JavaScript Front-End Web App Tutorial Part 5: Managing Bidirectional Associations

Learn how to manage bidirectional associations between object types, such as between books and their authors, and authors and their books

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

JavaScript Front-End Web App Tutorial Part 5: Managing Bidirectional Associations: Learn how to manage bidirectional associations between object types, such as between books and their authors, and authors and their books

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 [bidirectional-association-tutorial.pdf]. You may run the example app [BidirectionalAssociationApp/index.html] from our server, or download it as a ZIP archive file [BidirectionalAssociationApp.zip]. See also our Web Engineering project page [http://web-engineering.info/].

Publication date 2015-11-12

Table of Contents

List of Figures

List of Tables

Foreword

This tutorial is Part 5 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 takes care of the object types `Author`, `Publisher` and `Book` as well as the bidirectional associations between `Book` and `Author` and between `Book` and `Publisher`.

The app supports the four standard data management operations (**C**reate/**R**ead/**U**pdate/**D**elete). It extends the example app of part 3 by adding code for handling ***derived inverse reference properties***. The other parts of the tutorial 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 6 [subtyping-tutorial.html]: Handling **subtype** (inheritance) relationships between object types.

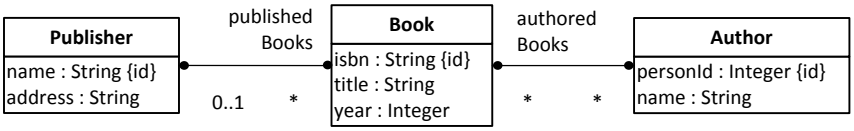
You may also want to take a look at our open access book Building Front-End Web Apps with Plain JavaScript [http://web-engineering.info/JsFrontendApp-Book], which includes all parts of the tutorial in one document, dealing with multiple object types ("books", "publishers" and "authors") and taking care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

Chapter 1. Bidirectional Associations

A bidirectional association is an association that is represented as a pair of mutually inverse reference properties.

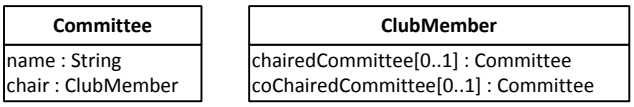
The model shown in Figure 1.1 below (about publishers, books and their authors) serves as our running example in all other parts of the tutorial. Notice that it contains two bidirectional associations, as indicated by the ownership dots at both association ends.

Figure 1.1. The Publisher-Book-Author information design model with two bidirectional associations

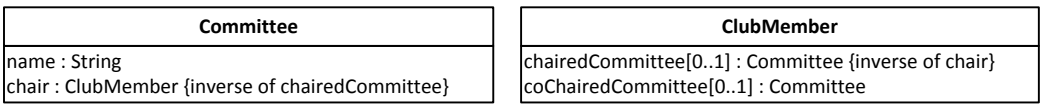


1. Inverse Reference Properties

For being able to easily retrieve the committees that are chaired or co-chaired by a club member, we add two reference properties to our `Committee-ClubMember` example model: the property of a club member to chair a committee (`ClubMember::chairedCommittee`) and the property of a club member to co-chair a committee (`ClubMember::coChairedCommittee`). We assume that any club member may chair or co-chair at most one committee (where the disjunction is non-exclusive). So, we get the following model:



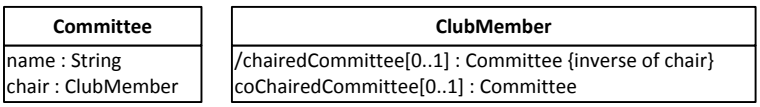
Notice that there is a close correspondence between the two reference properties `Committee::chair` and `ClubMember::chairedCommittee`. They are the **inverse** of each other: when the club member Tom is the chair of the budget committee, expressed by the tuple *"(budget committe"e, "To"m)*, then the budget committee is the committee chaired by the club member Tom, expressed by the inverse tuple *"(To"m, "budget committe"e)*. For expressing this inverse correspondence in the diagram, we append an inverse property constraint, `inverse of chair`, in curly braces to the declaration of the property `ClubMember::chairedCommittee`, and a similar one to the property `Committee::chair`, as shown in the following diagram:



We also call `Committee::chair` and `ClubMember::chairedCommittee` a **pair of mutually inverse reference properties**. Having such a pair in a model means having **redundancy** because each of the two involved reference properties can be derived from the other by inversion. This type of redundancy implies *data storage overhead* and *update overhead*, which is the price to pay for the bidirectional navigability that supports efficient object access in both directions.

For maintaining the duplicate information of a mutually inverse reference property pair, it is good practice to treat one of the two involved properties as the *master*, and the other one as the *slave*, and take this distinction into consideration in the code of the change methods (such as the property setters) of the

affected model classes. We indicate the slave of an inverse reference property pair in a model diagram by declaring the slave property to be a **derived** property using the UML notation of a slash as a prefix of the property name as shown in the following diagram:



The property `chairedCommittee` in `ClubMember` now has a slash-prefix (`/`) indicating that it is derived. The `{inverse of chair}` annotation in this example defines the derivation rule: the property is derived by inversion of `Committee::chair`. This implies that, the other way around, `Committee::chair` is the inverse of `ClubMember::chairedCommittee`.

In a UML class diagram, the derivation of a property can be specified, for instance, by an *Object Constraint Language (OCL)* expression that evaluates to the value of the derived property for the given object. In the case of a property being the inverse of another property, specified by the constraint expression `{inverse of` *anotherProperty*`}` appended to the property declaration, the derivation expression is implied. In our example, it evaluates to the committee object reference `c` such that `c.chair = this`.

There are two ways how to realize the derivation of a property: it may be *derived on read* via a read-time computation of its value, or it may be *derived on update* via an update-time computation performed whenever one of the variables in the derivation expression (typically, another property) changes its value. The latter case corresponds to a *materialized view* in an SQL database. While a reference property that is derived on read may not guarantee efficient navigation, because the on-read computation may create unacceptable latencies, a reference property that is derived on update does provide efficient navigation.

In the case of a derived reference property, the derivation expresses **life cycle dependencies**. These dependencies require special consideration in the code of the affected model classes by providing a number of change management mechanisms based on the functionality type of the represented association (either *one-to-one*, *many-to-one* or *many-to-many*).

In our example of the derived inverse reference property `ClubMember::chairedCommittee`, which is single-valued and optional, this means that

1. whenever a new committee object is created (with a mandatory `chair` assignment), the corresponding `ClubMember::chairedCommittee` property has to be assigned accordingly;

2. whenever the `chair` property is updated (that is, a new chair is assigned to a committee), the corresponding `ClubMember::chairedCommittee` property has to be updated as well;

3. whenever a committee object is destroyed, the corresponding `ClubMember::chairedCommittee` property has to be unassigned.

In the case of a derived inverse reference property that is multi-valued while its inverse base property is single-valued (like `Publisher::publishedBooks` in Figure 1.4 below being derived from `Book::publisher`), the life cycle dependencies imply that

1. whenever a new 'base object' (such as a book) is created, the corresponding inverse property has to be updated by adding a reference to the new base object to its value set (like adding a reference to the new book object to `Publisher::publishedBooks`);

2. whenever the base property is updated (e.g., a new publisher is assigned to a book), the corresponding inverse property (in our example, `Publisher::publishedBooks`) has to be updated as well by removing the old object reference from its value set and adding the new one;

3. whenever a base object (such as a book) is destroyed, the corresponding inverse property has to be updated by removing the reference to the base object from its value set (like removing a reference to the book object to be destroyed from `Publisher::publishedBooks`).

We use the slash-prefix (`/`) notation in the example above for indicating that the property `ClubMember::chairedCommittee` is derived on update from the corresponding committee object.

2. Making an Association-Free Information Design Model

How to eliminate explicit associations from a design model by replacing them with reference properties

Since classical OO programming languages do not support associations as first class citizens, but only classes and reference properties representing implicit associations, we have to eliminate all explicit associations for obtaining an OO design model.

2.1. The basic procedure

The starting point of our **association elimination** procedure is an information design model with various kinds of unidirectional and bidirectional associations, such as the model shown in Figure 1.1 above. If the model still contains any non-directional associations, we first have to turn them into directional ones by making a decision on the ownership of their ends, which is typically based on navigability requirements.

Notice that both associations in the *Publisher-Book-Author* information design model, *publisher-publishedBooks* and *authoredBooks-authors* (or *Authorship*), are bidirectional as indicated by the ownership dots at both association ends. For eliminating all explicit associations from an information design model, we have to perform the following steps:

1. **Eliminate unidirectional associations**, connecting a source with a target class, by replacing them with a reference property in the source class such that the target class is its range.

2. **Eliminate bidirectional associations** by replacing them with a pair of mutually inverse reference properties.

2.2. How to eliminate unidirectional associations

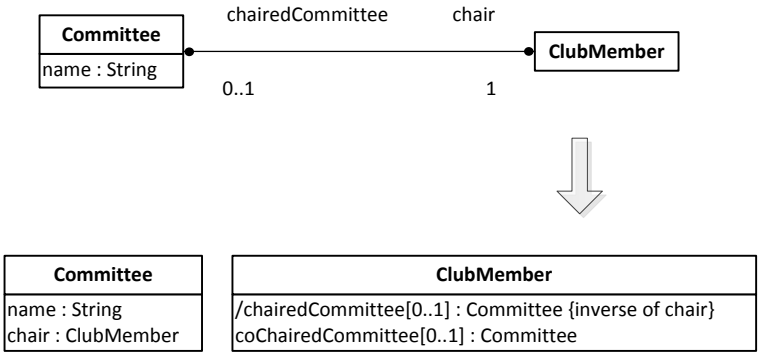
A unidirectional association connecting a source with a target class is replaced with a corresponding reference property in its source class having the target class as its range. Its multiplicity is the same as the multiplicity of the target association end. Its name is the name of the association end, if there is any, otherwise it is set to the name of the target class (possibly pluralized, if the reference property is multi-valued).

2.3. How to eliminate bidirectional associations

A bidirectional association, such as `Authorship` in the model shown in Figure 1.1 above, is replaced with a pair of mutually inverse reference properties, such as `Book::authors` and `Author::authoredBooks`. Since both reference properties represent the same information (the same set of binary relationships), it's an option to conisder one of them being the master and the other one the slave, which is derived from the master. We discuss the two cases of a one-to-one and a many-to-many association

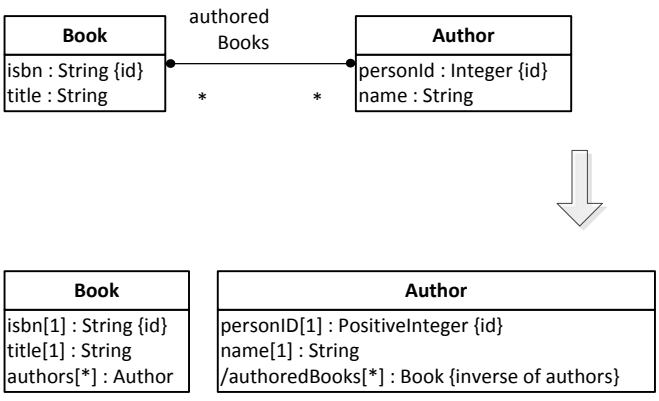
1. In the case of a bidirectional one-to-one association, this leads to a pair of mutually inverse single-valued reference properties, one in each of the two associated classes. Since both of them represent essentially the same information (one of them is the inverse of the other), one has to choose which of them is considered the "master", and which of them is the "slave", where the "slave" property is considered to represent the inverse of the "master". In the slave class, the reference property representing the inverse association is designated as a ***derived property*** that is automatically updated whenever 1) a new master object is created, 2) the master reference property is updated, or 3) a master object is destroyed.

Figure 1.2. Turn a bidirectional one-to-one association into a pair of mutually inverse single-valued reference properties



2. A bidirectional many-to-many association is mapped to a pair of mutually inverse multi-valued reference properties, one in each of the two classes participating in the association. Again, in one of the two classes, the multi-valued reference property representing the (inverse) association is designated as a ***derived property*** that is automatically updated whenever the corresponding property in the other class (where the association is maintained) is updated.

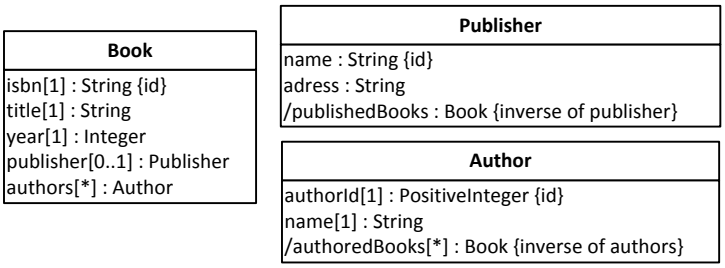
Figure 1.3. Turn a bidirectional many-to-many association into a master-slave pair of mutually inverse multi-valued reference properties



2.4. The resulting association-free design model

After replacing both bidirectional associations with reference properties, we obtain the following association-free design model:

Figure 1.4. The association-free design model



Chapter 2. Implementing Bidirectional Associations with Plain JavaScript

In this chapter of our tutorial, we show

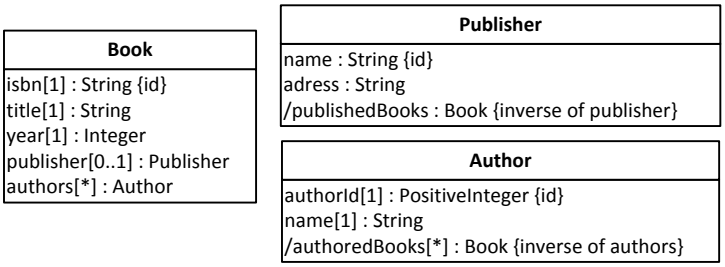
1. how to derive a JavaScript data model from an association-free information design model with **derived inverse reference properties**,

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. Make a JavaScript Data Model

The starting point for making our JavaScript data model is an association-free information design model with derived inverse reference properties like the following one:



Notice that there are two derived inverse reference properties: `Publisher::/publishedBooks` and `Author::/authoredBooks`.

The meaning of the design model and its reference properties `publisher` and `authors` can be illustrated by a sample data population for the three model classes:

Table 2.1. Sample data for `Publisher`

Name	Address	Published books
Bantam Books	New York, USA	0553345842
Basic Books	New York, USA	0465030793

Table 2.2. Sample data for `Book`

ISBN	Title	Year	Authors	Publisher
0553345842	The Mind's I	1982	1, 2	Bantam Books
1463794762	The Critique of Pure Reason	2011	3	
1928565379	The Critique of Practical Reason	2009	3	
0465030793	I Am A Strange Loop	2000	2	Basic Books

Table 2.3. Sample data for `Author`

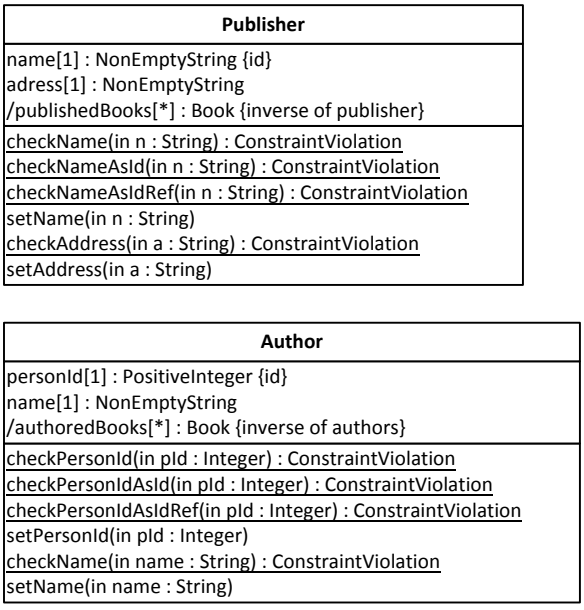
Author ID	Name	Authored books
1	Daniel Dennett	0553345842
2	Douglas Hofstadter	0553345842, 0465030793
3	Immanuel Kant	1463794762, 1928565379

We now show how to derive a JavaScript data model from the design model in three steps.

1. Create a **check** operation for each non-derived property. This step has been discussed in detail in the previous parts of the tutorial (about data validation and about unidirectional associations).
2. Create a **set** operation for each non-derived **single-valued** property. In the setter, the corresponding check operation is invoked and the property is only set, if the check does not detect any constraint violation.
3. Create an **add**, a **remove** and a **set** operation for each non-derived **multi-valued** property.

This leads to the following JavaScript data model classes `Publisher` and `Author`. Notice that we don't show the model class `Book`, since it is the same as in the data model for the unidirectional association app discussed in the previous Part of our tutorial.

Figure 2.1. The JavaScript data model without the class `Book`



2. Write the Model Code

How to Encode a JavaScript Data Model

The JavaScript data model can be directly encoded for getting the code of the model layer of our JavaScript frontend app.

2.1. New issues

Compared to the *unidirectional association app* discussed in a previous tutorial, we have to deal with a number of new technical issues:

1. In the *model code* you now have to take care of maintaining the derived inverse reference properties by maintaining the derived (sets of) inverse references that form the values of a derived inverse reference property. This requires in particular that

   a. whenever the value of a ***single-valued*** master reference property is **initialized or updated** with the help of a setter (such as assigning a reference to a `Publisher` instance `p` to `b.publisher` for a `Book` instance `b`), an inverse reference has to be assigned or added to the corresponding value of the derived inverse reference property (such as adding `b` to `p.publishedBooks`); when the value of the master reference property is *updated* and the derived inverse reference property is *multi-valued*, then the obsolete inverse reference to the previous value of the single-valued master reference property has to be deleted;

   b. whenever the value of an optional ***single-valued*** master reference property is ***unset*** (e.g. by assigning `null` to `b.publisher` for a `Book` instance `b`), the inverse reference has to be removed from the corresponding value of the derived inverse reference property (such as removing `b` from `p.publishedBooks`), if the derived inverse reference property is multi-valued, otherwise the corresponding value of the derived inverse reference property has to be unset or updated;

   c. whenever a reference is **added** to the value of a **multi-valued** master reference property with the help of an add method (such as adding an `Author` reference `a` to `b.authors` for a `Book` instance `b`), an inverse reference has to be assigned or added to the corresponding value of the derived inverse reference property (such as adding `b` to `a.authoredBooks`);

   d. whenever a reference is **removed** from the value of a **multi-valued** master reference property with the help of a `remove` method (such as removing a reference to an `Author` instance `a` from `b.authors` for a `Book` instance `b`), the inverse reference has to be removed from the corresponding value of the derived inverse reference property (such as removing `b` from `a.authoredBooks`), if the derived inverse reference property is multi-valued, otherwise the corresponding value of the derived inverse reference property has to be unset or updated;

   e. whenever an object with a single reference or with multiple references as the value of a master reference property is **destroyed** (e.g., when a `Book` instance `b` with a single reference `b.publisher` to a `Publisher` instance `p` is destroyed), the derived inverse refences have to be removed first (e.g., by removing `b` from `p.publishedBooks`).

   Notice that when a new object is created with a single reference or with multiple references as the value of a master reference property (e.g., a new `Book` instance `b` with a single reference `b.publisher`), its setter or add method will be invoked and will take care of creating the derived inverse references.

2. In the *UI code* we can now exploit the inverse reference properties for more efficiently creating a list of inversely associated objects in the *list objects* use case. For instance, we can more efficiently create a list of all published books for each publisher. However, we do not allow updating the set of inversely associated objects in the *update object* use case (e.g. updating the set of published books in the *update publisher* use case). Rather, such an update has to be done via updating the master objects (in our example, the books) concerned.

2.2. Summary

1. Encode each model class as a JavaScript constructor function.

2. Encode the property checks in the form of class-level ('static') methods. Take care that all constraints of a property as specified in the JavaScript data model are properly encoded in the property checks.

3. Encode the property setters as (instance-level) methods. In each setter, the corresponding property check is invoked and the property is only set, if the check does not detect any constraint violation. If the property is the inverse of a derived reference property (representing a bidirectional association), make sure that the setter also assigns (or adds) corresponding references to (the value set of) the inverse property.

4. Encode the add/remove operations as (instance-level) methods that invoke the corresponding property checks. If the multi-valued reference property is the inverse of a derived reference property (representing a bidirectional association), make sure that both the add and the remove operation also assign/add/remove corresponding references to/from (the value set of) the inverse property.

5. Encode any other operation.

6. Take care of the deletion dependencies in `deleteRow` methods.

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

2.3. Encode each class of the JavaScript data model as a constructor function

For instance, the `Publisher` class from the JavaScript data model is encoded in the following way:

```
function Publisher( slots) {
  // set the default values for the parameter-free default constructor
  this.name = "";            // String
  this.address = "";         // String, optional
  // derived inverse reference property (map of Book objects)
  this.publishedBooks = {};  // inverse of Book::publisher

  // constructor invocation with arguments
  if (arguments.length > 0) {
    this.setName( slots.name);
    this.setAddress( slots.address);
  }
};
```

Notice that we have added the (derived) multi-valued reference property `publishedBooks`, but we do not assign it in the constructor function because it will be assigned when the inverse reference property `Book::publisher` will be assigned.

2.4. Encode the property checks

The property checks from the *unidirectional association app* do not need to be changed in any way.

2.5. Encode the setter operations

Any setter for a reference property that is coupled to a derived inverse reference property (implementing a bidirectional association), now also needs to assign/add/remove inverse references to/from the corresponding value (set) of the inverse property. An example of such a setter is the following `setPublisher` method:

```
Book.prototype.setPublisher = function (p) {
  var constraintViolation = null;
  var publisherIdRef = "";
  // a publisher can be given as ...
  if (typeof(p) !== "object") {  // an ID reference or
```

```
    publisherIdRef = p;
  } else {                       // an object reference
    publisherIdRef = p.name;
  }
  constraintViolation = Book.checkPublisher( publisherIdRef);
  if (constraintViolation instanceof NoConstraintViolation) {
    if (this.publisher) {  // update existing book record
      // delete the obsolete inverse reference in Publisher::publishedBooks
      delete this.publisher.publishedBooks[ this.isbn];
    }
    // assign the new publisher reference
    this.publisher = Publisher.instances[ publisherIdRef];
    // add the inverse reference to publisher.publishedBooks
    this.publisher.publishedBooks[ this.isbn] = this;
  } else {
    throw constraintViolation;
  }
};
```

2.6. Encode the add and remove operations

For any multi-valued reference property that is coupled to a derived inverse reference propertiy, both the *add* and the *remove* method also have to assign/add/remove corresponding references to/from (the value set of) the inverse property.

For instance, for the multi-valued reference property `Book::authors` that is coupled to the derived inverse reference propertiy `Author:authoredBooks` for implementing the bidirectional authorship association between `Book` and `Author`, the `addAuthor` method is encoded in the following way:

```
Book.prototype.addAuthor = function( a) {
  var constraintViolation=null, authorIdRef=0, authorIdRefStr="";
  // an author can be given as ...
  if (typeof( a) !== "object") {  // an ID reference or
    authorIdRef = parseInt( a);
  } else {                       // an object reference
    authorIdRef = a.authorId;
  }
  constraintViolation = Book.checkAuthor( authorIdRef);
  if (authorIdRef && constraintViolation instanceof NoConstraintViolation) {
    authorIdRefStr = String( authorIdRef);
    // add the new author reference
    this.authors[ authorIdRefStr] = Author.instances[ authorIdRefStr];
    // automatically add the derived inverse reference
    this.authors[ authorIdRefStr].authoredBooks[ this.isbn] = this;
  }
};
```

For the remove operation `removeAuthor` we obtain the following code:

```
Book.prototype.removeAuthor = function( a) {
  var constraintViolation=null, authorIdRef=0, authorIdRefStr="";
  // an author can be given as an ID reference or an object reference
  if (typeof(a) !== "object") {
    authorIdRef = parseInt( a);
  } else {
    authorIdRef = a.authorId;
  }
  constraintViolation = Book.checkAuthor( authorIdRef);
  if (authorIdRef && constraintViolation instanceof NoConstraintViolation) {
    authorIdRefStr = String( authorIdRef);
    // automatically delete the derived inverse reference
    delete this.authors[ authorIdRefStr].authoredBooks[ this.isbn];
    // delete the author reference
    delete this.authors[ authorIdRefStr];
  }
};
```

2.7. Take care of deletion dependencies

When a `Book` instance `b`, with a single reference `b.publisher` to a `Publisher` instance `p` and multiple references `b.authors` to `Author` instances, is destroyed, the derived inverse references have to be removed first (e.g., by removing `b` from `p.publishedBooks`).

```
Book.deleteRow = function (isbn) {
  var book = Book.instances[isbn], keys=[], i=0;
  if (book) {
    console.log( book.toString() + " deleted!");
    if (book.publisher) {
      // remove inverse reference from book.publisher
      delete book.publisher.publishedBooks[isbn];
    }
    // remove inverse references from all book.authors
    keys = Object.keys( book.authors);
    for (i=0; i < keys.length; i++) {
      delete book.authors[keys[i]].authoredBooks[isbn];
    }
    // finally, delete book from Book.instances
    delete Book.instances[isbn];
  } else {
    console.log("There is no book with ISBN " + isbn + " in the database!");
  }
};
```

3. Exploiting Derived Inverse Reference Properties in the User Interface

We can now exploit the derived inverse reference properties `Publisher::publishedBooks` and `Author::authoredBooks` for more efficiently creating a list of associated books in the *list publishers* and *list authors* use cases.

3.1. Show information about published books in the *List Publishers* use case

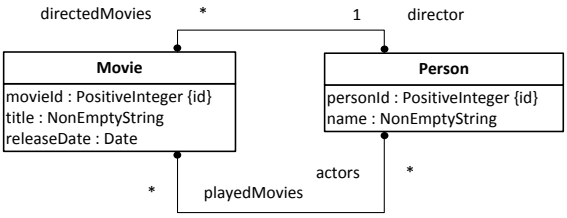
For showing information about published books in the *list publishers* use case, we can now exploit the derived inverse reference property `publishedBooks`:

```
pl.view.publishers.list = {
  setupUserInterface: function () {
    var tableBodyEl = document.querySelector("div#listPublishers>table>tbody");
    var pKeys = Object.keys( Publisher.instances);
    var row=null, publisher=null, listEl=null;
    tableBodyEl.innerHTML = "";
    for (var i=0; i < pKeys.length; i++) {
      publisher = Publisher.instances[pKeys[i]];
      row = tableBodyEl.insertRow(-1);
      row.insertCell(-1).textContent = publisher.name;
      row.insertCell(-1).textContent = publisher.address;
      // create list of books published by this publisher
      listEl = util.createListFromAssocArray( publisher.publishedBooks, "title");
      row.insertCell(-1).appendChild( listEl);
    }
    document.getElementById("managePublishers").style.display = "none";
    document.getElementById("listPublishers").style.display = "block";
  }
};
```

4. Practice Project

This project is based on the information design model below. The app from the previous assignment is to be extended by adding **derived inverse reference properties** for implementing the bidirectional associations. This is achieved by adding the multi-valued reference properties `directedMovies` and `playedMovies` to the model class `Person`, both with range `Movie`.

Figure 2.2. Two bidirectional associations between `Movie` and `Person`.



This project includes the following tasks:

1. Make an ***association-free design model*** derived from the given design model.

2. Make a ***JavaScript data model*** derived from the association-fee design model.

3. Encode your JavaScript data model, following the guidelines of the tutorial.

You can use the following sample data for testing your app:

Table 2.4. Movies

Movie ID	Title	Release date	Director	Actors
1	Pulp Fiction	1994-05-12	1	5, 6
2	Star Wars	1977-05-25	2	7, 8
3	Inglourious Basterds	2009-05-20	1	9, 1
4	The Godfather	1972-03-15	4	11, 12

Table 2.5. People

Person ID	Name	Directed movies	Played movies
1	Quentin Tarantino	1, 3	3
2	George Lucas	2	
4	Francis Ford Coppola	4	
5	Uma Thurman		1
6	John Travolta		1
7	Ewan McGregor		2
8	Natalie Portman		2
9	Brad Pitt		3
11	Marlon Brando		4
12	Al Pacino		4

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/].

If you have any questions about how to carry out this project, you can ask them on our discussion forum [http://web-engineering.info/forum/15].