

<b>Model Engineering Lab</b> 188.923 IT/ME VU, WS 2015/16	<b>Assignment 4</b>
<b>Deadline:</b> Upload (ZIP) in TUWEL until Sunday, January 10 <sup>th</sup> , 2016, 23:55 Assignment Review: Wednesday, January 13 <sup>th</sup> , 2016	<b>25 Points</b>


## Code Generation




The goal of this assignment is to develop code generators for the *Domain Views Modeling Language (DVML)* with Xtend that produce AngularJS and HTML code providing Web applications for managing business data modeled with a DVML models.

Therefore, in Part A, you have to develop an **AngularJS code generator** for DVML domain models generating AngularJS code that implements data management operations for the classes defined in DVML domain models.

In Part B, you have to develop an **HTML code generator** for DVML view models generating HTML code that implements Web interfaces according to DVML view models. Furthermore, AngularJS expressions are added to the generated HTML code to bind the AngularJS code generated in Part A to the Web interfaces such that class instances can be managed via the Web interfaces.

## Assignment Resources

 ME\_WS15\_Lab4\_Resources.zip

-  at.ac.tuwien.big.views (Modeling project with solution of Assignment 1)
-  at.ac.tuwien.big.views.gen (Xtend/Xtext project for developing the HTML and Angular JS code generators)
-  lab4.pdf (this document)

Before starting this assignment, make sure that you have all necessary components installed in your Eclipse. A detailed installation guide can be found in the TUWEL course.

It is recommended that you read the complete specification at least once. If there are any parts of the specification or the provided resources that are ambiguous to you, don't hesitate to ask in the TUWEL forum for clarification.

## Part A: AngularJS Code Generation

In this part you have to implement an Xtend code generator for DVML that generates valid AngularJS<sup>1</sup> code from DVML domain models. The generated code implements data management operations for classes defined in DVML domain models, such as saving a new class instance and updating an existing class instance.

The AngularJS code that has to be generated is in the following described based on the exemplary DVML domain model provided as part of the assignment resources, namely the domain model `instituteDomainModel.views` located in the folder `at.ac.tuwien.big.views.gen/input`. For a better understanding, some parts of the example code are highlighted in **light green** to indicate that the respective code has been generated from elements of a DVML domain model.

Figure 1 shows the AngularJS code that shall be generated for the exemplary domain model `instituteDomainModel.views`. Note that the complete AngularJS code that should

<sup>1</sup> AngularJS Documentation: <https://docs.angularjs.org>

be generated for this example model is provided in the assignment resources in the file `institute.js` located in `at.ac.tuwien.big.views.gen/expected_output`.

In AngularJS, application logic is organized into modules. For a DVML domain model, one module is generated that is named after the first class defined in the domain model. In the example, the first defined class is named “Institute”, thus, a module “`instituteApp`” is generated (line 1). The module generated for a domain model comprises services (lines 3-5) and controllers (lines 7-9) that provide data management operations for the classes defined in a domain model. They are described in the following.

**Figure 1: Excerpt of the generated AngularJS code**

```
1 var module = angular.module('instituteApp', []);
2
3 module.service('instituteService', function () {
4     //add services here
5 });
6
7 module.controller('instituteController', function ($scope, instituteService) {
8     //add controllers here
9 });
```

## Services

For each class defined in a DVML domain model, operations for saving (save), retrieving (get), deleting (delete), and listing (list) instances of the class have to be generated. Additionally, variables for holding all instances of a particular class, and variables for generating ids for new instances of classes have to be generated.

The example below shows the service operations generated for the class “Course” defined in the example DVML domain model `instituteDomainModel.views`.

Example
<pre>// variable holding all courses var courses = [];  // variable for generating course ids var courseid = 0;  //save course this.saveCourseService = function (course) {     if (course.id == null) {         course.id = courseid++;         courses.push(course);     } else {         for (i in courses) {             if (courses[i].id == course.id) {                 courses[i] = course;             }         }     } }  //get course by id this.getCourseService = function (id) {     for (i in courses) {         if (courses[i].id == id) {             return courses[i];         }     } }</pre>

```

//delete course by id
this.deleteCourseService = function (id) {
  for (i in courses) {
    if (courses[i].id == id) {
      courses.splice(i, 1);
    }
  }
}

//list all courses
this.listCourseService = function () {
  return courses;
}

```

## Controllers

All the service operations generated for a DVML domain model have to be bound to the Angular scope object `$scope`. Furthermore, for each class a navigation service operation has to be generated that enables the navigation between different views.

The example below shows the controller code generated for the class “Course” defined in the example DVML domain model `instituteDomainModel.views`.

### Example

```

//get courses
$scope.courses = instituteService.listCourseService();

//save course
$scope.saveCourse = function () {
  instituteService.saveCourseService($scope.newcourse);
  $scope.newcourse = {};
}

//delete course
$scope.deleteCourse = function (id) {
  instituteService.deleteCourseService(id);
}

//update course
$scope.updateCourse = function (id) {
  $scope.newcourse = angular.copy(instituteService.getCourseService(id));
}

//get course
$scope.getCourse = function (id) {
  $scope.course = angular.copy(instituteService.getCourseService(id));
}

//navigation from course views to other views
$scope.navigationCourse = function (targetView) {
  $(".container").hide();
  var view = angular.element('#'+targetView);
  view.show();
}

```

## Part B: HTML Code Generation

In this part you have to implement an HTML code generator for DVML that generates valid and interactive HTML code from DVML view models providing a Web interface for managing the data defined in DVML domain models according to the views defined in DVML view models. In particular, the generated HTML documents have to enable the navigation between views, the entering of data (class instances) by means of input fields, the checking of mandatory input data (properties), the verification of regular expressions, and the evaluation of conditions for making element visible, invisible, enabled, or disabled interactively based to the fulfillment of the condition.

The HTML code that has to be generated is in the following described based on the exemplary DVML view model `instituteViewModel.views` provided as part of the assignment resources (`at.ac.tuwien.big.views.gen/input`). Again, for a better understanding, some parts of the example code are highlighted in **light green** to indicate that the respective code has been generated from elements of a DVML model. Note that the complete HTML code that should be generated for the example model `instituteViewModel.views` is provided in the assignment resources in the file `institute.html` located in `at.ac.tuwien.big.views.gen/expected_output`.

Figure 2 shows the structure of the HTML code that has to be generated. In the head of the HTML file, first the JavaScript libraries and CSS files to be used by the HTML code have to be specified (line 6). Thereafter, JavaScript code has to be generated that registers the welcome view group serving as entry point of the web interface (line 12). In the body of the HTML file, you have to generate HTML elements according to the views defined in DVML view models (line 20).

**Figure 2: Excerpt of the generated HTML code**

```
1 <!DOCTYPE html>
2 <html lang="en" data-ng-app="instituteApp">
3 <head>
4   <title>Views</title>
5
6   //JavaScript libraries and CSS files
7
8   <script type="text/javascript">
9     $(document).ready(
10       function(){
11
12         //register welcome view group here
13
14         view.init();
15       });
16   </script>
17 </head>
18 <body data-ng-controller="instituteController">
19
20   //add HTML elements here
21
22 </body>
23 </html>
```

### JavaScript Libraries and CSS File

The code for generating the declarations of the used JavaScript libraries and CSS files is already provided in the HTML code generator `View2HTMLGenerator.xtend` located in the provided project `at.ac.tuwien.big.views.gen`. For testing the generated HTML code in the

browser, a connection to the Internet is required because some JavaScript libraries and CSS files are online resources.

## Registration of Welcome View Group

You have to declare the view that should be shown as first page of the Web interface by calling the JavaScript operation `addWelcomePage()`. This view is the start view of the welcome view group defined the DVML view model.

In the example `instituteViewModel.views`, the welcome view group is the view group named “Institute” and the view named “CreateInstitute” serves as start view of this view group. Thus the view “CreateInstitute” is registered as welcome page by passing its name to the operation `addWelcomePage()` (whitespaces in the name have to be removed).

Example
<pre>view.addWelcomePage('CreateInstitute');</pre>

## HTML Elements

For the views defined in a DVML view model, you have to generate suitable HTML code as described in the following based on examples.

### View Groups

First of all, a navigation bar has to be generated that allows the navigation between the view groups defined in a DVML view model. In particular, for each start view of a view group, a navigation bar entry has to be generated. A navigation bar entry consists of an `<li>` element containing an `<a>` element. The `target` of the `<a>` element corresponds to the name of the start view (whitespaces are removed), and its content corresponds to the name of the view group containing the start view.

Example
<pre>&lt;nav class="navbar navbar-default"&gt;   &lt;div class="container-fluid"&gt;     &lt;div&gt;       &lt;ul class="nav navbar-nav"&gt;         &lt;li&gt;&lt;a href="" class="viewgroup" target="CreateInstitute"&gt;Institute&lt;/a&gt;&lt;/li&gt;         &lt;li&gt;&lt;a href="" class="viewgroup" target="CourseIndex"&gt;Courses&lt;/a&gt;&lt;/li&gt;       &lt;/ul&gt;     &lt;/div&gt;   &lt;/div&gt; &lt;/nav&gt;</pre>

### Class Index Views

For a class index view, a `<div>` element has to be generated with the class "container" and an id corresponding to the name of the view (whitespaces are removed). Furthermore, an `<h2>` and an `<h3>` heading are generated for the header and the description of the view, respectively. For displaying the class instances, one `<ul>` element containing one `<li>` element is created. The `<li>` element defines the attribute `data-ng-repeat` set to "`<classname>` in `<classname>s`" where `<classname>` corresponds to the name of the class referenced by the view (in lower case letters). Furthermore, the expression `{{<classname>.<idname>}}` is generated as the content of the `<li>` element

where `<idname>` corresponds to the name of the id property defined by the class of the view.

#### Example

```
<div class="container" id="CourseIndex">
  <h2>Courses</h2>
  <h3>List of all current courses</h3>
  <ul>
    <li data-ng-repeat="course in courses">
      {{ course.title }}
      //add links here
    </li>
    //add "add buttons" here
  </ul>
</div>
```

### Read and Delete Views

For read and delete views, a `<div>` element is created with the id `"modal<viewname>"` where `<viewname>` corresponds to the name of the view (whitespaces are removed). This `<div>` elements contains four more `<div>` elements. For the header of the view, an `<h4>` heading is created within the `<div>` element with the class `"modal-header"`. For the description of the view and the name of the view, a `<p>` element and an `<h5>` heading are generated, respectively. Both of them are contained in the `<div>` element with the class `"modal-body"`.

For each property element contained by the view, a `<p>` element is created containing the following text: `"<label>: {{<classname>.<propertyname>}}"` where `<label>` corresponds to the label defined for the property element, `<classname>` corresponds to the name of the class that is referenced by the read/delete view, and `<propertyname>` corresponds to the name of the property referenced by the property element.

#### Example

```
<div class="modal fade" id="modalDeleteCourse">
  <div class="modal-dialog">
    <div class="modal-content">
      <div class="modal-header">
        <h4 class="modal-title">Course</h4>
      </div>
      <div class="modal-body">
        <p>Do you really want to delete this course?</p>
        <h5>Delete Course</h5>
        <p>Title: {{ course.title }}</p>
      </div>
      <div class="modal-footer">
        //add "modal buttons" here
      </div>
    </div>
  </div>
</div>
```

### Create and Update Views

For create and update views, a `<div>` element is created with an id corresponding to the name of the view (whitespaces are removed). For the header of the view, an `<h2>`

heading is generated. Furthermore, a `<form>` element is generated with the name “`<viewname>Form`” where `<viewname>` stands for the name of the view (whitespaces are removed). The `<form>` element contains a `<p>` element for the description of the view, and a `<div>` element with the class “panel-group” that holds the HTML elements generated for the element groups of the view. In case that the layout of the view is a horizontal layout, a `<div>` element with the class “row” has to be generated in addition.

Example
<pre> &lt;div class="container" id="CreateInstitute"&gt;   &lt;h2&gt;Institute&lt;/h2&gt;   &lt;form name="CreateInstituteForm" novalidate&gt;     &lt;p&gt;This is a form for creating institutes.&lt;/p&gt;     &lt;div class="panel-group"&gt;       &lt;div class="row"&gt; //only for views with horizontal layout          //add element groups here        &lt;/div&gt;     &lt;/div&gt;      //add "save button" here    &lt;/form&gt; &lt;/div&gt; </pre>

## Element Groups

For element groups, a `<div>` element is generated. The class of this `<div>` element depends on the layout of the view containing the element group. The `<div>` element contains an `<h4>` heading for the header of the element group and a `<div>` element with the class “panel-body” that contains the HTML elements generated for the view elements of the element group. In case the layout of the element group is a horizontal layout, an additional `<div>` element with the class “form-inline” and the role “form” is generated.

Example
<pre> // for views with vertical layout: &lt;div class="elementgroup" /* add condition */&gt;  // for views with horizontal layout: // &lt;div class="elementgroup col-sm-6" /* add condition */&gt;    &lt;h4&gt;Institute Details&lt;/h4&gt;   &lt;div class="panel-body"&gt;     &lt;div class="form-inline" role="form"&gt; //only for views with horizontal layout        //add view elements here      &lt;/div&gt;   &lt;/div&gt; &lt;/div&gt; </pre>

## Property Elements

For property view elements, suitable HTML element are generated, namely `<input>` elements for short text view elements and date time pickers; `<textarea>` elements for long text view elements; and `<select>` elements for selection view elements. The ids of these elements correspond to the elementIDs of the property view elements. Furthermore, `<label>` elements are generated for each property view element. The `<label>` elements have the attribute for set to the id of the respective element and

contain as text the label of the property view element followed by “:”. The generated `<input>`, `<textarea>`, `<select>`, and `<label>` elements are contained by `<div>` elements with the class “form group”. Furthermore, they define a value for the attribute `data-ng-model` following the following pattern: “new<classname>.<propertyname>” where `<classname>` corresponds to the name of the class referenced by the view containing the property view element and `<propertyname>` corresponds to the name of the property referenced by the property view element.

Text view elements and date time pickers can define a regular expression for specifying the required format of input values. For text view elements and date time pickers defining such a format, the attribute `data-ng-pattern` has to be set for the generated `<input>` or `<textarea>` element following the following pattern: “/<format>/” where `<format>` stands for the format defined by the text view element or data time picker.

A property element might be mandatory. In particular, it is mandatory if the property referenced by the property view element defines a lower bound and upper bound of “1”. To enforce that a value is provided for mandatory properties, the HTML elements generated for the respective property view element (`<input>`, `<textarea>`, or `<select>`) have to define the attribute `required`. Furthermore, the label generated for the respective element has to be extended with the character “\*”.

For selections view elements, a `<select>` element is created as described above. This `<select>` element contains one `<option>` element that defines the attributes `disabled` and `selected`, as well as the text “Select your option”. Furthermore, for each selection item defined by the selection view element, one `<option>` element is added to the `<select>` element. If the selection item is an enumeration literal item, the `value` attribute of the `<option>` element is set to the value of the referenced enumeration literal and the `<option>` element contains as text the name of the referenced enumeration literal. Otherwise, the value of the selection item is used for the `value` attribute and for the text.

For a date time picker, an `<input>` element is created as described above. This `<input>` element is contained by a `<div>` element with the class “input-group date” and the id “picker<elementID>”. The value of the attribute `style` of this `<div>` element depends on the data type of the property referenced by the date time picker. Additionally, a `<span>` element with the class “input-group-addon” containing another `<span>` element is created. The class of the latter `<span>` element again depends on the data type of the property referenced by the date time picker. The `<div>` with the class “form-group” generated for a date time picker is nested within three `<div>` elements.

Example
<pre>//short mandatory text with format string &lt;div class="form-group"&gt;   &lt;label for="01"&gt;Number&lt;span&gt;*&lt;/span&gt;&lt;/label&gt;   &lt;input type="text" class="form-control" id="01" name="number"     data-ng-model="newinstitute.number" required data-ng-pattern="/^[0-9]+\$/"     /* add condition */ /&gt;   //add error span tags &lt;/div&gt;  //long text &lt;div class="form-group"&gt;   &lt;label for="14"&gt;Description:&lt;/label&gt;   &lt;textarea class="form-control" rows="4" id="14" name="description"     data-ng-model="newcourse.description" /* add condition */ &gt;   &lt;/textarea&gt;   //add error span tags &lt;/div&gt;</pre>



```

//mandatory selection for selection items
<div class="form-group">
  <label for="08">Visiting professor<span>*</span></label>
  <select data-ng-option class="form-control" id="08"
    data-ng-model="newprofessor.visitingprofessor" required /* add condition */ >
    <option value="" disabled selected>Select your option</option>
    <option value="true">true</option>
    <option value="false">false</option>
  </select>
  //add error span tags here
</div>

//selection for enumeration literal items
<div class="form-group">
  <label for="11">Type<span>*</span></label>
  <select data-ng-option class="form-control" id="11"
    data-ng-model="newcourse.type" required /* add condition */ >
    <option value="" disabled selected>Select your option</option>
    <option value="LE">Lecture</option>
    <option value="TH">Thesis</option>
  </select>
  //add error span tags here
</div>

//date time picker
<div class="form-group">
  <div class="row">
    <div class="col-xs-6 col-sm-12">
      <div class="form-group">
        <label for="15">Date:</label>
        //pickers for properties with datatype "Date":
        <div class="input-group date" id="picker15" style="calendar">
          /* pickers for properties with datatype "Time":
          <div class='input-group date' id='picker15' style='time'> */
            <input type="text" class="form-control" id="15" name="date"
              data-ng-model="newcourse.date" data-ng-pattern="/dddd, MMMM Do YYYY/"
              /* add condition */ />
            <span class="input-group-addon">
              //pickers for properties with datatype "Date":
              <span class="glyphicon glyphicon-calendar"></span>
              /*pickers for properties with datatype "Time":
              <span class="glyphicon glyphicon-time"></span>*/
            </span>
          </div>
        </div>
      </div>
    </div>
  </div>
  //add error span tags here
</div>

```

## Association Elements

For an association element (list or table) a <div> elements with the class "form-group" and an <h5> heading with the label of the association element are generated. For a list a <ul> element is created and for a table a <table> element is created. These <ul> and <table> elements define as id the elementId of the respective list or table.

For a list, one <li> element is created with the attribute data-ng-repeat set to "<classname> in <classname>s" where <classname> corresponds to the name of the class that is set as type of the navigable end of the list's association. The <li> element defines

as content the expressions “{{<classname>.<idname>}}” where <idname> corresponds to the name of the id attribute of the class serving as type of the navigable association end.

For tables, the <thead> of the generated <table> element comprises <th> elements for each column of the table. Thereby, the contents of the generated <th> elements correspond to the labels of the columns. The <tr> element of the <tbody> defines the attribute data-ng-repeat following the pattern “<classname> in <classname>s”. Again, <classname> corresponds to the name of the class that is set as type of the navigable association end. For each column of the table, one <td> element is generated that defines an expression following the pattern “{{<classname>.<propertyname>}}” where <propertyname> corresponds to the name of the property referenced by the column.

#### Example

```
//list
<div class="form-group">
  <div /* add condition */>
    <h5>Professors</h5>
    <ul id="03">
      <li data-ng-repeat="professor in professors">
        {{ professor.email }}
        //add links here
      </li>
    </ul>
    //add "add buttons" here
  </div>
</div>

//table
<div class="form-group">
  <div /* add condition */>
    <h5>Courses</h5>
    <table class="table table-striped" id="04">
      <thead>
        <tr>
          <th>Type</th>
          <th>Title</th>
          <th>Credits</th>
          <th></th>
        </tr>
      </thead>
      <tbody>
        <tr data-ng-repeat="course in courses">
          <td> {{ course.type }} </td>
          <td> {{ course.title }} </td>
          <td> {{ course.credits }} </td>
          <td>
            //add links here
          </td>
        </tr>
      </tbody>
    </table>
    //add "add buttons" here
  </div>
</div>
```

## Error Span Tags

For checking whether user input has been provided for an `<input>`, `<textarea>`, or `<selection>` element and whether the user input is valid, `<span>` elements have to be generated for property elements. The `<span>` elements have as class set "`<viewname>Span`" where `<viewname>` corresponds to the name of the view containing the property element (whitespaces are removed). Furthermore, they define the attribute `data-ng-show` with a value corresponding to the pattern "`<viewname>Form.<label>.$dirty && <viewname>Form.<label>.$invalid`" where `<label>` corresponds to the label of the property element.

For mandatory properties, an additional `<span>` element is generated that defines for the attribute `data-ng-show` the value "`<viewname>Form.<label>.$error.required`" and the content "Input is mandatory."

For text view elements and data time pickers defining a format string, an additional `<span>` element is generated with the `data-ng-show` attribute "`<viewname>Form.<label>.$error.pattern`" and the content "Input doesn't match expected pattern."

Example
<pre>&lt;span class="CreateInstituteSpan" style="color:red"   data-ng-show="CreateInstituteForm.number.\$dirty &amp;&amp;               CreateInstituteForm.number.\$invalid"&gt;  //only for mandatory property elements: &lt;span data-ng-show="CreateInstituteForm.number.\$error.required"&gt;   Input is mandatory. &lt;/span&gt;  //only for text view elements or date time pickers defining a format strings &lt;span data-ng-show="CreateInstituteForm.number.\$error.pattern"&gt;   Input doesn't match expected pattern. &lt;/span&gt;  &lt;/span&gt;</pre>

## Buttons

### Add Buttons

For a link contained by a linkable element that points to a create view as target view, a `<button>` elements is created that has the attribute value set to the name of the target view.

```
//add button
<button value="CreateProfessor" class="btn btn-primary btn-sm">Add</button>
```

### Save Buttons

For create and update views, save buttons are generated and added to the `<form>` elements generated for the create/update views. Please note that if the create or update view is the start view of the welcome group, no save button is generated for this view. A save button is a `<button>` element with the value set to the name of the start view of the welcome group (whitespaces are removed). The attribute `data-ng-disabled` is set to "`<viewname>Form.$invalid`" where `<viewname>` corresponds to the name of the

create/update view. The attribute `data-ng-click` is set to the value `"save<classname>()"` where `<classname>` corresponds to the name of the class that is referenced by the create/update view.

```
//save button
<button value="CreateInstitute" class="btn btn-primary btn-sm"
  data-ng-disabled="CreateProfessorForm.$invalid"
  data-ng-click="saveProfessor()">
  Save
</button>
```

## Modal Buttons

For read and delete views, modal buttons are generated. For a read view, one modal button “Close” is generated. For a delete view, two modal buttons “Delete” and “Cancel” are generated. The “Delete” button defines the attribute `data-ng-click` with the value `"delete<classname>(<classname>.id)"` where `<classname>` corresponds to the name of the class that is referenced by the delete view.

### Example

```
//modal buttons for read view
<button class="btn btn-default" data-dismiss="modal">Close</button>

//modal buttons for delete view
<button class="btn btn-default" data-dismiss="modal"
  data-ng-click="deleteCourse(course.id)">Delete</button>
<button class="btn btn-default" data-dismiss="modal">Cancel</button>
```

## Links

For links of linkable elements pointing to read, delete, or update views as target views, `<a>` element are generated.

For links to read and delete views, the generated `<a>` element defines an empty `href` attribute, a `data-toggle` attribute with the value `"modal"`, a `data-target` attribute `"#modal<viewname>"` where `<viewname>` corresponds to the name of the target view (whitespaces are removed), and the `data-ng-click` attribute `"get<classname>(<classname>.id)"` where `<classname>` corresponds to the name of the class that is referenced by the target view. `<a>` elements generated for links to read views contain the text “show” and `<a>` elements generated for links to delete views contain the text “delete”.

For links to update views, the generated `<a>` element defines the `data-ng-click` with the value `"navigation<viewheader>('<viewname>'); update<classname>(<classname>.id)"` where `<viewheader>` corresponds to the header of the target view. Furthermore, the `<a>` element contains the value “update”.

### Example

```
//link to read view
<a href="" data-toggle="modal" data-target="#modalShowCourse"
  data-ng-click="getCourse(course.id)">show</a>
```

```
//link to delete view
<a href="" data-toggle="modal" data-target="#modalDeleteCourse"
  data-ng-click="getCourse(course.id)">delete</a></td>

//link to update view
<a href="" data-ng-click="navigationProfessor('UpdateProfessor');
  updateProfessor(professor.id)">update</a>
```

## Condition

For conditional elements, an additional attribute has to be generated that declares the defined visibility condition. For the visibility condition types hide, show, enable, and disable the attributes data-ng-hide, data-ng-show, data-ng-enabled, and data-ng-disabled are generated, respectively.

For comparison conditions, the value of this attribute has to be defined following the pattern “new<classname>.<propertyname> <comparison> <value>” where <classname> corresponds to the name of the class that is referenced by the view containing the condition, <propertyname> corresponds to the name of the property referenced by the comparison condition, <comparison> corresponds to the comparison condition type (“==” for “Equal”; “<” for “Less”; “>” for “Greater”), and <value> corresponds to the comparison value.

For composite conditions, the generated data-ng-\* attribute has to be defined following the pattern “<condition1> <composition> <condition2>” where <condition1> and <condition2> correspond to the composed conditions, and <composition> corresponds to the composite condition type (“&&” defined for “And”; “||” for “Or”).

Example
<pre>data-ng-hide="newprofessor.visitingprofessor == 'false'" data-ng-show="newcourse.type == 'LE'" data-ng-enabled="newcourse.credits &gt; '1.0' &amp;&amp; newcourse.credits &lt; '30.0'" data-ng-disabled="newcourse.credits &lt; '1.0'    newcourse.credits &gt; '30.0'"</pre>

## Additional Information

### 1. Setting up your Workspace

Both code generators have to be based on the DVML metamodel provided as part of the sample solution for lab 1. We provide this sample solution as well a skeleton Xtend project in the assignment resources. They have to be used for developing the code generators.

#### Import Projects

To import the provided project in Eclipse select *File* → *Import* → *General/Existing Projects into Workspace* → *Select archive file* → *Browse*. Choose the downloaded archive *ME\_WS15\_Lab4\_Resources.zip* and import the projects called *at.ac.tuwien.big.views* and *at.ac.tuwien.big.views.gen*.

After you have successfully imported the two projects, make sure that the first one has the "Modeling" nature applied or apply it if necessary (Right click on project → *Configure* → *Add Modeling Project Nature*). The other one serves as base for your code generators and should have the Xtext Nature applied.

### Register DVML Metamodel

Before you start implementing your code generators, you have to register the DVML metamodel in your Eclipse instance, so that the Xtend editor with which you develop your generators knows which metamodel you want to use. You can register the provided metamodel by clicking right on *at.ac.tuwien.big.views/model/views.ecore* → *EPackages registration* → *Register EPackages into repository*.

## 2. Developing your Code Generators

Define the AngularJS code generator in the file *Domain2AngularJSGenerator.xtend* and the HTML code generator in the file *View2HTMLGenerator.xtend*. Both files are located in the folder *src* of the provided project *at.ac.tuwien.big.views.gen*.

Note: In this assignment you have to implement your solution with Xtend. Make sure that your Eclipse instance is setup as described in the Eclipse Setup Guide provided in TUWEL. Documentation about how to use Xtend can be found in the Xtend documentation<sup>2</sup> and in the lecture slides.

## 3. Generate your Code

For running your code generators *Domain2AngularJSGenerator.xtend* and *View2HTMLGenerator.xtend* you have to run the workflow file *Dvm12WebGenerator.mwe2* located next to the *.xtend* files. The workflow will run both of your generators for the DVML models located in the folder *input* of the project *at.ac.tuwien.big.forms.gen*. To run the workflow, right click on the workflow file *Dvm12WebGenerator.mwe2* and select *Run As* → *MWE2 Workflow*. This will start the code generation. Check the output in the console to see whether the generation was successful. The code generated for the input models will be located in the folder *output* of the project *at.ac.tuwien.big.forms.gen*. Note that after each change in one of your code generators, you have to re-run the workflow to update the generated code. If you start the code generator for the first time the following message may appear:

**\*ATTENTION\***  
It is recommended to use the ANTLR 3 parser generator (BSD licence - <http://www.antlr.org/license.html>). Do you agree to download it (size 1MB) from '<http://download.itemis.com/antlr-generator-3.2.0.jar>'? (type 'y' or 'n' and hit enter)

Type y and download the jar file. It will be automatically integrated into your project.

## 4. Testing your Generated Code

**Markup Validation:** You have to check the markup validity<sup>3</sup> of your generated HTML files using the W3C Markup Validation Service. Visit the site <http://validator.w3.org/> -> *Validate by Direct Input*, copy your generated HTML code into the text area, and click *Check*. You have to make sure that the validation of your HTML code is successful. However, you can ignore displayed warnings.

<sup>2</sup> Xtend documentation: <http://www.eclipse.org/xtend/documentation.html>

<sup>3</sup> Markup validity: [http://validator.w3.org/docs/help.html#validation\\_basics](http://validator.w3.org/docs/help.html#validation_basics)

**View Testing:** Open your generated `institute.html` file by right-clicking on it and selecting *Open with -> Web Browser*. The following manual tests must work without any unexpected behavior.

#### Manual Tests:

---

**Condition Test 1** In the view 'CreateProfessor', the selection field 'Home university' is only visible if either no value or the value 'true' is selected for the selection field 'Visiting professor'. Otherwise, if the value 'false' is selected, the selection field 'Home university' becomes invisible. When you select 'true', the selection field 'Home university' becomes visible again.  
(For switching to the 'CreateProfessor' view, hit the button 'Add' located at the welcome page 'Institute' below the list 'Professors'.)

---

**Condition Test 2** In the view 'CreateCourse', the selection field 'Type' is only enabled if either no value or a double value greater than 0.0 and smaller than 30.0 is provided as input in the field 'Credits', otherwise the type field should be disabled.  
(For switching to the 'CreateCourse' view, hit the button 'Add' located at the welcome page 'Institute' below the table 'Courses'.)

---

**Condition Test 3** After selecting a 'Type' at the view 'CreateCourse' the correct elements have to be displayed. In particular, for 'Lecture' the header 'Lecture' with the text area 'Topic' should be shown. For "Thesis" the header 'Thesis' with the text field 'Keywords' should be shown.

---

**Regular Expression** Make sure that the input values of input fields generated for view elements with regular expression are accordingly checked. For instance, the input field 'Number' of the welcome page 'CreateInstitute' should show the error message 'Input doesn't match expected pattern.' if non numeric value is provided.

---

**Mandatory Element Test 1** A mandatory element has to be marked with a star (\*) after its label (e.g. 'Number\*' of the welcome page 'CreateInstitute'). As long as no inputs are made for mandatory element the button 'Save' is disabled.

---

**Mandatory Element Test 2** The input field 'Number\*' of the welcome page 'CreateInstitute' should show the error message 'Input is mandatory.' if an input was provided and deleted afterwards.

---

The AngularJS code and the HTML code that should be generated for the provided example models `instituteDomainModel.views` and `instituteViewModel.views` are provided in the assignment resources. The expected AngularJS code is provided in the AngularJS file `institute.js` and the expected HTML code is provided in the HTML file `institute.html`. Both files are located in the folder `expected_output` of the project `at.ac.tuwien.big.views.gen`. Compare the behavior of these expected output files with the behavior of the output files generated by your code generators.

**Note:** Your code generators have to be able to generate suitable AngularJS and HTML code for any valid DVML model (not only for the provided example models).

# Submission & Assignment Review

## Upload the following components in TUWEL:

You have to upload one archive file, which contains the `at.ac.tuwien.big.views.gen` project. For exporting the project, select *File* → *Export* → *General/Archive File* and select the project.

**Make sure to include all resources that are necessary for the code generation!**

If you are unsure whether you have provided all required resources, test your exported projects by importing them into a new workspace, running the code generators, and testing the generated code again.

## Assignment Review

At the assignment review you will have to present your solution, in particular, your Xtend code generators. You will also have to show that you understand the theoretical concepts underlying the assignment.

**All group members have to be present at the assignment review.** Registration for the assignment review can be done in TUWEL. The assignment review consists of two parts:

- **Submission and group evaluation:** 20 out of 25 points can be reached.
- **Individual evaluation:** Every group member is interviewed and evaluated separately. The remaining 5 points can be reached. If a group member does not succeed in the individual evaluation, the points reached in the group evaluation are also revoked for this student, which results in a negative grade for the entire course.