Declarative and Responsive Constraint Validation with mODELcLASSjs

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

Declarative and Responsive Constraint Validation with mODELcLASSjs

by Gerd Wagner

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

This tutorial is also available in the following formats: PDF [validation-tutorial.pdf]. See also the project page [http://web-engineering.info/index.html], or run the example app [ValidationApp/index.html] from our server, or download the code [ValidationApp.zip] as a ZIP archive file.

Publication date 2016-01-16

Table of Contents

List of Figures

Foreword

This tutorial is Part 1 of a series of tutorials about model-based development of JavaScript web applications using the libraries mODELcLASSjs and mODELvIEWjs, which help to avoid repetitive ("boilerplate") code. mODELcLASSjs and mODELvIEWjs support both front-end JavaScript web apps with local or cloud storage and distributed JavaScript web apps with NodeJS-based back-end components and remote storage.

The tutorial shows how to express integrity constraints declaratively in a model class defined as an instance of the meta-class mODELcLASS, and how to validate data in the model layer of the app as well as perform *responsive constraint validation* in the user interface.

The simple form of a JavaScript data management application presented in this tutorial takes care of only one object type ("books") for which it supports the four standard data management operations (**C**reate/**R**etrieve/**U**pdate/**D**elete). It improves the validation app discussed in the plain JavaScript validation tutorial [http://web-engineering.info/tech/JsFrontendApp/validation-tutorial.html] by ***avoiding boilerplate code*** in

1. the model layer for checks and setters, and

2. the storage layer for the standard (CRUD) storage management operations.

It needs to be enhanced by adding further object types with associations and subtype/inheritance relationships between them.

You may also want to take a look at the book Building Front-End Web Apps with Plain JavaScript [http://web-engineering.info/JsFrontendApp-Book], which explains how to deal with multiple object types ("books", "publishers" and "authors"), taking care of constraint validation, association management and subtyping/inheritance.

Chapter 1. The Model-Based Development Framework mODELcLASSjs

1. Application Architecture

As in all cases of designing a complex system, no matter if a new building, a new space shuttle, a new computer or a new software application is to be designed, an architecture provides a kind of master plan for defining the structure of the system. Any good architecture is based on the fundamental engineering principle of **separation of concerns**, which helps in managing complexity by breaking a system down into smaller, functionally defined parts such that their **interdependencies are minimized**, keeping the more fundamental parts independent of the less fundamental parts. Such an architecture also makes it easier to develop, or source, and update certain system parts independently of others.

1.1. The Model-View-Controller (MVC) architecture metaphor

The most popular and most widely used approach in software application architecture is the **M**odel-**V**iew-**C**ontroller code partitioning pattern. Although it has not been precisely defined, and has been implemented in many different ways, especially in web application frameworks, it is based on the principle of *separation of concerns* and on the fundamental insight that the basis for all other parts of an application, and in particular for the user interface, is the **model**. Consequently, even if the MVC approach doesn't provide a precise definition of what a 'model' is, we can consider it to be a **model-based** approach.

According to Wikipedia, the first MVC architecture was introduced to application programming with *Smalltalk-76* by Trygve Reenskaug [https://en.wikipedia.org/wiki/Trygve_Reenskaug] in the 1970s. In a later article about Smalltalk-80 [https://web.archive.org/web/20100921030808/http://www.itu.dk/courses/VOP/E2005/VOP2005E/8_mvc_krasner_and_pope.pdf], MVC is explained as a "three-way division of an application" that entails "separating (1) the parts that represent the model of the underlying application domain from (2) the way the model is presented to the user and from (3) the way the user interacts with it". The authors, who also use the term "MVC metaphor", point out that their approach would allow programmers to "write an application model by first defining new classes that would embody the special application domain-specific information".

Notice that the *model* is defined to consist of classes that capture the required domain information. We call them **model classes**. Notice also that in this early MVC approach, there is no well-defined concept of a user interface (UI). The 'view' is defined as comprising the output side of a UI, only, while the user input side is separated from it and subsumed under the term 'controller'. This does not reflect how a UI is really organized: by combining certain forms of application output with certain forms of user input like two sides of the same coin. A general UI concept includes both the output (the information output provided to the user, as well as system actions) and the input (including information input provided, as well as actions performed, by the user).

The original Smalltalk MVC metaphor was developed for (monochromatic) text-screen-based user interfaces with no general notion of UI events. This may explain why they did not consider an integral concept of a UI. While they distinguished between the state of objects in the model and their state in the UI, which are both in the scope of a user session, they did not consider the distinction between the model state and the database state.

In his web essay GUI Architectures [http://martinfowler.com/eaaDev/uiArchs.html] (2006), Martin Fowler summarizes the main principles of the original MVC approach in the following way:

1. Separation between UI and model.

2. Divide UI into a 'controller' and 'view'.

3. Views are synchronized with the model (by means of a data binding mechanism).

While the first and third principles are fundamental for the architecture of software applications, the second principle has just a historic meaning and was soon abandoned by the developers of Smalltalk.

Compared to the 1980s, computers, human-computer interaction and software application architecture have evolved. In particular, the establishment of the web as the pre-dominant computing platform has made web browsers to be the most important infrastructure for user interfaces.

The MVC terminology is still widely used today, especially by web application frameworks, but with different meanings ascribed to "M", "V" and "C". Typically, the "view" denotes the HTML-forms-based user interface, and the "controller" is in charge of mediating between the "view" and the "model", which is often tightly coupled, via an *object-relational mapping* (ORM) approach, with the underlying SQL database technology, violating the principle of minimizing interdependencies.

For instance, in the *Active Record [http://martinfowler.com/eaaCatalog/activeRecord.html]* paradigm of the influential *Ruby-on-Rails* framework, which has been adopted by many other web application frameworks (such as by *CakePHP*), the "model" is a direct representation of the schema of the underlying database system, where each entity table of the database is represented by a "model" class that inherits data manipulation methods for performing Create/Retrieve/Update/Delete (CRUD) operations. In this table-to-model-class mapping approach, the "model" depends on the schema of the underlying database and is therefore tightly coupled with the underlying ORM data storage technology. While this may be a suitable approach for a *database-first* development methodology, where an SQL database is the foundation of an application, it is certainly not a general approach and it turns the *model* into a secondary asset.

Also in frameworks based on ORM annotations, such as *JavaEE* with *JPA* annotations, the C# framework *ASP.NET MVC* with *Entity Framework* and *Data Annotations*, or the PHP framework *Symfony* with *Doctrine* annotations, the "model" is coupled with the underlying ORM technology through the ORM annotations woven into the model code, thus making the model dependent on the ORM technology used. All these frameworks use the *Data Mapper [http://martinfowler.com/eaaCatalog/dataMapper.html]* approach for performing CRUD operations based on ORM annotations.

1.2. The Onion architecture metaphor

The term “onion architecture” was coined by Jeffry Palermo in a series of blog posts [http://jeffreypalermo.com/blog/the-onion-architecture-part-1/] in 2008. The main principles of this architecture metaphor are (1) to use a hierarchy of dependencies, where less fundamental (or central) parts depend on more fundamental parts, but never the other way around, and (2) the most fundamental part is the *model*, which implements the application's *data model* in the form of *model classes* while *data storage* is a separate and less fundamental part that must not be coupled with the *model*.

In fact, Palermo and his followers put a lot more into this architecture metaphor, such as using "repository interfaces" and "service interfaces", but I don't think that's really essential for the onion metaphor. Also, they are using a different terminology. When they are using the term "domain model" instead of simply *model*, they are confusing the term "domain model" with "implementation of *data mode"l*, which is what *model classes* do. A *data model* is derived from an *information design model*, which may itself be derived from a *domain information model*. This is the basic development chain in model-based software engineering.

In principle, a *Data Mapper* approach, if it is not based on a platform-specific ORM (annotation) technology, but rather on some form of platform-independent mapping logic, can be used for storage management in an onion architecture.

1.3. The Model-Storage-View-Controller (MSVC) architecture pattern

The mODELcLASSjs architecture is based on a pattern called *Model-Storage-View-Controller (MSVC)*, which combines the MVC metaphor with the onion metaphor.

It should be clear that the three most important parts of any software application involving data management are:

1. the ***model** classes*, which implement the application's data model, defining data structures and constraints;
2. the *data **storage** system*, which is typically, but not necessarily, an SQL database system;
3. the **user interface** (UI), including both information provision (or output) to the user, e.g., on the computer screen, and user input provided by user actions in the form of UI events, e.g., keyboard or mouse events, such that all required user interactions are supported.

The MSVC architecture pattern follows the basic principles of the onion architecture metaphor by separating the *model* layer not only from the UI layer, but also from the *data storage* layer, and by making the *model* the most fundamental part, which must not depend on any other part. The MSVC architecture pattern also follows the good parts of the MVC architecture metaphor and its widely used terminology by using the term *view* interchangeably with *user interface (UI)*, and the term *controller* for denoting the glue code layer needed for integrating the UI code with the underlying model classes and storage management code, or, in MVC jargon, for integrating the *view* with the *model*.

Using a model-based approach, **the model classes of an app are obtained by encoding the app's data model**, which is typically expressed in the form of a UML class diagram. Since it is the task of the *model* and *data storage* layers to define and validate constraints and to manage persistent data, we need reusable model code taking care of this in a generic manner for avoiding per-class and per-property boilerplate code for constraint validation, and per-class boilerplate code for data storage management. mODELcLASSjs provides

1. a generic **check** method for validating property constraints, and
2. the generic storage management methods **add** and **retrieve**, **update** and **destroy** for creating new persistent objects (or records) and for retrieving, updating and deleting existing ones.

2. A Quick Tour of mODELcLASSjs

For using the functionality of mODELcLASSjs in your app, you have to include its code, either by downloading mODELcLASS.js [https://bitbucket.org/gwagner57/entitytypejs/downloads] to the lib folder and use a local script loading element like the following:

```
<script src="lib/mODELcLASSjs.js"></script>
```

or with the help of a remote script loading element like the following:

```
<script src="http://web-engineering.info/tech/JsMODELvIEW/mODELcLASS.js"></script>
```

Then you can create your app's model classes (with property and method declarations) as instances of the meta-class mODELcLASS:

```
Book = new mODELcLASS({
  name: "Book",
  properties: {
    isbn: {range:"NonEmptyString", isStandardId: true, label:"ISBN", pattern:/\b\d{9}(\d|X)\b/,
        patternMessage:'The ISBN must be a 10-digit string or a 9-digit string followed by "X"!'},
    title: {range:"NonEmptyString", min: 2, max: 50, label:"Title"},
    year: {range:"Integer", min: 1459, max: util.nextYear(), label:"Year"}
  },
  methods: {
    ...
  }
});
```

Notice that the declaration of a property includes the constraints that apply to it. For instance, the declaration of the property `isbn` includes a pattern constraint requiring that the ISBN must be a 10-digit string or a 9-digit string followed by "X".

After defining a model class, you can create new 'model objects' instantiating it by invoking its `create` method pre-defined by mODELcLASS:

```
var book1 = Book.create({isbn:"006251587X", title:"Weaving the Web", year: 2000});
```

You can then apply the following properties and methods, all pre-defined by mODELcLASS

1. the property `type` for retrieving the object's direct type,
2. the method `set(` *prop*, *val* `)` for setting an object property after checking all property constraints,
3. the method `isInstanceOf(` *Class* `)` for testing if an object is an instance of a certain model class.
4. the method `toString()` for serializing an object,
5. the method `toRecord()` for converting an object to a record,

The use of these mODELcLASS features is illustrated by the following examples:

```
console.log( book1.type.name);  // "Book"
book1.set("year", 1001);  // "IntervalConstraintViolation: Year must not be smaller than 1459"
book1.set("year", 2001);  // change the year to 2001
console.log( book1.isInstanceOf( Book));  // true
console.log( book1.toString());  // "Book{ isbn:"006251587X", ...}"
```

You can also invoke the generic check method provided by mODELcLASS in the user interface code. For instance, for responsive validation with the HTML5 constraint validation API, you may do the following:

```
var formEl = document.forms["Book"];
formEl.isbn.addEventListener("input", function () {
  formEl.isbn.setCustomValidity(
      Book.check( "isbn", formEl.isbn.value).message);
});
```

Here we define an event handler for `input` events on the ISBN input field. It invokes the `setCustomValidity` method of the HTML5 constraint validation API for setting a validation error message that results from invoking `Book.check` for validating the constraints defined for the `isbn` property for the user input value from the form field `formEl.isbn`. The `check` method returns a constraint violation object with a message property. If no constraint is violated, the message is an empty string, so nothing happens. Notice that you don't have to write the code of the `check` method, as it is pre-defined by mODELcLASSjs.

For managing data storage, a sTORAGEmANAGER object has to be created first:

```
storageManager = new sTORAGEmANAGER();
```

By default, when providing no argument in the invocation of `new sTORAGEmANAGER()`, data is stored locally with the help of JavaScript's *Local Storage* API in the form of 'stringified' JSON tables.

With the help of the created storage manager, you can then, for instance, create new model objects and store them using the storage manager's `add` method, like so,

```
storageManager.add( Book, {isbn:"006251587X",
    title:"Weaving the Web", year: 2000});
```

Notice that the first parameter of all data management methods is the model class concerned, which is `Book` in the example code.

3. Model-Based Development

We must not confuse the term *model* as used in the MVC metaphor, and adopted by many web application frameworks, and the term *model* as used in UML and other modeling languages. While the former refers to the model classes of an app, the latter refers to the concept of a model either as a simplified description of some part of the real world, or as a design blueprint for construction.

In model-based engineering, models are the basis for designing and implementing a system, no matter if the system to be built is a software system or another kind of complex system such as a manufacturing machine, a car, or an organisation.

In model-based software development, we distinguish between three kinds of models:

1. solution-independent **domain models** describing a specific part of the real-world and resulting from requirements and domain engineering in the system analysis, or inception, phase of a development project;

2. platform-independent **design models** specifying a logical system design resulting from the design activities in the elaboration phase;

3. platform-specific **implementation models** as the result of technical system design in the implementation phase.

Domain models are the basis for designing a software system by making a platform-independent *design model*, which is in turn the basis for implementing a system by making an *implementation model* and encoding it in the language of the chosen platform. Concerning information modeling, we first make a (conceptual) *domain information model*, then we derive an **information design model** from it, and finally map the information design model to a **data model** for the target platform. We then code this data model in the form of **model classes**, which are the basis for any data management user interface.

mODELcLASSjs facilitates model-based app development by allowing a direct encoding of an information design model, including subtype/inheritance relationships.

4. The Philosophy and Features of mODELcLASSjs

The concept of a **class** is fundamental in *object-oriented* programming. Objects *instantiate* (or *are classified by*) a class. A class defines the properties and methods for the objetcs that instantiate it.

There is no explicit class concept in JavaScript. However, classes can be defined in two ways:

1. In the form of a **constructor** function that allows to create new instances of the class with the help of the `new` operator. This is the classical approach recommended in the Mozilla JavaScript documents.

2. In the form of a **factory** object that uses the predefined `Object.create` method for creating new instances of the class.

Since we normally need to define class hierarchies, and not just single classes, these two alternative approaches cannot be mixed within a class hierarchy, and we have to make a choice whenever we build an app. With mODELcLASSjs, we choose the second approach with the following benefits:

1. Properties are declared (with a property label, a range and many other constraints).

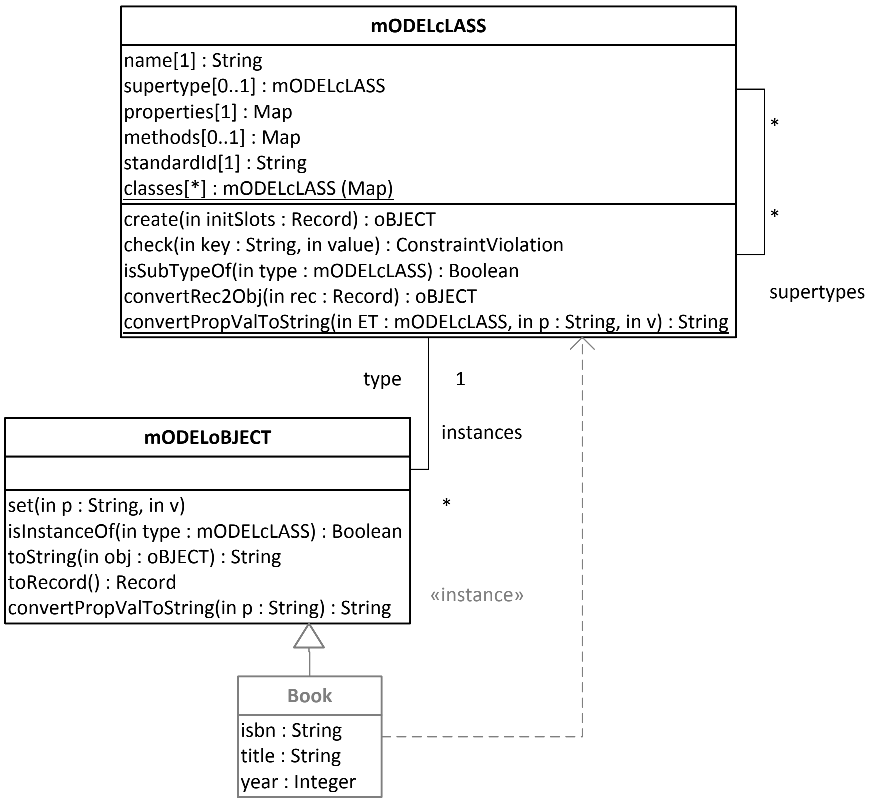
2. Multiple inheritance and multiple classification are supported.

3. Object pools are supported.

These benefits come with a price: objects are created with lower performance, mainly due to the fact that `Object.create` is slower than `new`. On the other hand, for apps with lots of object creation and destruction, such as games and simulations, mODELcLASSjs provides object pools for avoiding performance problems due to garbage collection.

The properties and methods of the meta-class mODELcLASS, and the properties and methods it injects in any model class created with its `create` method, are listed in the following class diagram:

Figure 1.1. The meta-class mODELcLASS


mODELcLASS
name[1] : String
supertype[0..1] : mODELcLASS
properties[1] : Map
methods[0..1] : Map
standardId[1] : String
classes[*] : mODELcLASS (Map)
create(in initSlots : Record) : oBJECT
check(in key : String, in value) : ConstraintViolation
isSubTypeOf(in type : mODELcLASS) : Boolean
convertRec2Obj(in rec : Record) : oBJECT
convertPropValToString(in ET : mODELcLASS, in p : String, in v) : String
*
*
supertypes
type
1
instances
*
mODELoBJECT
set(in p : String, in v)
isInstanceOf(in type : mODELcLASS) : Boolean
toString(in obj : oBJECT) : String
toRecord() : Record
convertPropValToString(in p : String) : String
«instance»
Book
isbn : String
title : String
year : Integer

Notice that in the diagram, we use the (imaginary) type `mODELoBJECT` for showing the pre-defined property `type` and several methods pre-defined for all objects that instantiate a class created with mODELcLASS. The class `Book` is an example of a `mODELcLASS`. As a subclass of `mODELoBJECT`, it inherits its property `type` and all its methods.

The values of the pre-defined property `instances` are maps of `mODELoBJECT`s representing the extension (or population) of a model class.

5. The Check Method

Any mODELcLASS has a class-level `check` method for validating all kinds of property constraints. Since this method can validate all properties of a model class, it takes as its first parameter the name of the property, and as its second parameter the value to be validated:

```
mODELcLASS.prototype.check = function (prop, val) {
  var propDeclParams = this.properties[prop],
      range = propDeclParams.range,
      min = propDeclParams.min,
      max = propDeclParams.max,
      minCard = propDeclParams.minCard,
      maxCard = propDeclParams.maxCard,
      pattern = propDeclParams.pattern,
      msg = propDeclParams.patternMessage,
      label = propDeclParams.label || prop;
```

Notice that the definition of a model class comes with a set of property declarations in `properties`. An example of such a property declaration is the declaration of the attribute `title`:

```
title: {range:"NonEmptyString", min: 2, max: 50, label:"Title"},
```

In this example we have the property declaration parameters `range`, `min`, `max` and `label`. In lines 2-10 above, all these property declaration parameters are copied to local variables for having convenient shortcuts.

In the `check` method, the first check is concerned with **mandatory value constraints**:

```
  if (!propDeclParams.optional && val === undefined) {
      return new MandatoryValueConstraintViolation("A value for "+
          label +" is required!");
  }
```

The next check is concerned with **range constraints**:

```
  switch (range) {
  case "String":
    if (typeof( val) !== "string") {
      return new RangeConstraintViolation("The "+ label +
          " must be a string!");
    }
    break;
  case "NonEmptyString":
    if (typeof(val) !== "string" || val.trim() === "") {
      return new RangeConstraintViolation("The "+ label +
          " must be a non-empty string!");
    }
    break;
  ...  // other cases
  case "Boolean":
    if (typeof( val) !== "boolean") {
      return new RangeConstraintViolation("The value of "+ label +
          " must be either 'true' or 'false'!");
    }
    break;
  }
```

Then there are several range-specific checks concerning (1) **string length constraints** and **pattern constraints**:

```
if (range === "String" || range === "NonEmptyString") {
  if (min !== undefined && val.length < min) {
    return new StringLengthConstraintViolation("The length of "+
        label + " must be greater than "+ min);
  } else if (max !== undefined && val.length > max) {
    return new StringLengthConstraintViolation("The length of "+
        label + " must be smaller than "+ max);
  } else if (pattern !== undefined && !pattern.test( val)) {
      return new PatternConstraintViolation( msg || val +
          "does not comply with the pattern defined for "+ label);
  }
}
```

and (2) **interval constraints**:

```
if (range === "Integer" || range === "NonNegativeInteger" ||
    range === "PositiveInteger") {
  if (min !== undefined && val < min) {
    return new IntervalConstraintViolation( label +
        " must be greater than "+ min);
  } else if (max !== undefined && val > max) {
    return new IntervalConstraintViolation( label +
        " must be smaller than "+ max);
  }
}
```

Then the next check is concerned with **cardinality constraints**, which may apply to list-valued or map-valued properties.

```
if (minCard !== undefined &&
    (Array.isArray(val) && val.length < minCard ||
     typeof(val)==="object" && Object.keys(val).length < minCard)) {
  return new CardinalityConstraintViolation(
      "A set of at least "+ minCard +" values is required for "+ label);
}
if (maxCard !== undefined &&
    (Array.isArray(val) && val.length > maxCard ||
     typeof(val)==="object" && Object.keys(val).length > maxCard)) {
  return new CardinalityConstraintViolation("A value set for "+ label +
      " must not have more than "+ maxCard +" members!");
}
```

The next check is concerned with **uniqueness constraints**, which can only be checked by inspecting the entire population of the model class. Assuming that this population has been loaded into the main memory collection *modelclass*`.instances`, the following code is used:

```
if (propDeclParams.unique && this.instances) {
  keys = Object.keys( this.instances);
  for (i=0; i < keys.length; i++) {
    if ( this.instances[keys[i]][prop] === val) {
      return new UniquenessConstraintViolation("There is already a "+
          this.name +" with a(n) "+ label +" value "+ val +"!");
    }
  }
}
```

Finally, the *mandatory value constraints* and the *uniqueness constraints* implied by a **standard identifier declaration** are checked:

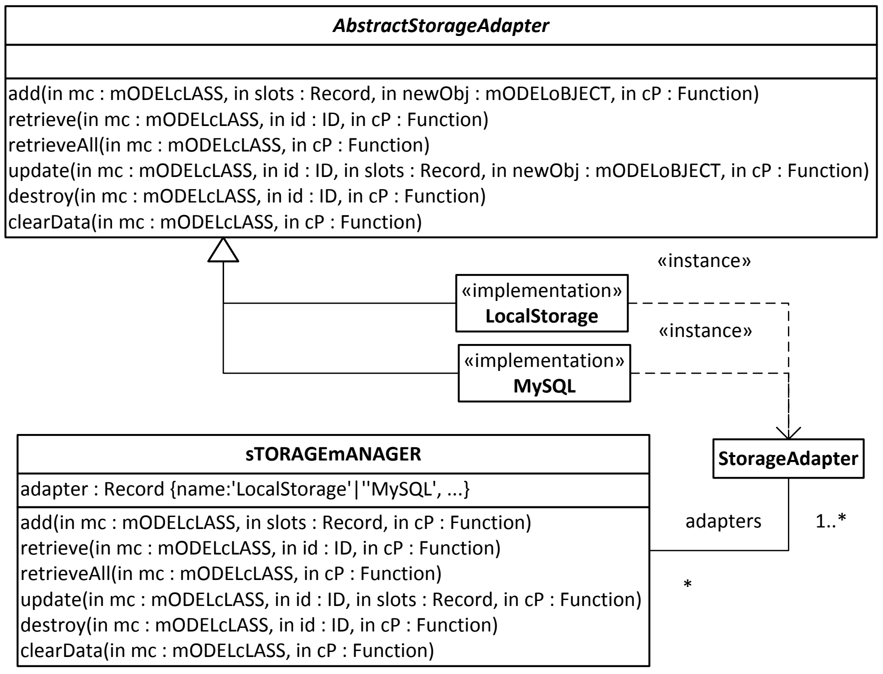
```
if (propDeclParams.isStandardId) {
  if (val === undefined) {
    return new MandatoryValueConstraintViolation("A value for the " +
        "standard identifier attribute "+ label +" is required!");
  } else if (this.instances && this.instances[val]) {
    return new UniquenessConstraintViolation("There is already a "+
        this.name +" with a(n) "+ label +" value "+ val +"!");
  }
}
```

6. Storage Management

As explained above in Section 1.3, “The Model-Storage-View-Controller (MSVC) architecture pattern”, mODELcLASSjs is using a *data mapper* approach for separating model classes and storage management. A mODELcLASSjs application creates a sTORAGEmANAGER object at application start-up time, typically in some piece of controller code, by providing parameter values for a storage adapter (such as local storage or back-end storage with MySQL).

The storage manager has a list of available storage adapters, one of which is assigned as the currently used adapter. Each available storage adapter (such as `LocaStorage` or `MySQL` in the diagram below) implements the CRUD methods specified in the abstract class `AbstractStorageAdapter`.

Figure 1.2. The sTORAGEmANAGER classes


AbstractStorageAdapter
add(in mc : mODELcLASS, in slots : Record, in newObj : mODELoBJECT, in cP : Function)
retrieve(in mc : mODELcLASS, in id : ID, in cP : Function)
retrieveAll(in mc : mODELcLASS, in cP : Function)
update(in mc : mODELcLASS, in id : ID, in slots : Record, in newObj : mODELoBJECT, in cP : Function)
destroy(in mc : mODELcLASS, in id : ID, in cP : Function)
clearData(in mc : mODELcLASS, in cP : Function)
«instance»
«implementation» LocalStorage
«instance»
«implementation» MySQL
sTORAGEmANAGER
adapter : Record {name:'LocalStorage'|''MySQL', ...}
add(in mc : mODELcLASS, in slots : Record, in cP : Function)
retrieve(in mc : mODELcLASS, in id : ID, in cP : Function)
retrieveAll(in mc : mODELcLASS, in cP : Function)
update(in mc : mODELcLASS, in id : ID, in slots : Record, in cP : Function)
destroy(in mc : mODELcLASS, in id : ID, in cP : Function)
clearData(in mc : mODELcLASS, in cP : Function)
StorageAdapter
adapters
1..*
*

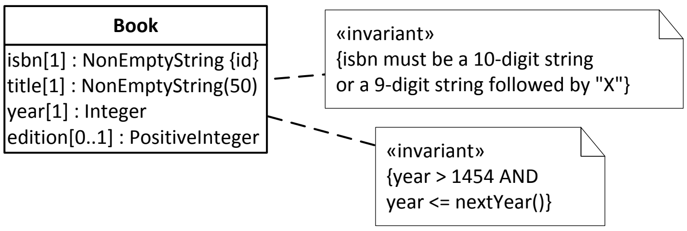
Chapter 2. Constraint Validation with mODELcLASSjs

In this part of the tutorial, we show how to build a simple app with constraint validation using the mODELcLASSjs library for avoiding boilerplate model code. Compared to the app discussed in the plain JavaScript validation app tutorial [http://web-engineering.info/tech/JsFrontendApp/validation-tutorial.html], we deal with the same issues: showing 1) how to **define constraints** in a model class, 2) how to **perform responsive validation** in the user interface based on the constraints defined in the model classes.

The main difference when using mODELcLASSjs is that defining constraints becomes much simpler. Since mODELcLASSjs provides a generic method for checking property constraints, the property-specific check methods for checking property constraints are no longer needed. Since constraints are defined in a purely declarative manner, their textual encoding corresponds directly to their expression in the information design model. This implies that we can directly code the information design model without first creating a data model from it.

As in other tutorials, the purpose of our app is to manage information about books. The information items and constraints are described in the information design model shown in Figure 2.1 below.

Figure 2.1. A platform-independent design model with the class `Book` and two invariants


Book
isbn[1] : NonEmptyString {id}
title[1] : NonEmptyString(50)
year[1] : Integer
edition[0..1] : PositiveInteger
«invariant»
{isbn must be a 10-digit string
or a 9-digit string followed by "X"}
«invariant»
{year > 1454 AND
year <= nextYear()}

1. Encoding the Design Model

We now show how to code the ten integrity constraints defined by the design model shown in Figure 2.1 above.

1. For the first three of the four properties defined in the `Book` class, we have a *mandatory value constraint*, indicated by the multiplicity expression [1]. However, since properties are mandatory by default in mODELcLASSjs, we don't have to code anything for them. Only for the property `edition`, we need to code that it is optional with the key-value pair `optional: true`, as shown in the `edition` property declaration in the class definition below.

2. The `isbn` attribute is declared to be the *standard identifier* of `Book`. We code this (and the implied uniqueness constraint) in the `isbn` property declaration with the key-value pair `isStandardId: true`, as shown in the class definition below.

3. The `isbn` attribute has a *pattern constraint* requiring its values to match the ISBN-10 format that admits only 10-digit strings or 9-digit strings followed by "X". We code this with the key-value

pair `pattern:/\b\d{9}(\d|X)\b/` and the special constraint violation message defined by `patternMessage:"The ISBN must be a 10-digit string or a 9-digit string followed by 'X'!"`.

4. The `title` attribute has an *string length constraint* with a maximum of 50 characters. This is coded with `max: 50`.

5. The `year` attribute has an *interval constraint* with a minimum of 1459 and a maximum that is not fixed, but provided by the utility function `nextYear()`. We can code this constraint with the key-value pairs `min: 1459` and `max: util.nextYear()`.

6. Finally, there are four *range constraints*, one for each property. We code them with corresponding key-value pairs, like `range:"NonEmptyString"`.

This leads to the following definition of the model class `Book` :

```
Book = new mODELcLASS({
  name: "Book",
  properties: {
    isbn: {range:"NonEmptyString", isStandardId: true, label:"ISBN", pattern:/\b\d{9}(\d|X)\b/,
        patternMessage:"The ISBN must be a 10-digit string or a 9-digit string followed by 'X'!"},
    title: {range:"NonEmptyString", max: 50},
    year: {range:"Integer", min: 1459, max: util.nextYear()},
    edition: {range:"PositiveInteger", optional: true}
  }
});
```

For such a model class definition, mODELcLASSjs provides generic data management operations (`Book.add`, `Book.update`, `Book.destroy`, etc.) as well as property checks and setters (`Book.check` and `bookObject.set`).

2. Project Set-Up

The MVC folder structure of this project is the same as discussed in the plain JavaScript validation app tutorial [http://web-engineering.info/tech/JsFrontendApp/validation-tutorial.html]. Also, the same library files are used.

The start page of the app first takes care of the page styling by loading `normalize.css` and our `main.css` file with the help of the two `link` elements (in lines 6 and 7), then it loads the mODELcLASSjs library file, the app initialization script `initialize.js` from the `src/ctrl` folder and the model class `Book.js` from the `src/model` folder.

Figure 2.2. The mODELcLASSjs validation app's start page `index.html`.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <meta charset="UTF-8" />
  <title>mODELcLASSjs Validation App</title>
  <link rel="stylesheet" type="text/css" href="../css/normalize.css" />
  <link rel="stylesheet" type="text/css" href="../css/main.css" />
  <script src="../mODELcLASSjs.min.js"></script>
  <script src="src/ctrl/initialize.js"></script>
  <script src="src/model/Book.js"></script>
</head>
<body>
  <h1>Example: Public Library</h1>
  <h2>mODELcLASS Validation App</h2>
  <p>This app supports the following operations:</p>
  <menu>
    <li><a href="listBooks.html">
         <button type="button">List all books</button></a></li>
    <li><a href="createBook.html">
         <button type="button">Add a new book</button></a></li>
    <li><a href="updateBook.html">
         <button type="button">Update a book</button></a></li>
    <li><a href="deleteBook.html">
         <button type="button">Delete a book</button></a></li>
    <li><button type="button" onclick="Book.clearData()">
         Clear database</button></li>
    <li><button type="button" onclick="Book.createTestData()">
         Create test data</button></li>
  </menu>
</body>
</html>
```

The app initialization script `initialize.js` first defines the app's namespaces and then creates a local storage manager. It also defines a method for creating test data:

```
// main namespace pl = "public library"
var pl = {model:{}, view: {}, ctrl:{}};
// define a localStorage manager
pl.ctrl.storageManager =  new sTORAGEmANAGER();
/**
 *  Create and save test data
 */
pl.ctrl.createTestData =  function () {
  pl.ctrl.storageManager.add( Book, {isbn:"006251587X",
      title:"Weaving the Web", year: 2000, edition: 2});
  pl.ctrl.storageManager.add( Book, {isbn:"0465026567",
      title:"Gödel, Escher, Bach", year: 1999});
  pl.ctrl.storageManager.add( Book, {isbn:"0465030793",
      title:"I Am A Strange Loop", year: 2008});
};
```

3. The View and Controller Layers

The user interface (UI) is the same as explained in the plain JavaScript validation app tutorial [http://web-engineering.info/tech/JsFrontendApp/validation-tutorial.html]. There is only one difference. For responsive constraint validation, where input event handlers are used to check constraints on user input, now the generic `Book.check` function is used, as shown in the following code fragment of the `setupUserInterface` method from `src/view/createBook.js`:

```
pl.view.createBook = {
  setupUserInterface: function () {
    var formEl = document.forms['Book'],
        submitButton = formEl.commit;
    submitButton.addEventListener("click",
        this.handleSubmitButtonClickEvent);
    formEl.isbn.addEventListener("input", function () {
        formEl.isbn.setCustomValidity(
            Book.check("isbn", formEl.isbn.value).message);
    });
    formEl.title.addEventListener("input", function () {
        formEl.title.setCustomValidity(
            Book.check("title", formEl.title.value).message);
    });
   ...
  },
};
```

While the validation on user input enhances the usability of the UI by providing immediate feedback to the user, validation on form submission is even more important for catching invalid data. Therefore, the event handler `handleSubmitButtonClickEvent()` performs the property checks again, as shown in the following program listing:

```
handleSaveButtonClickEvent: function () {
  var formEl = document.forms['Book'], slots = {};
  // create error messages in case of constraint violations
  Object.keys( Book.properties).forEach( function (prop) {
    var errMsg="";
    slots[prop] = formEl[prop].value;
    errMsg = Book.check( prop, slots[prop]).message;
    formEl[prop].setCustomValidity( errMsg);
  });
  // save the input data only if all of the form fields are valid
  if (formEl.checkValidity()) {
    pl.ctrl.storageManager.add( Book, slots);
  }
}
```

4. Run the App and Get the Code

You can run the mODELcLASS validation app [mODELcLASS-ValidationApp/index.html] from our server or download the code [mODELcLASS-ValidationApp.zip] as a ZIP archive file.

5. Concluding Remarks

After eliminating the repetitive code structures (called *boilerplate code*) needed in the model layer for constraint validation and for the data storage management methods, there is still a lot of boilerplate code needed in the UI. In a follow-up article of our tutorial series, we will present an approach how to avoid this UI boilerplate code.