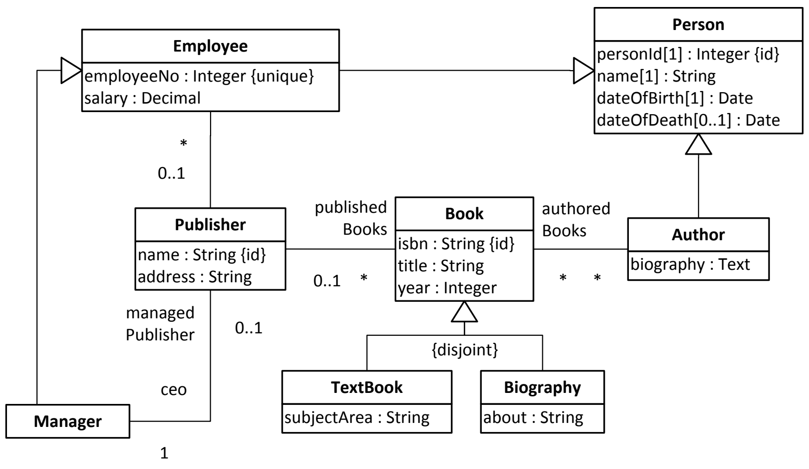
When generalizing two or more object types, we move (and centralize) a set of features shared by them in the newly added supertype. In the case of `Employee` and `Author`, this set of shared features consists of the attributes `name`, `dateOfBirth` and `dateOfDeath`. In general, shared features may include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

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:

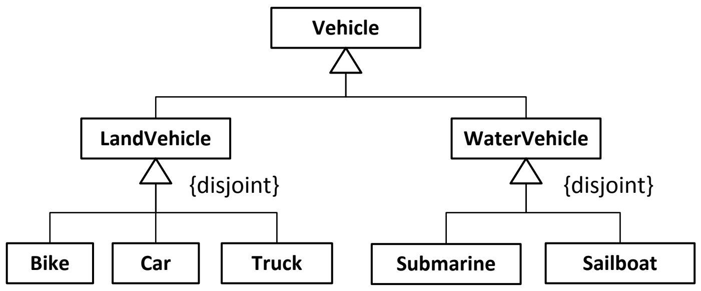
1. By **extending the type's intension** through adding features in the new subtype (such as adding the attribute `subjectArea` in the subtype `TextBook`).
2. By **restricting the type's extension** through adding a constraint (such as defining a subtype `MathTextBook` as a `TextBook` where the attribute `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

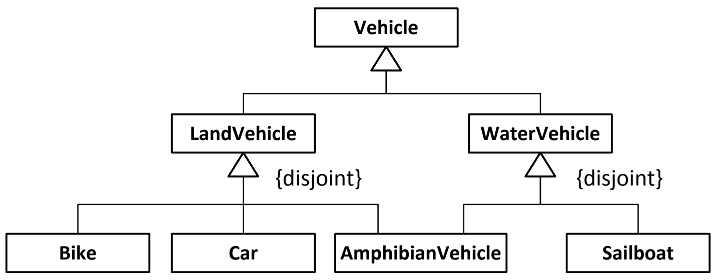
A *type hierarchy* (or *class hierarchy*) consists of two or more types, one of them being the root (or top-level) type, and all others having at least one direct supertype. When all non-root types have a unique direct supertype, the type hierarchy is a **single-inheritance hierarchy**, otherwise it's 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

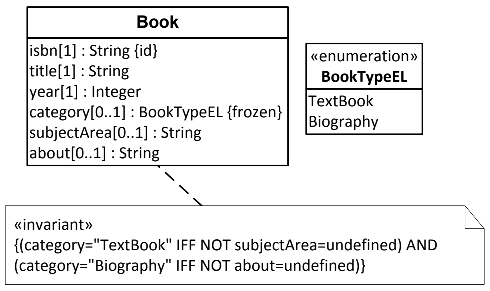
Consider the simple class hierarchy of the design model in Figure 1.1 above, showing a disjoint segmentation of the class `Book`. In such a case, whenever there is only one level (or there are only a few levels) of subtyping and each subtype has only one (or a few) additional properties, it's an option 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


Book
isbn[1] : String {id}
title[1] : String
year[1] : Integer
category[0..1] : BookTypeEL {frozen}
subjectArea[0..1] : String
about[0..1] : String
«enumeration»
BookTypeEL
TextBook
Biography
«invariant»
{(category="TextBook" IFF NOT subjectArea=undefined) AND
(category="Biography" IFF NOT about=undefined)}

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

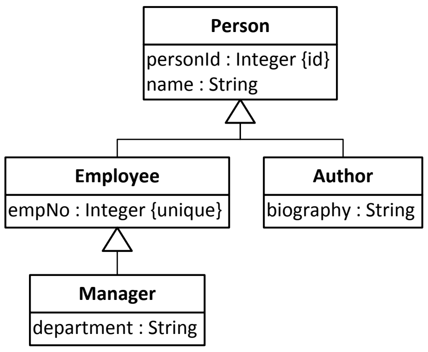
1. **Single Table Inheritance** [http://www.martinfowler.com/eaaCatalog/singleTableInheritance.html] is the simplest approach, where the entire class hierarchy is represented with a single table, containing columns for all attributes of the root class and of all its subclasses, and named after the name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

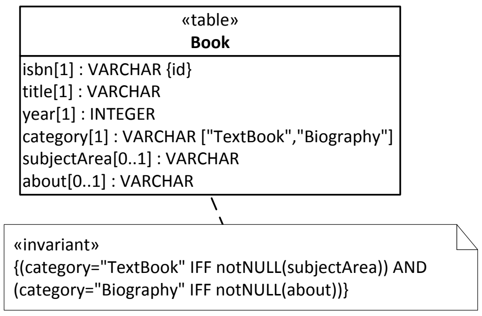
Figure 1.8. A class model with a `Person` roles hierarchy



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

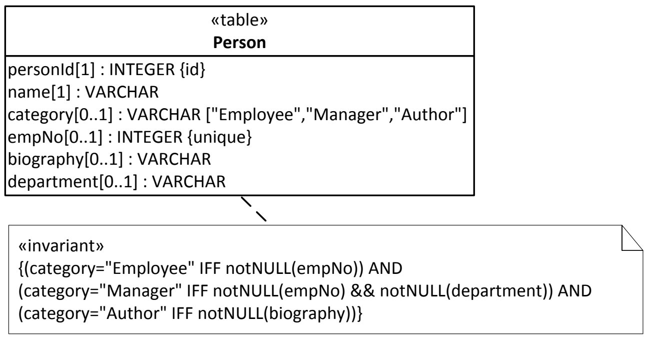

«table»
Book
isbn[1] : VARCHAR {id}
title[1] : VARCHAR
year[1] : INTEGER
category[1] : VARCHAR ["TextBook","Biography"]
subjectArea[0..1] : VARCHAR
about[0..1] : VARCHAR
«invariant»
{(category="TextBook" IFF notNULL(subjectArea)) AND
(category="Biography" IFF notNULL(about))}

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

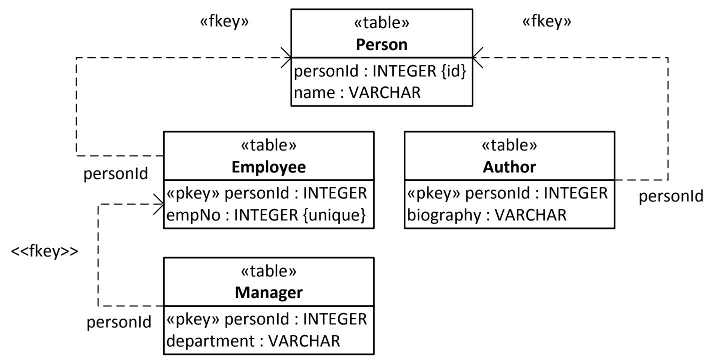

«table»
Person
personId[1] : INTEGER {id}
name[1] : VARCHAR
category[0..1] : VARCHAR ["Employee","Manager","Author"]
empNo[0..1] : INTEGER {unique}
biography[0..1] : VARCHAR
department[0..1] : VARCHAR
«invariant»
{(category="Employee" IFF notNULL(empNo)) AND
(category="Manager" IFF notNULL(empNo) && notNULL(department)) AND
(category="Author" IFF notNULL(biography))}

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.

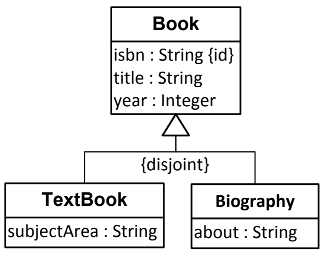
Chapter 2. Subtyping in a Plain JavaScript Front-End App

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 first explain the general approach to constructor-based subtyping in JavaScript before presenting two case studies based on fragments of the information model of our running example, the *Public Library* app, shown above.

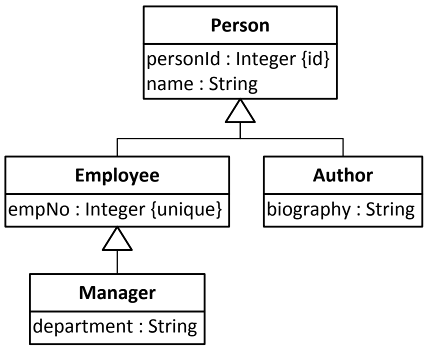
In the first case study, we consider the single-level class hierarchy with root `Book` shown in Figure 2.1 below (which is an incomplete disjoint segmentation). We use the *Single Table Inheritance* approach for re-factoring this class hierarchy to a single class that is mapped to a persistent database table stored with JavaScript's *Local Storage*.

Figure 2.1. The object type `Book` as the root of a disjoint segmentation



In the second case study, we consider the multi-level class hierarchy consisting of the `Person` roles `Employee`, `Manager` and `Author`, shown in Figure 2.2 below. We use the *Joined Tables Inheritance* approach for mapping this class hierarchy to a set of database tables that are related with each other via foreign key dependencies.

Figure 2.2. The `Person` roles hierarchy



In both cases we show

1. how to derive a JavaScript data model, and a corresponding entity table model, from the class hierarchy (representing an information design model),
2. how to encode the JavaScript data model in the form of JavaScript model classes,
3. how to write the view and controller code based on the model code.

1. Subtyping with Constructor-Based Classes

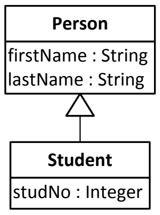
Since JavaScript does not have an explicit class concept, subtyping is not directly supported, but certain forms of subtyping can be implemented with the help of certain code patterns. Any subtyping code pattern should provide two inheritance mechanisms: (1) inheritance of properties and (2) inheritance of methods.

As we have explained in Part 1 of this tutorial [minimal-tutorial.html] , classes can be defined in two alternative ways: **constructor-based** and **factory-based**. Both approaches have their own way of implementing inheritance. In this part of our tutorial we only discuss subtyping and inheritance for constructor-based classes, while in the book Building Front-End Web Apps with Plain JavaScript [http://web-engineering.info/JsFrontendApp-Book] we also discuss subtyping and inheritance for factory-based classes

We summarize the 3-part code pattern for defining a superclass and a subclass In a constructor-based class hierarchy with the help of an example, illustrated in Figure 2.3 below:

```
// (1) Define superclass
function Person( first, last) {
  this.firstName = first;
  this.lastName = last;
}
// (2) Define subclass
function Student( first, last, studNo) {
  // invoke superclass constructor
  Person.call( this, first, last);
  // define and assign additional properties
  this.studNo = studNo;
}
// (3) Inherit methods from superclass
Student.prototype = Object.create( Person.prototype);
// adjust the subtype's constructor property
Student.prototype.constructor = Student;
```

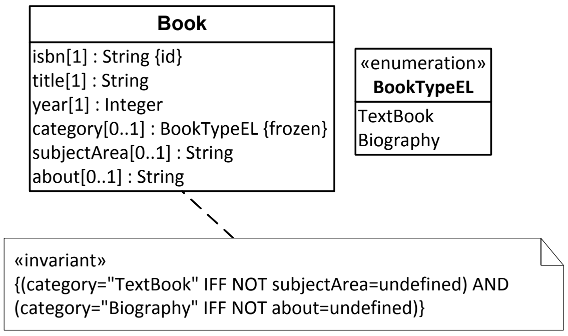
Figure 2.3. `Student` is a subclass of `Person`



2. Case Study 1: Eliminating a Class Hierarchy

Simple class hierarchies can be eliminated by applying the *Class Hierarchy Merge* design pattern. The starting point for our case study is the simple class hierarchy shown in the information design model of Figure 2.1 above, representing a disjoint (but incomplete) segmentation of `Book` into `TextBook` and `Biography`. This model is first simplified by applying the *Class Hierarchy Merge* design pattern, resulting in the model shown in Figure 2.4.

Figure 2.4. The simplified information design model obtained by applying the Class Hierarchy Merge design pattern


Book
isbn[1] : String {id}
title[1] : String
year[1] : Integer
category[0..1] : BookTypeEL {frozen}
subjectArea[0..1] : String
about[0..1] : String
«enumeration»
BookTypeEL
TextBook
Biography
«invariant»
{(category="TextBook" IFF NOT subjectArea=undefined) AND
(category="Biography" IFF NOT about=undefined)}

We can now derive a *JavaScript data model* from this design model.

2.1. Make the JavaScript data model

We make the *JavaScript data model* in 3 steps:

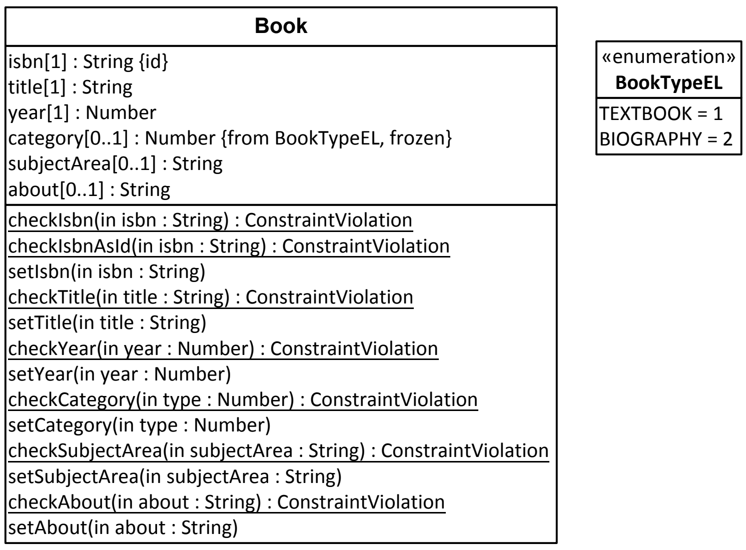
1. Turn the design model's **enumeration type**, which contains an enumeration literal for each segment subclass, into a corresponding JavaScript map where the enumeration literals are (by convention uppercase) keys associated with an integer value that enumerates the literal. For instance, for the first enumeration literal "TextBook" we get the key-value pair TEXTBOOK=1.

2. Turn the platform-independent datatypes (defined as the ranges of attributes) into JavaScript datatypes. This includes the case of enumeration-valued attributes, such as `category`, which are turned into numeric attributes restricted to the enumeration integers of the underlying enumeration type.

3. Add property **checks** and **setters**, as described in Part 2 of this tutorial. The `checkCategory` and `setCategory` methods, as well as the checks and setters of the segment properties need special consideration according to their implied semantics. In particular, a segment property's check and setter methods must ensure that the property can only be assigned if the `category` attribute has a value representing the corresponding segment. We explain this implied validation semantics in more detail below when we discuss how the JavaScript data model is encoded.

This leads to the JavaScript data model shown in Figure 2.5, where the class-level ('static') methods are underlined:

Figure 2.5. The JavaScript data model


Book
isbn[1] : String {id}
title[1] : String
year[1] : Number
category[0..1] : Number {from BookTypeEL, frozen}
subjectArea[0..1] : String
about[0..1] : String
checkIsbn(in isbn : String) : ConstraintViolation
checkIsbnAsId(in isbn : String) : ConstraintViolation
setIsbn(in isbn : String)
checkTitle(in title : String) : ConstraintViolation
setTitle(in title : String)
checkYear(in year : Number) : ConstraintViolation
setYear(in year : Number)
checkCategory(in type : Number) : ConstraintViolation
setCategory(in type : Number)
checkSubjectArea(in subjectArea : String) : ConstraintViolation
setSubjectArea(in subjectArea : String)
checkAbout(in about : String) : ConstraintViolation
setAbout(in about : String)
«enumeration»
BookTypeEL
TEXTBOOK = 1
BIOGRAPHY = 2

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. Adding the constraint violation class *FrozenValueConstraintViolation* to `errorTypes.js`.

   b. Encoding the enumeration type to be used as the range of the `category` attribute (`BookCategoryEL` in our example).

   c. Encoding the `checkCategory` and `setCategory` methods for the `category` attribute. In our example this attribute is *optional*, due to the fact that the book types segmentation is *incomplete*. If the segmentation, to which the *Class Hierarchy Merge* pattern is applied, is complete, then the `category` attribute is *mandatory*.

   d. Encoding the *checks* and *setters* for all segment properties such that the check methods take the category as a second parameter for being able to test if the segment property concerned applies to a given instance.

   e. Refining the serialization method `toString()` by adding a `category` case distinction (`switch`) statement for serializing only the segment properties that apply to the given category.

   f. Implementing the *Frozen Value Constraint* for the `category` attribute in `Book.update` by updating the `category` of a book only if it has not yet been defined. This means it cannot be updated anymore as soon as it has been defined.

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 `books.html`. 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. This requires a corresponding switch statement in `pl.view.books.list.setupUserInterface` in the `books.js` view code file.

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 `books.html`. Segment property form fields are only displayed, and their validation event handlers set, 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. Encode the model classes of the JavaScript data model

The JavaScript data model can be directly encoded for getting the code of the model classes of our JavaScript front-end app.

2.3.1. Summary

1. Encode the enumeration type (to be used as the range of the `category` attribute) as a special JavaScript object mapping upper-case keys, representing enumeration literals, to corresponding enumeration integers.

2. Encode the model class (obtained by applying the *Class Hierarchy Merge* pattern) in the form of a JavaScript constructor function with class-level check methods attached to it, and with instance-level setter methods attached to its `prototype`.

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

2.3.2. Encode the enumeration type `BookCategoryEL`

The enumeration type `BookCategoryEL` is encoded with the help of our special meta-class `Enumeration`, discussed in the tutorial on enumerations, at the beginning of the `Book.js` model class file in the following way:

```
BookCategoryEL = new Enumeration([ "Textbook", "Biography"]);
```

2.3.3. Encode the model class `Book`

We encode the model class `Book` in the form of a constructor function where the `category` attribute as well as the segment attributes `subjectArea` and `about` are optional:

```
function Book( slots) {
  // set the default values for the parameter-free default constructor
  this.isbn = "";         // String
  this.title = "";        // String
  this.year = 0;          // Number (PositiveInteger)
/* optional properties
  this.category            // Number {from BookCategoryEL}
  this.subjectArea        // String
  this.about              // String
*/
  if (arguments.length > 0) {
    this.setIsbn( slots.isbn);
    this.setTitle( slots.title);
    this.setYear( slots.year);
    if (slots.category) this.setCategory( slots.category);
    if (slots.subjectArea) this.setSubjectArea( slots.subjectArea);
    if (slots.about) this.setAbout( slots.about);
  }
}
```

We encode the `checkCategory` and `setCategory` methods for the `category` attribute in the following way:

```
Book.checkCategory = function (t) {
  if (!t) {
    return new NoConstraintViolation();
  } else {
    if (!Number.isInteger( t) || t < 1 || t > BookCategoryEL.MAX) {
      return new RangeConstraintViolation(
          "The value of category must represent a book type!");
    } else {
      return new NoConstraintViolation();
    }
  }
};

Book.prototype.setCategory = function (t) {
  var constraintViolation = null;
  if (this.category) {  // already set/assigned
    constraintViolation = new FrozenValueConstraintViolation(
        "The category cannot be changed!");
  } else {
    t = parseInt( t);
    constraintViolation = Book.checkCategory( t);
  }
  if (constraintViolation instanceof NoConstraintViolation) {
    this.category = t;
  } else {
    throw constraintViolation;
  }
};
```

While the setters for segment properties follow the standard pattern, their checks have to make sure that the attribute applies to the category of the instance being checked. This is achieved by checking a combination of a property value and a category, as in the following example:

```
Book.checkSubjectArea = function (sa,t) {
  if (t === undefined) t = BookCategoryEL.TEXTBOOK;
  if (t === BookCategoryEL.TEXTBOOK && !sa) {
    return new MandatoryValueConstraintViolation(
        "A subject area must be provided for a textbook!");
  } else if (t !== BookCategoryEL.TEXTBOOK && sa) {
    return new OtherConstraintViolation("A subject area must not
        be provided if the book is not a textbook!");
  } else if (sa && (typeof(sa) !== "string" || sa.trim() === "")) {
    return new RangeConstraintViolation(
        "The subject area must be a non-empty string!");
  } else {
    return new NoConstraintViolation();
  }
};
```

In the serialization function `toString`, we serialize the category attribute and the segment properties in a `switch` statement:

```
Book.prototype.toString = function () {
  var bookStr = "Book{ ISBN:"+ this.isbn +", title:"+ this.title +
      ", year:"+ this.year;
  switch (this.category) {
  case BookCategoryEL.TEXTBOOK:
    bookStr += ", textbook subject area:"+ this.subjectArea;
    break;
  case BookCategoryEL.BIOGRAPHY:
    bookStr += ", biography about: "+ this.about;
    break;
  }
  return bookStr +" }";
};
```

In the update method of a model class, we only set a property if it is to be updated (that is, if there is a corresponding slot in the `slots` parameter) and if the new value is different from the old value. In the special case of a `category` attribute with a *Frozen Value Constraint*, we need to make sure that it can only be updated, along with an accompanying set of segement properties, if it has not yet been assigned. Thus, in the `Book.update` method, we perform the special test if `book.category === undefined` for handling the special case of an initial assignment, while we handle updates of the optional segment properties `subjectArea` and `about` in a more standard way:

```
Book.update = function (slots) {
  var book = Book.instances[slots.isbn],
      updatedProperties = [],
      ...;
  try {
    ...
    if ("category" in slots && book.category === undefined) {
      book.setCategory( slots.category);
      updatedProperties.push("category");
      switch (slots.category) {
      case BookCategoryEL.TEXTBOOK:
        book.setSubjectArea( slots.subjectArea);
        updatedProperties.push("subjectArea");
        break;
      case BookCategoryEL.BIOGRAPHY:
        book.setBiography( slots.biography);
        updatedProperties.push("biography");
        break;
      }
    }
    if ("subjectArea" in slots && "subjectArea" in book &&
            book.subjectArea !== slots.subjectArea) {
      book.setSubjectArea( slots.subjectArea);
      updatedProperties.push("subjectArea");
    }
    if ("about" in slots && "about" in book &&
            book.about !== slots.about) {
      book.setAbout( slots.about);
      updatedProperties.push("about");
    }
  } catch (e) {
    ...
  }
  ...
};
```

2.4. 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 `books.html`). Such a data management page contains 5 sections: *manage books*, *list books*, *create book*, *update book* and *delete book*, such that only one of them is displayed at any time (by setting the CSS property `display:none` for all others).

2.4.1. Summary

We have to take care of handling the `category` attribute 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 `books.html`.

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 `books.html`. Segment property form fields are only displayed, and their validation event handlers set, 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".

2.4.2. 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`:

```
<table id="books">
  <thead><tr><th>ISBN</th><th>Title</th><th>Year</th><th>Special type</th></tr></thead>
  <tbody></tbody>
</table>
```

A book without a special category will have an empty table cell in this column, while for all other books their category will be shown in this column, along with other segment-specific information. This requires a corresponding switch statement in `pl.view.books.list.setupUserInterface` in the `view/books.js` file:

```
if (book.category) {
  switch (book.category) {
  case BookCategoryEL.TEXTBOOK:
    row.insertCell(-1).textContent = book.subjectArea + " textbook";
    break;
  case BookCategoryEL.BIOGRAPHY:
    row.insertCell(-1).textContent = "Biography about "+ book.about;
    break;
  }
}
```

2.4.3. 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:

```
<p class="pure-control-group">
  <label for="creBookType">Special type: </label>
  <select id="creBookType" name="category"></select>
</p>
<p class="pure-control-group Textbook">
  <label for="creSubjectArea">Subject area: </label><input id="creSubjectArea" name="subjectArea" />
</p>
<p class="pure-control-group Biography">
  <label for="creAbout">About: </label><input id="creAbout" name="about" />
</p>
```

Notice that we have added "Textbook" and "Biography" as additional class values to the HTML `class` attributes of the `p` elements containing the corresponding form controls. This allows easy rendering and un-rendering of "Textbook" and "Biography" form controls, depending on the value of the `category` attribute (a mechanism called *dynamic forms*).

In the `handleTypeSelectChangeEvent` handler, segment property form fields are only displayed, with `pl.view.app.displaySegmentFields`, and their validation event handlers set, when a corresponding book category has been selected:

```
pl.view.books.handleTypeSelectChangeEvent = function (e) {
  var formEl = e.currentTarget.form,
      typeIndexStr = formEl.category.value,  // the array index of BookCategoryEL.names
      category=0;
  if (typeIndexStr) {
    category = parseInt( typeIndexStr) + 1;
    switch (category) {
    case BookCategoryEL.TEXTBOOK:
      formEl.subjectArea.addEventListener("input", function () {
        formEl.subjectArea.setCustomValidity(
            Book.checkSubjectArea( formEl.subjectArea.value).message);
      });
      break;
    case BookCategoryEL.BIOGRAPHY:
      formEl.about.addEventListener("input", function () {
        formEl.about.setCustomValidity(
            Book.checkAbout( formEl.about.value).message);
      });
      break;
    }
    pl.view.app.displaySegmentFields( formEl, BookCategoryEL.names, category);
  } else {
    pl.view.app.undisplayAllSegmentFields( formEl, BookCategoryEL.names);
  }
};
```

3. Case Study 2: Implementing a Class Hierarchy

Whenever a class hierarchy is more complex, we cannot simply eliminate it, but have to implement it (1) in the app's model code, (2) in its user interface and (3) in the underlying database. The starting point for our case study is the design model shown in Figure 2.2 above. In the following sections, we derive a *JavaScript data model* and a *entity table model* from the design model. The entity table model is used as a design for the object-to-JSON mapping that we need for storing the objects of our app in Local Storage.

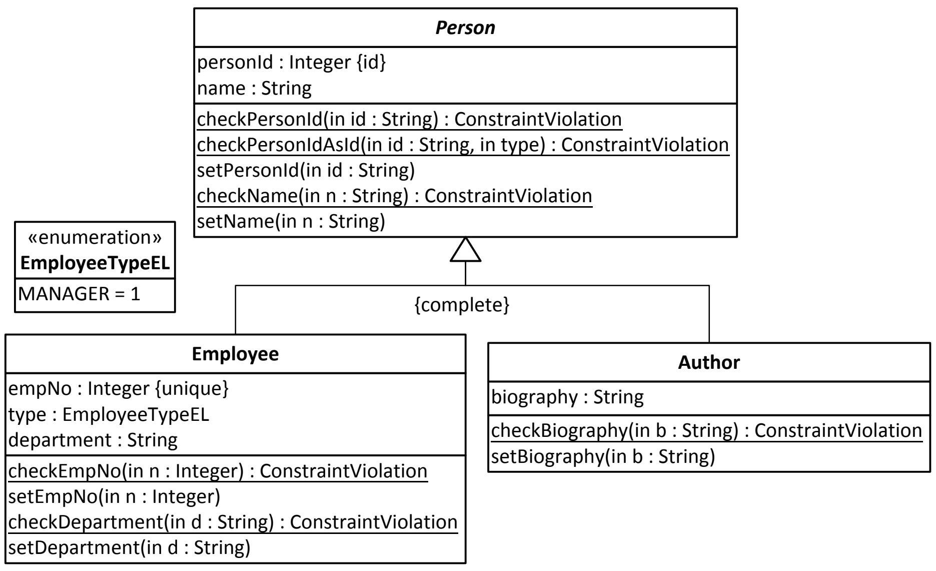
3.1. Make the JavaScript data model

We design the *model classes* of our example app with the help of a *JavaScript data model* that we derive from the *design model* by essentially leaving the generalization arrows as they are and just adding *checks* and *setters* to each class, as described in Part 2 of this tutorial. However, in the case of our example app, it is natural to apply the *Class Hierarchy Merge* design pattern (discussed in Section 5) to the segmentation of `Employee` for simplifying the data model by eliminating the `Manager` subclass. This leads to the model shown in Figure 2.6 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`, implying that we do not need to maintain its extension (in a map like `Person.instances`), as we do for all other non-abstract classes. 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 2.6. The JavaScript data model of the Person class hierarchy


Person
personId : Integer {id}
name : String
checkPersonId(in id : String) : ConstraintViolation
checkPersonIdAsId(in id : String, in type) : ConstraintViolation
setPersonId(in id : String)
checkName(in n : String) : ConstraintViolation
setName(in n : String)
«enumeration» EmployeeTypeEL
MANAGER = 1
{complete}
Employee
empNo : Integer {unique}
type : EmployeeTypeEL
department : String
checkEmpNo(in n : Integer) : ConstraintViolation
setEmpNo(in n : Integer)
checkDepartment(in d : String) : ConstraintViolation
setDepartment(in d : String)
Author
biography : String
checkBiography(in b : String) : ConstraintViolation
setBiography(in b : String)

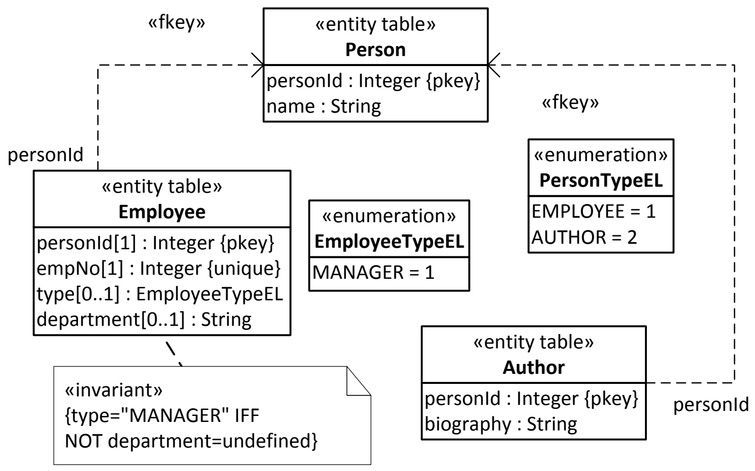
3.2. Make the entity table model

Since we use *Local Storage* as the persistent storage technology for our example app, we have to deal with simple key-value storage. For each model class with a singular name `Entity`, we use its pluralized name `entities` as a key such that its associated value is the entity table serialization of the main memory object collection `Entity.instances`.

We design a set of suitable JSON tables and the structure of their records in the form of a *entity table model* that we derive from the design model by following certain rules. We basically have two choices how to organize our JSON data store and how to derive a corresponding entity table model: either according to the *Single Table Inheritance* approach, where a segmentation or an entire class hierarchy is represented with a single table, or according to the *Joined Tables Inheritance* approach, where we have a separate table for each model class of the class hierarchy. Both approaches, which are discussed in Section 6.2, can be combined for the same design model.

In our example it seems natural to apply the *Single Table Inheritance* approach to the incomplete segmentation of `Employee` with just one segment subclass `Manager`, while we apply the *Joined Tables Inheritance* approach to the complete segmentation of `Person` into `Employee` and `Author`. This results in the model shown in Figure 2.7 below.

Figure 2.7. The entity table model of the Person class hierarchy



Notice that we have replaced the `{id}` property modifier for designating the standard ID attribute(s) with `{pkey}` for indicating that the attributes concerned act as primary keys in combination with foreign keys expressed by the dashed dependency arrows stereotyped with «fkey» and annotated with the foreign key attribute (here: `personId`) at their source end. An example of an admissible population for this table model is the following:

Person	Employee	Author
{personId: 1001, name:"Gerd Wagner"}	{personId: 1001, empNo: 21035}	{personId: 1001, biography:"Born in ..."}
{personId: 1002, name:"Tom Boss"}	{personId: 1002, empNo:23107, category: EmployeeTypeEL.MANAGER, department:"Faculty1"}	
{personId: 1077, name:"Immanuel Kant"}		{personId: 1077, biography:"Kant was ..."}

Notice the mismatch between the JavaScript data model shown in Figure Figure 2.5 above, which is the basis for the model classes `Person`, `Employee` and `Author` as well as for the main memory database consisting of `Employee.instances` and `Author.instances` on one hand side, and the entity table model, shown in Figure 2.7 above on the other hand side. While we do not have any `Person` records in the main memory database, we do have them in the persistent datastore based on the entity table model. This mismatch results from the complete structure of JavaScript subclass instances, which include all property slots, as opposed to the fragmented structure of database tables based on the *Joined Tables Inheritance* approach.

3.3. New issues

Compared to the model of our first case study, shown in Figure 2.5 above, we have to deal with a number of new issues in the *model code*:

1. Defining the category relationships between `Employee` and `Person`, as well as between `Author` and `Person`, using the JavaScript code pattern for constructor-based inheritance discussed in Section 1.

2. When loading the instances of a category from persistent storage (as in `Employee.loadAll` and `Author.loadAll`), their slots for inherited supertype properties, except for the standard identifier attribute, have to be reconstructed from corresponding rows of the supertable (`persons`).

3. When saving the instances of `Employee` and `Author` as records of the JSON tables `employees` and `authors` to persistent storage (as in `pl.view.employees.manage.exit` and `pl.view.authors.manage.exit`), we also need to save the records of the supertable `persons` by extracting their data from corresponding `Employee` or `Author` instances.

3.4. Encode the model classes of the JavaScript data model

The JavaScript data model shown in Figure 2.6 above can be directly encoded for getting the code of the model classes `Person`, `Employee` and `Author` as well as for the enumeration type `EmployeeTypeEL`.

3.4.1. Define the category relationships

We define the category relationships between `Employee` and `Person`, as well as between `Author` and `Person`, using the JavaScript code pattern for constructor-based inheritance discussed in Section 1. For instance, in `model/Employee.js` we define:

```
function Employee( slots) {
  // set the default values for the parameter-free default constructor
  Person.call( this);  // invoke the default constructor of the supertype
  this.empNo = 0;      // Number (PositiveInteger)
  // constructor invocation with arguments
  if (arguments.length > 0) {
    Person.call( this, slots);  // invoke the constructor of the supertype
    this.setEmpNo( slots.empNo);
    if (slots.category) this.setCategory( slots.category);  // optional
    if (slots.department) this.setDepartment( slots.department);  // optional
  }
};
Employee.prototype = Object.create( Person.prototype);  // inherit from Person
Employee.prototype.constructor = Employee;  // adjust the constructor property
```

3.4.2. Reconstruct inherited supertype properties when loading the instances of a category

When loading the instances of a category from persistent storage (as in `Employee.loadAll` and `Author.loadAll`), their slots for inherited supertype properties, except for the standard identifier attribute, have to be reconstructed from corresponding rows of the supertable (`persons`). For instance, in `model/Employee.js` we define:

```
Employee.loadAll = function () {
  var key="", keys=[], persons={}, employees={}, employeeRow={}, i=0;
  if (!localStorage["employees"]) {
    localStorage.setItem("employees", JSON.stringify({}));
  }
  try {
    persons = JSON.parse( localStorage["persons"]);
    employees = JSON.parse( localStorage["employees"]);
  } catch (e) {
    console.log("Error when reading from Local Storage\n" + e);
  }
  keys = Object.keys( employees);
  console.log( keys.length +" employees loaded.");
  for (i=0; i < keys.length; i++) {
    key = keys[i];
    employeeRow = employees[key];
    // complete record by adding slots ("name") from supertable
    employeeRow.name = persons[key].name;
    Employee.instances[key] = Employee.convertRow2Obj( employeeRow);
  }
};
```

3.4.3. Reconstruct and save the supertable `Person` when saving the main memory data

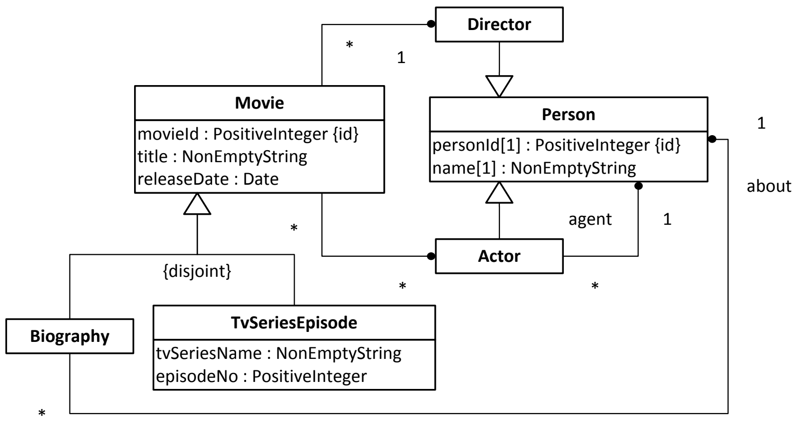
When saving the instances of `Employee` and `Author` as records of the JSON tables `employees` and `authors` to persistent storage (as in `pl.view.employees.manage.exit` and `pl.view.authors.manage.exit`), we also need to save the records of the supertable `persons` by extracting their data from corresponding `Employee` or `Author` instances. For this purpose, we define in `model/Person.js`:

```
Person.saveAll = function () {
  var key="", keys=[], persons={}, i=0, n=0;
  keys = Object.keys( Employee.instances);
  for (i=0; i < keys.length; i++) {
    key = keys[i];
    emp = Employee.instances[key];
    persons[key] = {personId: emp.personId, name:emp.name};
  }
  keys = Object.keys( Author.instances);
  for (i=0; i < keys.length; i++) {
    key = keys[i];
    if (!persons[key]) {
      author = Author.instances[key];
      persons[key] = {personId: author.personId, name: author.name};
    }
  }
  try {
    localStorage["persons"] = JSON.stringify( persons);
    n = Object.keys( persons).length;
    console.log( n +" persons saved.");
  } catch (e) {
    alert("Error when writing to Local Storage\n" + e);
  }
};
```

4. Practice Project

The purpose of the app to be built in this project is managing information about movies as well as their directors and actors where two types of movies are distinguished: biographies and episodes of TV series.

Figure 2.8. Two class hierarchies: `Movie` with two disjoint subtypes and `Person` with two overlapping subtypes.



Notice that `Movie` has two disjoint subtypes, `Biography` and `TvSeriesEpisode`, forming an incomplete segmentation of `Movie`, while `Person` has two overlapping subtypes, `Director` and `Actor`, forming an incomplete segmentation of `Person`.

Encode the app by following the guidance provided in the tutorial.

Make sure that your pages comply with the XML syntax of HTML5, and that your JavaScript code complies with our Coding Guidelines [http://oxygen.informatik.tu-cottbus.de/webeng/Coding-Guidelines.html] (and is checked with JSLint [http://www.jslint.com/]).