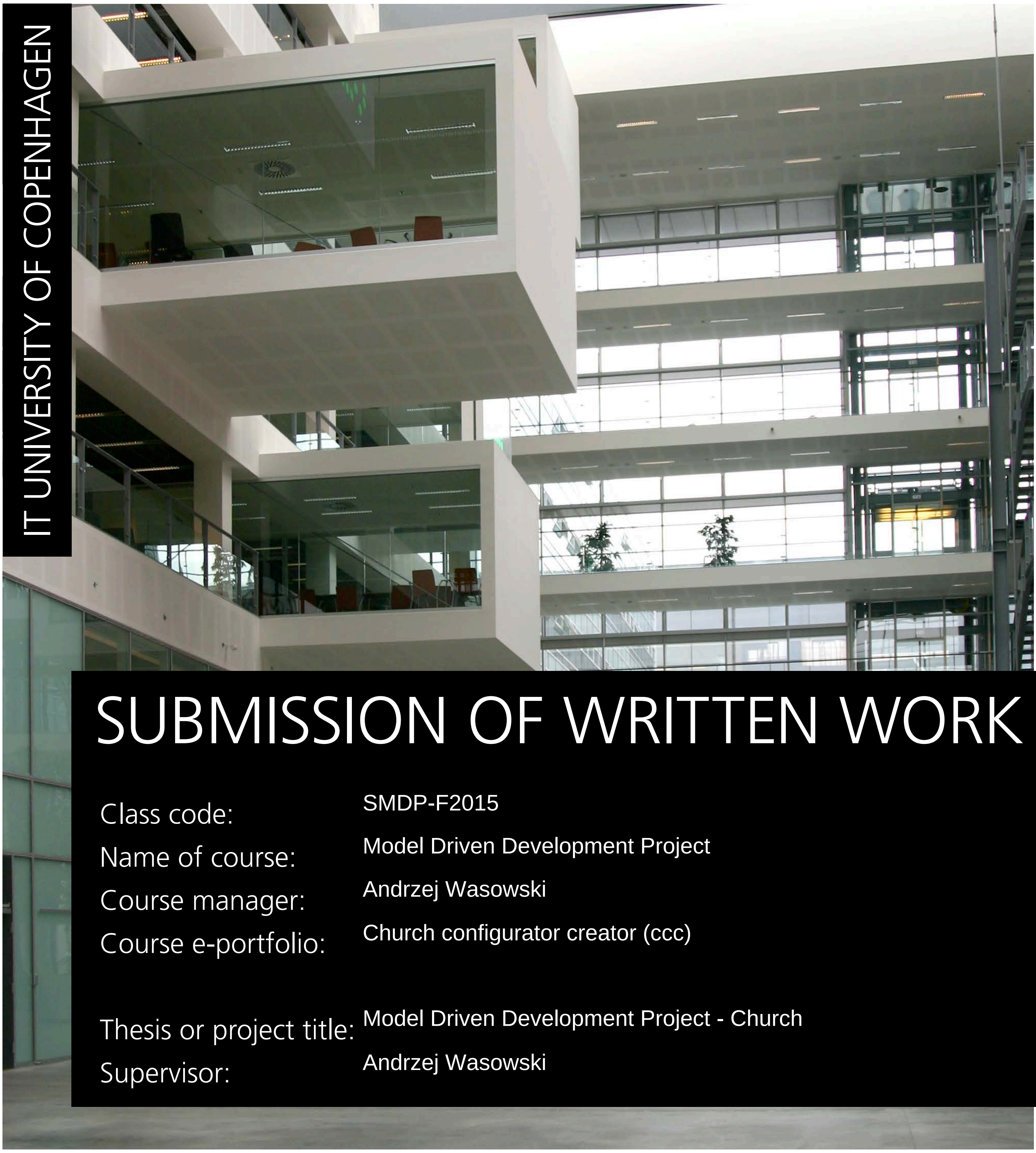


IT UNIVERSITY OF COPENHAGEN

SUBMISSION OF WRITTEN WORK

Class code:	SMDP-F2015
Name of course:	Model Driven Development Project
Course manager:	Andrzej Wasowski
Course e-portfolio:	Church configurator creator (ccc)
Thesis or project title:	Model Driven Development Project - Church
Supervisor:	Andrzej Wasowski

	Full Name:	Birthdate (dd/mm-yyyy):	E-mail:
1.	Rasmus Lohals Larsen	20/11-1977	rlla@itu.dk
2.	Sten Anderson	30/06 - 1972	stea@itu.dk
3.	Ole Ertmann Nielsen	13/09 - 1971	oen@itu.dk
4.	Morten Egelund Rasmussen	30/03 - 1972	megr@itu.dk
5.			@itu.dk
6.			@itu.dk
7.			@itu.dk

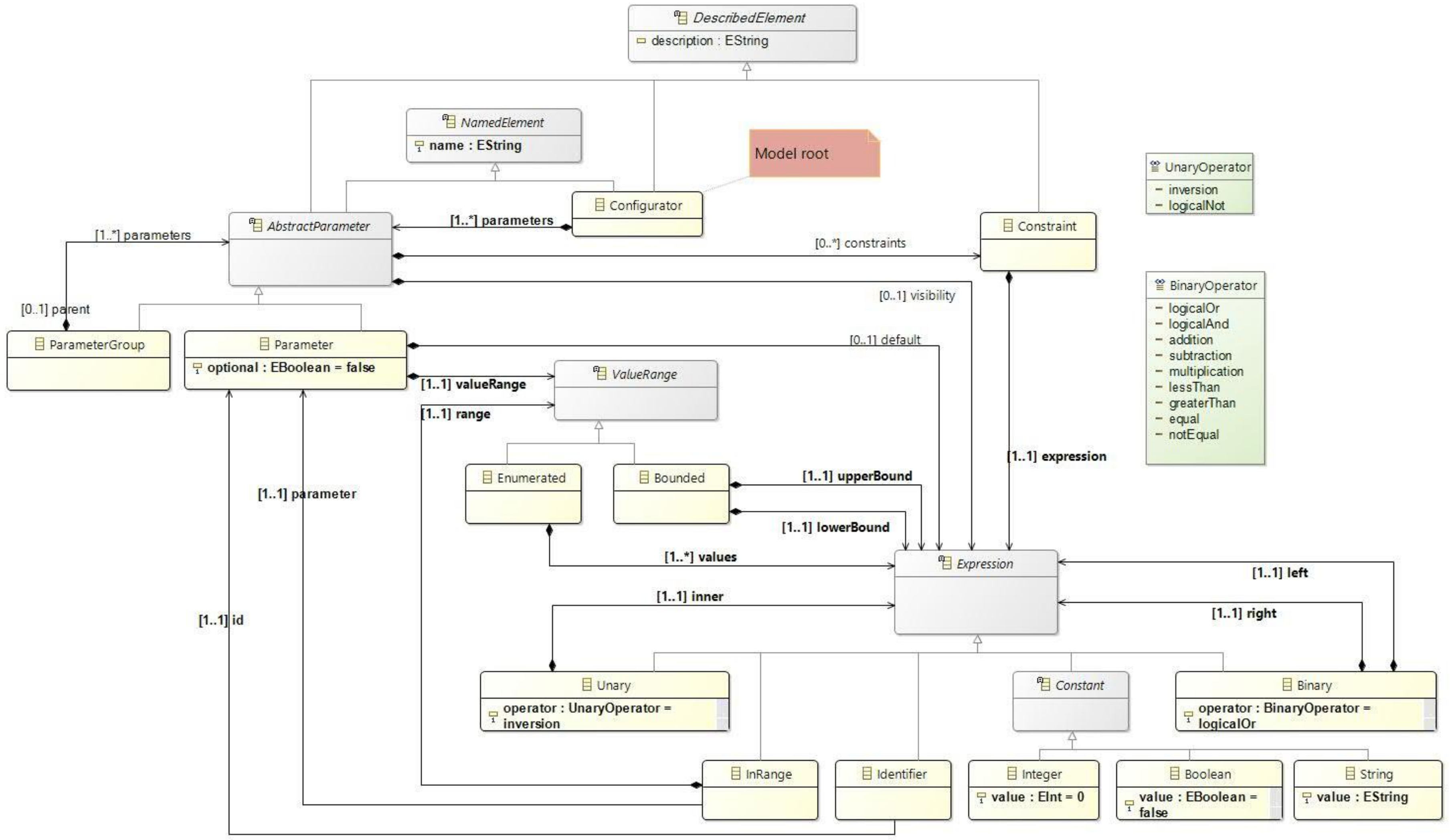
1 Table of Contents

2 Example textual model

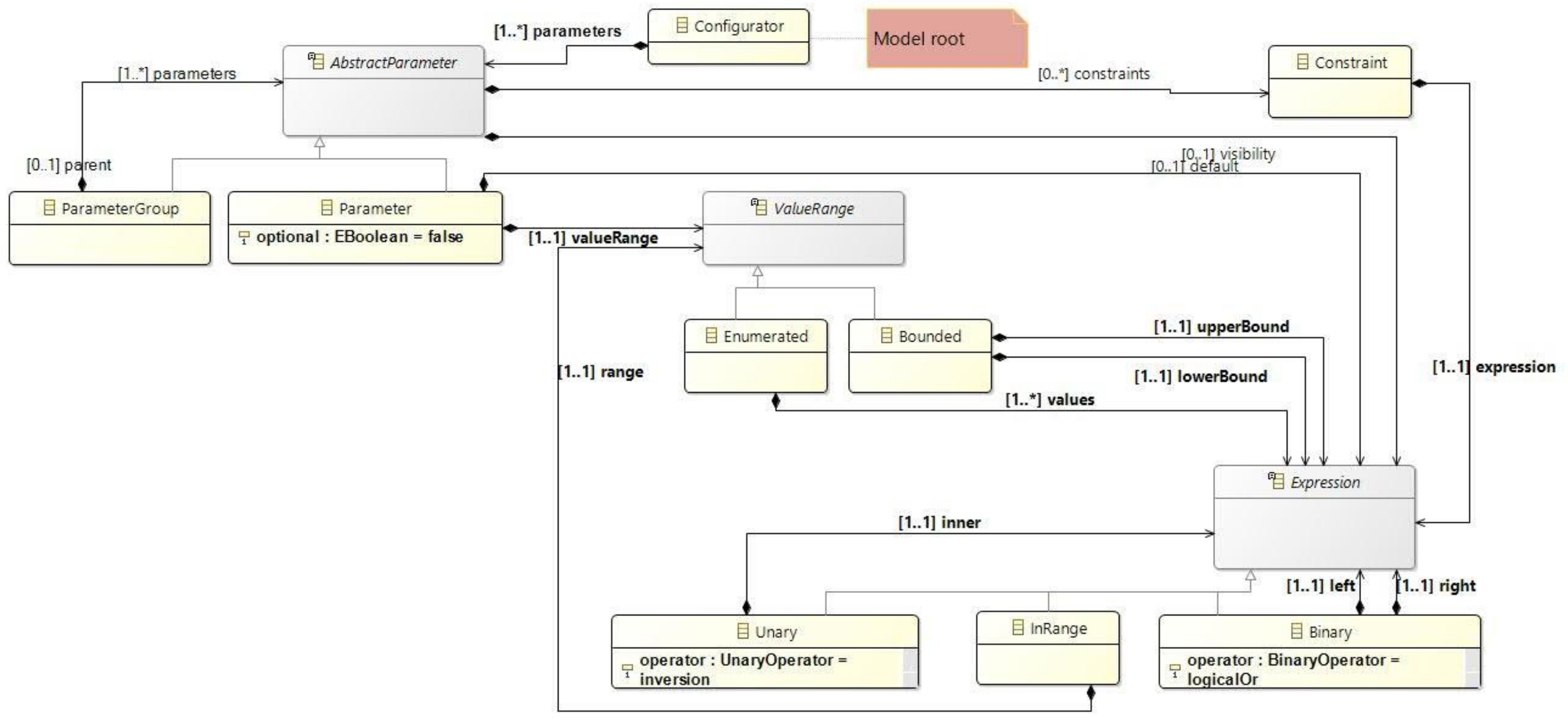
```
configurator car "A configurator for a car" {

        parameter length "length of car" mandatory values [(1+1)*2;9-1]

        parameter air_condition values (true, false) default-value true

        parameter doors optional values [3;5] visible-if
                (not (variant == "sport") and length > 5)

        parameter variant values ("standard", "sport", "luxury") default-value "standard"

        parameter engine values ("TFSI 1.2", "TFSI 1.4", "TFSI 2.0")
                constraints {
                        engine in ("TFSI 1.2", "TFSI 1.4") or variant == "sport"
                },

        parameter fog_lights optional values (true,false),

        group seats "type of seats" visible-if variant != "standard" {
                parameter material values ("leather", "cloth") mandatory
                parameter colour "the seat colour" values ("red", "blue", "black"),
        } constraints {
                description "invalid seats: leather can only have colour red, black"
                        material != "leather" or (colour in ("red", "black")),
                description "invalid seats: cloth can not be red"
                        material != "cloth" or colour in ("blue", "black")
        }
}
```

3 Meta-model

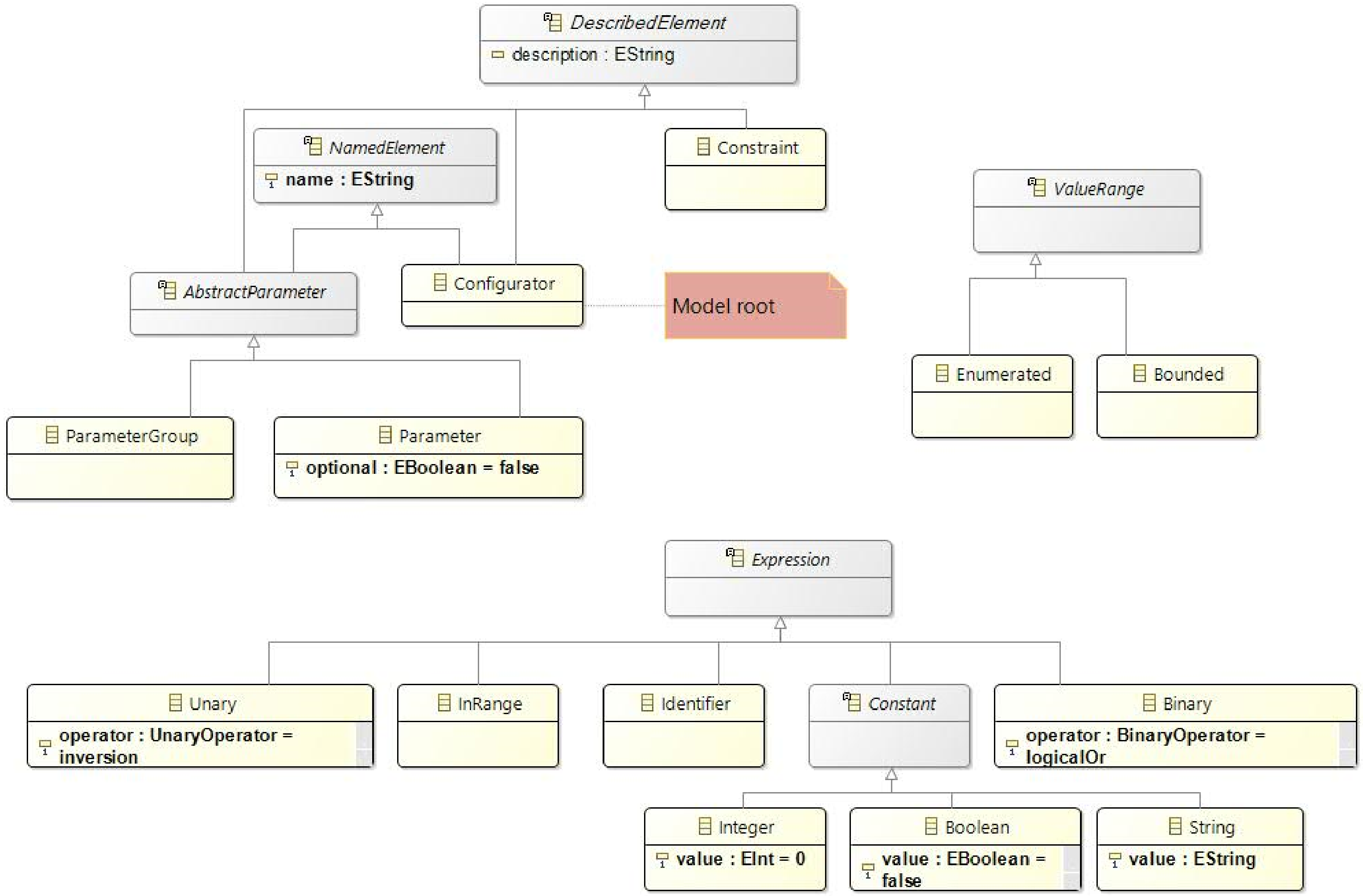
3.1 Class diagram



3.2 Partonomy (with a few type relations preserved for clarification purposes)



3.3 Taxonomy


DescribedElement
description : EString
NamedElement
name : EString
Constraint
ValueRange
AbstractParameter
Configurator
Model root
Enumerated
Bounded
ParameterGroup
Parameter
optional : EBoolean = false
Expression
Unary
operator : UnaryOperator = inversion
InRange
Identifier
Constant
Binary
operator : BinaryOperator = logicalOr
Integer
value : EInt = 0
Boolean
value : EBoolean = false
String
value : EString

4 Static semantics

4.1 Static constraints

```
@Check
def checkEnumeratedExpressionIsConstant(Enumerated it) {
	// Check that each value in the enumerated expression can be evaluated as a static value:
	values.forEach[
		if (staticValue == null) {
			error('Enumerated item should be a constant.', ConfiguratorPackage.Literals.ENUMERATED__VALUES,
				INVALID_ENUMERATION)
		}]
}

@Check
def checkBoundedExpressionUpperBoundIsConstant(Bounded bounded) {
	// Check that the upper bound in a bounded expression can be evaluated as a static value:
	if (bounded.upperBound.staticValue == null) {
		error('Upper bound should be a constant.', ConfiguratorPackage.Literals.BOUNDED__UPPER_BOUND,
			INVALID_BOUND)
	}
}

@Check
def checkBoundedExpressionLowerBoundIsConstant(Bounded bounded) {
	// Check that the lower bound in a bounded expression can be evaluated as a static value:
	if (bounded.lowerBound.staticValue == null) {
		error('Lower bound should be a constant.', ConfiguratorPackage.Literals.BOUNDED__LOWER_BOUND,
			INVALID_BOUND)
	}
}

@Check
def checkBoundedExpressionLowerIsBelowUpper(Bounded bounded) {
	// Check that the lower bound in a bound expression is less than the upper bound
	val lowerVal = bounded.lowerBound?.staticValue
	val upperVal = bounded.upperBound?.staticValue
	var c = -1;
	if (lowerVal instanceof Integer && upperVal instanceof Integer) {
		// If values are of type Integer:
		c = (lowerVal as Integer).compareTo(upperVal as Integer)
	}
	if (lowerVal instanceof String && upperVal instanceof String) {
		// If values are of type String:
		c = (lowerVal as String).compareTo(upperVal as String)
	}
	if (lowerVal instanceof Boolean && upperVal instanceof Boolean) {
				// If values are of type Boolean:
				c = (lowerVal as Boolean).compareTo(upperVal as Boolean)
		}
		if (c >= 0) {
				error('Lower bound should be less than upper bound', ConfiguratorPackage.Literals.BOUNDED__LOWER_BOUND,
					INVALID_BOUND)
		}
}

@Check
def checkEnumeratedSequence(Enumerated enumerated) {
		// Check that values contained in Enumerated all have unique static values
		// (e.g. 2+2 must not be in the same sequence as 4)
		enumerated.values.forEach [ v |
				if (enumerated.values.filter[staticValue == v.staticValue].size != 1)
					error('Enumerated values should be unique', ConfiguratorPackage.Literals.ENUMERATED__VALUES,
						INVALID_ENUMERATION)
		]
}

@Check
def checkDefaultValue(Parameter parameter) {
		// Constraint check on parameter default value.
		// Does this parameter have a default value at all?:
		if (parameter.^default != null) {
				val defVal = parameter.^default.staticValue
				val range = parameter.valueRange
				switch (range) {
					Enumerated:
						// Constraint check on Enumerated ValueRange type:
						if (!range.values.exists[staticValue == defVal])
							// Error if default value is not among the listed elements in the Enumerated collection:
							error('Default value should be among the listed values',
								ConfiguratorPackage.Literals.PARAMETER__DEFAULT, INVALID_BOUND)
					Bounded: {
						// Constraint check on Bounded ValueRange type:
						var defaultValueIsValid = true;
						if (range.lowerBound.staticValue instanceof Integer) // Bounded ValueRange elements are of type Integer
							// Check that default value lies between lower and upper bound:
							defaultValueIsValid = (range.lowerBound.staticValue as Integer) <= (defVal as Integer)
								&& (range.upperBound.staticValue as Integer) >= (defVal as Integer)
						else if (range.lowerBound.staticValue instanceof String) // Bounded ValueRange elements are of type String
							// Check that default value lies between lower and upper bound (based on string values):
							defaultValueIsValid = (range.lowerBound.staticValue as String) <= (defVal as String)
								&& (range.upperBound.staticValue as String) >= (defVal as String)
						if (!defaultValueIsValid)
							// Throw an error:
							error('Default value should be within the specified value range',
								ConfiguratorPackage.Literals.PARAMETER__DEFAULT, INVALID_BOUND)
```

```
                        }
                }
        }
}

@Check
def checkIdentifierOptional(Identifier identifier) {
        // Check if any Identifier refers to an optional parameter:
        if (identifier.id.optional) {
                error('Identifier cannot refer to an optional parameter', ConfiguratorPackage.Literals.IDENTIFIER__ID, OPTIONAL_PARAMETER_INVALID)
        }
}

@Check
def checkInRangeOptional(InRange inRange) {
        // Check if any InRange refers to an optional parameter:
        if (inRange.parameter.optional) {
                error('Identifier cannot refer to an optional parameter', ConfiguratorPackage.Literals.IN_RANGE__PARAMETER,
                        OPTIONAL_PARAMETER_INVALID)
        }
}

@Check
def checkUniqueParameterNames(Configurator configurator) {
        // Check that all parameters and parameter groups have globally unique names:
        var params = configurator.parameters.names
        if (params.length != params.toSet.length) {
                error('All parameters and parameter groups must have globally unique names', ConfiguratorPackage.Literals.NAMED_ELEMENT__NAME,
                        PARAMETER_NAME_NOT_UNIQUE)
        }
}

def private List<String> names(EList<AbstractParameter> it) {
         // Get all abstract parameter names:
        var paramNames =
                fold(new ArrayList<String>) [ parameterNames, abstractParameter | parameterNames.add(abstractParameter.name); parameterNames]
        // Add names of all parameters in any underlying parameter groups (notice the recursion):
        paramNames.addAll(
                it.filter(ParameterGroup).fold(new ArrayList<String>)
                [parameterNames, parameterGroup | parameterNames.addAll(parameterGroup.parameters.names); parameterNames]
        )
        // Just return parameter names
        paramNames
}

def private checkExpectedType(ExpressionType actualType, ExpressionType expectedType, EReference reference) {
        if (actualType != expectedType) {
                error("expected type " + expectedType + ", actual type is " + actualType, reference, WRONG_TYPE)
        }
```

```
}

def private ExpressionType getTypeAndCheckNotNull(Expression expression, EReference reference) {
        var type = expression?.typeFor
        if (type == null)
                error("unknown type", reference, WRONG_TYPE)
        type
}

def private ExpressionType getTypeAndCheckNotNull(ValueRange range, EReference reference) {
        var type = range?.rangeType
        if (type == null)
                error("unknown type", reference, WRONG_TYPE)
        type
}
```

4.2 Type Checking

The meta model allows three distinct types of expressions and values.
For simplicity, there are only one kind of numbers. Strings do not have an ordering nor can they be added (concatenated).
The class ExpressionTypeProvider provides the overloaded typeFor method that gives the expected type of an expression without looking at sub expressions. This is used in the overloaded checkType method of the ConfiguratorValidator class.

```
enum ExpressionType {
        String, Integer, Boolean
}

class ExpressionTypeProvider {

        def dispatch ExpressionType typeFor(Constant constant) {
                switch (constant) {
                        String: ExpressionType.String
                        Boolean: ExpressionType.Boolean
                        Integer: ExpressionType.Integer
                }
        }

        def dispatch ExpressionType typeFor(Binary binary) {
                switch (binary.operator) {
                        case ADDITION:
                                ExpressionType.Integer
                        case LOGICAL_AND:
                                ExpressionType.Boolean
                // …

                }
        }



        def private checkExpectedType(ExpressionType actualType, ExpressionType expectedType,
EReference reference) {
                if (actualType != expectedType) {
                        error("expected type " + expectedType + ", actual type is " + actualType,
reference, WRONG_TYPE)
                }
        }

        def private ExpressionType getTypeAndCheckNotNull(Expression expression, EReference reference)
{
                var type = expression?.typeFor
                if (type == null)
                        error("unknown type", reference, WRONG_TYPE)
                type
        }

        @Check
        def checkType(Unary unary) {
                val innerLiteral = ConfiguratorPackage.Literals.UNARY__INNER
                val innerType = getTypeAndCheckNotNull(unary.inner, innerLiteral)
                switch (unary.operator) {
                        case INVERSION: {
                                checkExpectedType(innerType, ExpressionType.Integer, innerLiteral)
                        }
                        case LOGICAL_NOT: {
                                checkExpectedType(innerType, ExpressionType.Boolean, innerLiteral)
                        }
                }
        }
```

4.3 Static value checking

The meta model allows for arbitrary expressions in value ranges where constant expressions are expected. This is due to the fact that constant numbers in the grammar are unsigned. A signed number therefore requires a unary expression (unary minus). In order to validate that expressions in value ranges indeed are constant and relate properly (e.g. increasing size) all expressions are evaluated to a constant value if possible.

A static value is simply a java.lang.Integer/Boolean/String object, or null if no static value exists. No static value exists if the expression contains reference to a Parameter value (by an Identifier or InRange expression).

The extension method ExpressionValueProvider.staticValue calculates the value of an expression based on the value of any sub expressions:

```
        def dispatch Object staticValue(Constant constant) {
                switch (constant) {
                        String: constant.value
                        dk.itu.smdp2015.church.model.configurator.Boolean: constant.value
                        dk.itu.smdp2015.church.model.configurator.Integer: constant.value
                }
        }

        def dispatch Object staticValue(Binary binary) {
                val vleft = binary.left.staticValue
                val vright = binary.right.staticValue
                switch (binary.operator) {
                        case ADDITION:
                                if (vleft instanceof Integer && vright instanceof Integer) {
                                        val ileft = (vleft as Integer).intValue
                                        val iright = (vright as Integer).intValue
                                        new Integer(ileft + iright)
                                }
                        case LOGICAL_AND:
                                if (vleft instanceof Boolean && vright instanceof Boolean) {
                                        val bleft = (vleft as Boolean).booleanValue
                                        val bright = (vright as Boolean).booleanValue
                                        new Boolean(bleft && bright)
                                }

                        /// ... (more cases here -- see source code) ...
                }
        }

        def dispatch ExpressionType staticValue(Identifier identifier) {
                null
        }
}
```

5 Xtext grammar

```
Configurator:
        'configurator' name=ID
        (description=STRING)?
        '{' parameters+=AbstractParameter ( ','? parameters+=AbstractParameter)* '}';

AbstractParameter:
        ParameterGroup | Parameter;

ParameterGroup:
        'group' name=ID
        (description=STRING)?
        ( ('visible-if' visibility=Expression)?
        & ('constraints' '{' constraints+=Constraint ( ',' constraints+=Constraint)* ','? '}' )?
        & '{' parameters+=AbstractParameter ( ','? parameters+=AbstractParameter)* ','? '}' );

Parameter:
        'parameter'     name=ID
        (description=STRING)?
        ( ((optional?='optional')|'mandatory')?
        & ('visible-if' visibility=Expression)?
        & ('default-value' default=Expression)?
        & ('constraints' '{' constraints+=Constraint ( ',' constraints+=Constraint)* '}' )?
        & 'values' valueRange=ValueRange );

ValueRange:
        Enumerated | Bounded;

Enumerated returns Enumerated:
        '(' values+=Expression ( ',' values+=Expression)* ')';

Bounded returns Bounded:
        '[' lowerBound=Expression ';' upperBound=Expression ']';

Constraint:
        ('description' description=STRING)?
         expression=Expression;

Expression:
        LogicalOr;

enum LogicalOrOperator returns BinaryOperator:
        logicalOr = 'or' ;

LogicalOr returns Expression:
        LogicalAnd ( {Binary.left=current} operator=LogicalOrOperator right=LogicalAnd )*;

enum LogicalAndOperator returns BinaryOperator:
        logicalAnd = 'and';

LogicalAnd returns Expression:
        Equality ( {Binary.left=current} operator=LogicalAndOperator right=Equality )*;

enum EqualityOperator returns BinaryOperator:
        equal = '==' | notEqual = '!=';

Equality returns Expression:
        Comparative ( {Binary.left=current} operator=EqualityOperator right=Comparative )*;

enum ComparativeOperator returns BinaryOperator:
        lessThan = '<' | greaterThan = '>' ;

Comparative returns Expression:
        Additive ( {Binary.left=current} operator=ComparativeOperator right=Additive )*;

enum AdditiveOperator returns BinaryOperator:
        addition = '+' | subtraction = '-';

Additive returns Expression:
        Multiplicative ( {Binary.left=current} operator=AdditiveOperator right=Multiplicative )*;
```

```
enum MultiplicativeOperator returns BinaryOperator:
        multiplication = '*';

Multiplicative returns Expression:
        Primitive ( {Binary.left=current} operator=MultiplicativeOperator right=Primitive )*;

Primitive returns Expression:
        Unary | InRange | Integer | Boolean | String0 | Identifier | '(' Expression ')';

enum UnaryOperator:
        inversion = '-' | logicalNot = 'not';

Unary:
        operator=UnaryOperator inner=Primitive;

Constant:
        Integer | Boolean | String0;

InRange:
        parameter=[Parameter] 'in' range=ValueRange;

Integer:
        value=EInt;

Boolean:
        value=EBoolean;

String0 returns String:
        value=STRING;

Identifier:
        id=[Parameter];

EInt returns ecore::EInt:
        /* '-'? */ INT;

EDouble returns ecore::EDouble:
        /* '-'? */ INT? '.' INT (('E'|'e') '-'? INT)?;

EBoolean returns ecore::EBoolean:
        'true' | 'false';
```

6 Backends

6.1 HTML 5 mobile web client

The HTML client is build using HTML5, javascript and CSS. The code generated is purely html and javascript, so no compilation is taking place as these scripts are interpreted by a browser. We have used two popular javacript frameworks Jquery Mobile (JQM) and Knockout to build a single page web application (SPA), with a clearly defined user interface architecture. Jquery mobile enables mobile oriented user experiences using a simple declarative markup. Depending on the markup the framework applies javascript and CSS to give the application a native mobile look and feel. Knockout is a two data binding javascript framework that uses the Model-View-ViewModel (MVVM) user interface architectual pattern to facilitate a clear separation of concerns between user interface logic and data model manipulation. This separation made it fairly straight forward to generate code from an instance of our Meta Model. Using the Knockout validation plugin, converting our validation expressions into javascript code was also straight forward, as this plugin enable custom validation rules, which is automatically applied by the framework.

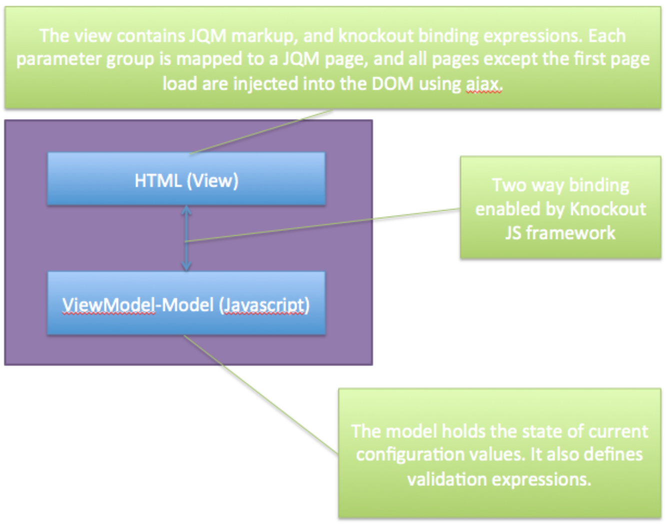

The view contains JQM markup, and knockout binding expressions. Each parameter group is mapped to a JQM page, and all pages except the first page load are injected into the DOM using ajax.
HTML (View)
Two way binding enabled by Knockout JS framework
ViewModel-Model (Javascript)
The model holds the state of current configuration values. It also defines validation expressions.

Figure 6.1.1: Overview of HTML client architecture

Jquery mobile page sample

```
<div id="main" data-role="page" data-add-back-btn="true">
   <div data-role="header">
      <h1>
         car
      </h1>
      <button class="ui-btn-right ui-icon-check ui-btn-icon-right ui-btn"
onclick="submitconfiguration();">submit</button>
   </div>

   <div role="main">
      <section class="description">
         A configurator for a car
      </section>
      <section class="validationSection" data-bind="css:{showValidationSummary: !isModelValid()}">
         <div class="validationSummary">
            <h4>Validation summary</h4>
            <ul data-bind="foreach: currentErrors">
               <li><span data-bind="text: $data"> </span> </li>
            </ul>
         </div>
      </section>
```

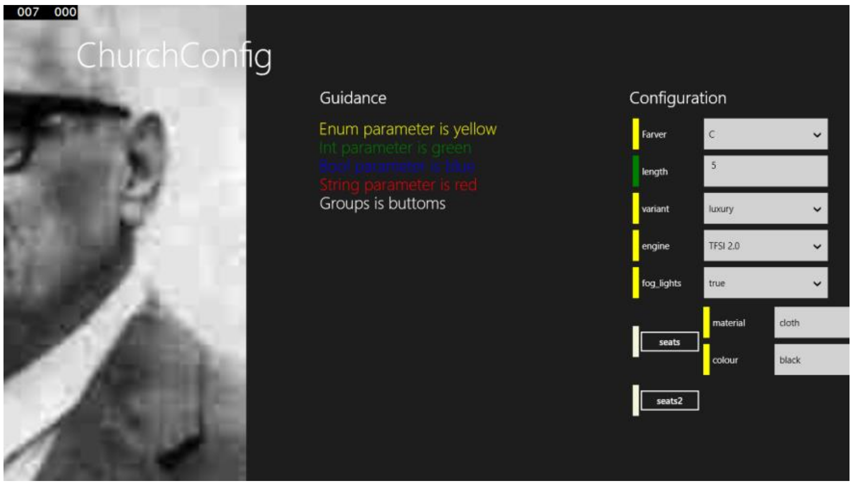
```
    <ul data-role="listview">
      <li>
        <label for="engine-param">engine:</label>
         <select  id="engine-param" data-bind="options: engine.choices, selectedOptions:
engine.value,optionsCaption:'Choose...'">
          </select>
        <p class="validationMessage" data-bind="validationMessage: engine.value"></p>
      <li data-bind="visible: group_seats().isVisible">
        <a href="#seats">seats
          <p class="validationMessage" data-bind="validationMessage: group_seats"></p></a>
          </li>
    </ul>
  </div>
</div>
```

Knockout ViewModel object sample

```
engine:
{
  choices: ['TFSI 1.2', 'TFSI 1.4', 'TFSI 2.02'],
  value: ko.observable()
    .extend({
      validation: {
        validator: function (val, param) {
          if(App.ViewModel==null)//not initialized
            return true;
          //Expression here:
          var result =
            (
              $.inArray("TFSI 1.2", App.ViewModel().engine.value()) > -1 ||
              $.inArray("TFSI 1.4", App.ViewModel().engine.value()) > -1
            ) ||
            $.inArray("sport", App.ViewModel().variant.value()) > -1
          return result;

        },
        message: "Big engines only available for sports model"
      }})
}
```

6.2 Windows client(Windows store app)

Image of the CS configurator application with the 'VW' test configuration



Guidance

Enum parameter is yellow
Int parameter is green
Bool parameter is blue
String parameter is red
Groups is buttoms

Configuration

Farver	C
length	5
variant	luxury
engine	TFSI 2.0
fog_lights	true
seats: material	cloth
seats: colour	black
seats2	

https://github.com/smdp2015/project/tree/master/Smdp2015DotNetClient

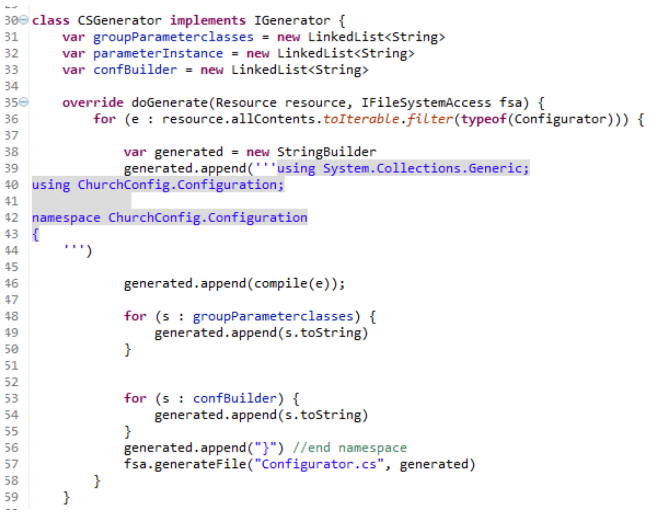
FileDescription

ChurchConfig / Configuration / Configurator.cs – generated code
ChurchConfig / Configuration / CommonConfig.cs – BaseClasses to the generated code
ChurchConfig / ConfigControl.xaml – configurator usercontrol
ChurchConfig / HubPage.xaml – Application mainpage

6.2.1 Code generator

https://github.com/smdp2015/project/tree/master/dk.itu.smdp2015.church.configurator.syntax/src/dk/itu/smdp2015/church/generator/CSGenerator.xtend

All in one file.



```
class CSGenerator implements IGenerator {
    var groupParameterclasses = new LinkedList<String>
    var parameterInstance = new LinkedList<String>
    var confBuilder = new LinkedList<String>

    override doGenerate(Resource resource, IFileSystemAccess fsa) {
        for (e : resource.allContents.toIterable.filter(typeof(Configurator))) {

            var generated = new StringBuilder
            generated.append('''using System.Collections.Generic;
using ChurchConfig.Configuration;

namespace ChurchConfig.Configuration
{
    ''')

            generated.append(compile(e));

            for (s : groupParameterclasses) {
                generated.append(s.toString)
            }


            for (s : confBuilder) {
                generated.append(s.toString)
            }
            generated.append("}") //end namespace
            fsa.generateFile("Configurator.cs", generated)
        }
    }
```

The generator collects codes in three linkedLists.

- GroupParameterClasses contains generated Groupparameter classes.
- parameterInstance contains parameter instanciation.
- confBuilder contains code to create the configuration instance.

6.2.2 Code generation output

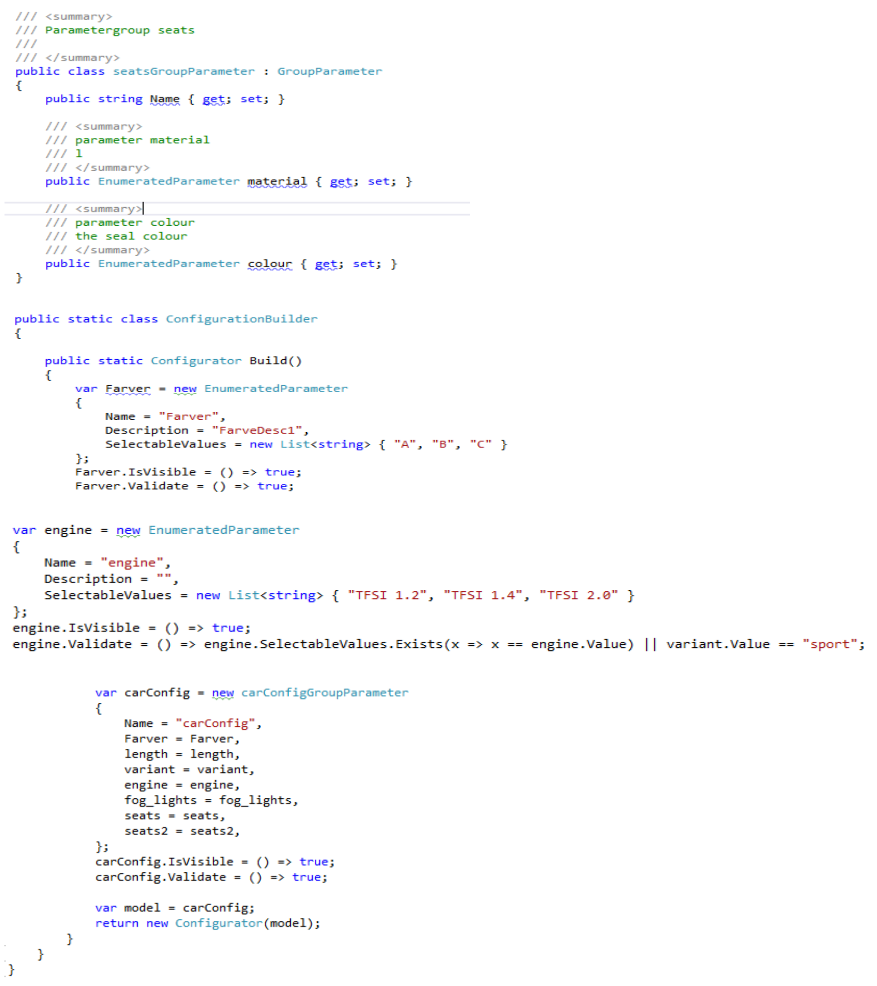
The code generator generate a static typed C# file called Configurator.cs.

Parameters and groupParameters is handled differently. Parameters have a predefined class for each instantiated type. Ex. EnumeratedParameter and IntParameter. Parametergroup's is defined as individual classes. The reason for differensation is there is only a few parameter types and almost all parameterGroups are unique.

The configurator has global scope and it is possible to reference all defined named elements from all Validate and IsVisible functions. That's the reason all parameters and parametergroups is instantiated with a named reference(ex. var name = new…).

There is no UI related code in the generated code.

Some small snippet from code generated files



```
/// <summary>
/// Parametergroup seats
///
/// </summary>
public class seatsGroupParameter : GroupParameter
{
    public string Name { get; set; }

    /// <summary>
    /// parameter material
    /// 1
    /// </summary>
    public EnumeratedParameter material { get; set; }

    /// <summary>
    /// parameter colour
    /// the seal colour
    /// </summary>
    public EnumeratedParameter colour { get; set; }
}


public static class ConfigurationBuilder
{

    public static Configurator Build()
    {
        var Farver = new EnumeratedParameter
        {
            Name = "Farver",
            Description = "FarveDesc1",
            SelectableValues = new List<string> { "A", "B", "C" }
        };
        Farver.IsVisible = () => true;
        Farver.Validate = () => true;


var engine = new EnumeratedParameter
{
    Name = "engine",
    Description = "",
    SelectableValues = new List<string> { "TFSI 1.2", "TFSI 1.4", "TFSI 2.0" }
};
engine.IsVisible = () => true;
engine.Validate = () => engine.SelectableValues.Exists(x => x == engine.Value) || variant.Value == "sport";


          var carConfig = new carConfigGroupParameter
          {
              Name = "carConfig",
              Farver = Farver,
              length = length,
              variant = variant,
              engine = engine,
              fog_lights = fog_lights,
              seats = seats,
              seats2 = seats2,
          };
          carConfig.IsVisible = () => true;
          carConfig.Validate = () => true;

          var model = carConfig;
          return new Configurator(model);
      }
   }
}
```

All parameters and parametergroups are instantiated so validation and Isvisible methods can reference other parameters. They are all in global scope.

All IsVisible and Validate properties is defined as Func<bool> delegates, because they are defined in the configuration and not in the static parameter class.

The static method ConfigurationBuilder.Build() creates an instance of the configuration model.

6.2.3 Client UI

The UI is build in xaml(which is a domain specific language ☺)

The configuration UI is created from a ListView Control.

```
<ScrollViewer >
    <ListView x:Name="Lv1"/>
</ScrollViewer>
```

In the view constructor the configurator model is instantiated.
Next the BuildListView method recursively traverse the model so every parameter and groupparameter is binded to a ListViewItems datacontext and added to the listView.

```
public ConfigControl()
{
    this.InitializeComponent();
    var model = ConfigurationBuilder.Build();
    BuildListView(Lv1, model.Configuration);
}
```

A DataTemplateSelector decides how each parameter is rendered

```
<configuration:ConfDataTemplateSelector x:Key="ConfTemplateSelector"
                                IntParameterTemplate="{StaticResource IntDataTemplate}"
                                BoolParameterTemplate="{StaticResource BoolDataTemplate}"
                                StringParameterTemplate="{StaticResource StringDataTemplate}"/>
```

DataTemplate for a string parameter

```
<DataTemplate x:Name="StringDataTemplate">
    <StackPanel Orientation="Horizontal">
        <Rectangle Width="10" Height="50"  Fill="Red"/>
        <TextBlock Text="{Binding Name}" Width="100" VerticalAlignment="Center" Margin="5,0,0,0"/>
        <TextBox  Width="200"/>
    </StackPanel>
</DataTemplate>
```

7 Test methods and artefacts

7.1 Test strategy

We have written tests covering the following parts of our project:

- Meta model: Tested through dynamic model instances.
- Parser: Testing grammar syntax.
- Constraints: Testing syntax that satisfies/violates the constraint in question.
- Code generators: Testing that different elements returns expected generated code.

We have written unit-tests for each part, which are all based on a known initial state / input (i.e. a test bench with a fixed input), and a confirmation that the tested element returns the expected output.

We have written unit tests which validates valid input, or (correctly) invalidates invalid input. Thus, we have both positive and negative test cases.

We have focused on making each unit test as small as possible, in order to give a detailed overview of the test results. This gives a clear indication for any possible test errors.

We are aware that unit tests cannot stand alone as a full test of the developed feature. An easy way extend the system tests would be to perform a compilation of the generated code (if the generated code needs to be compiled), subsequently performing an exploratory test of the final application which the user sees.

7.2 Metamodel test case examples

```
def static dispatch constraint(Configurator it) {
        !parameters.empty && !name.empty
}

def static dispatch constraint(Parameter it) {
        !name.empty
}

def static dispatch constraint(ParameterGroup it) {
        !parameters.empty && !name.empty
}

def static dispatch constraint(Bounded it) {
        var lBound = (lowerBound as dk.itu.smdp2015.church.model.configurator.Integer)
        var uBound = (upperBound as dk.itu.smdp2015.church.model.configurator.Integer)
        lBound.value < uBound.value
}

....

// Fallback
def static dispatch constraint(EObject it) {
        true
}
```

7.3 Grammar test case examples

```
@Test
def void testInvalidModelNoParameters() {
        var model = '''configurator Empty'''.parse
        model.assertError(ConfiguratorPackage.Literals.CONFIGURATOR, Diagnostic.SYNTAX_DIAGNOSTIC,
mismatched input")
}

@Test
def void testValidBoundedRange() {
        var model = '''configurator Bicycle "Bicycle configuration" { parameter wheel_size values
[16;24] }'''.parse
        assertEquals("Bicycle configuration", model.description)
        var param = model.parameters.get(0) as Parameter
        assertEquals("wheel_size", param.name)
        var valueRange = param.valueRange as Bounded
        assertEquals(16, (valueRange.lowerBound as IntegerImpl).value)
        assertEquals(24, (valueRange.upperBound as IntegerImpl).value)
        model.assertNoErrors
}
```

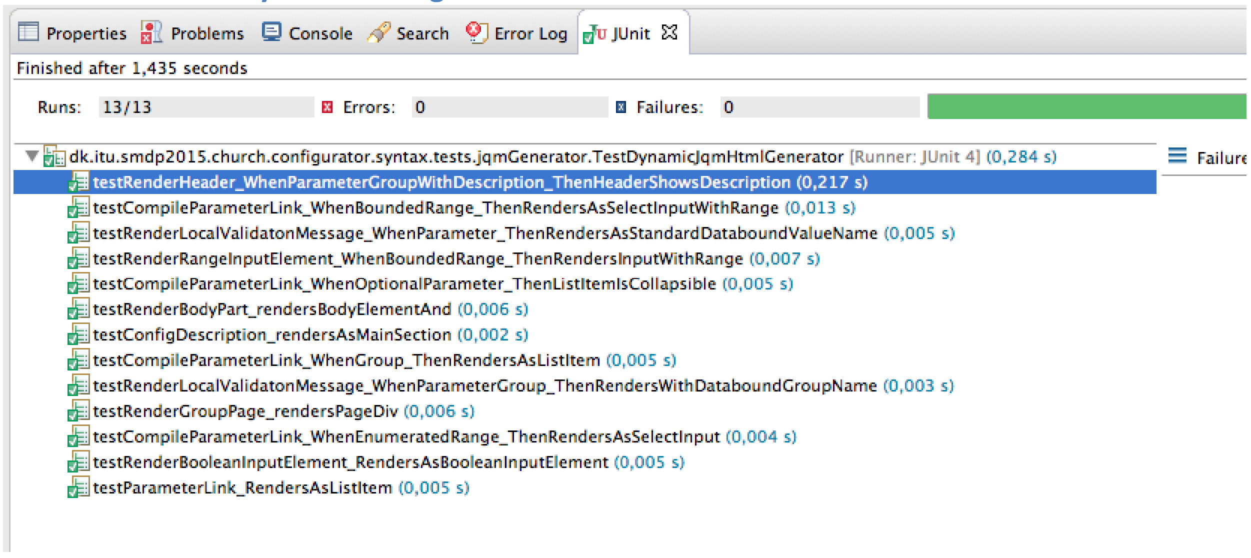
7.4 Code generators test case examples

7.4.1 1. Overview of HTML5 mobile web client

Here we only show the jUnit tests for the html generator part. The test case are divided into a dymanic html generator and a static html generator.

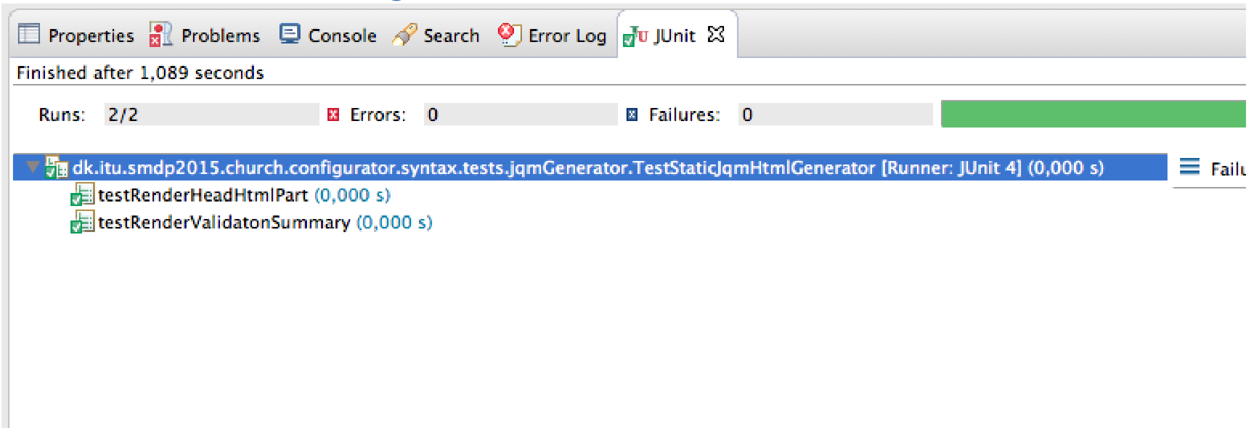
First we show an overview of the passing tests as the present themselves in the Eclipse IDE.

7.4.2 Test of dynamic html generation



```
Properties  Problems  Console  Search  Error Log  JUnit
Finished after 1,435 seconds

Runs: 13/13    Errors: 0    Failures: 0

dk.itu.smdp2015.church.configurator.syntax.tests.jqmGenerator.TestDynamicJqmHtmlGenerator [Runner: JUnit 4] (0,284 s)
    testRenderHeader_WhenParameterGroupWithDescription_ThenHeaderShowsDescription (0,217 s)
    testCompileParameterLink_WhenBoundedRange_ThenRendersAsSelectInputWithRange (0,013 s)
    testRenderLocalValidatonMessage_WhenParameter_ThenRendersAsStandardDataboundValueName (0,005 s)
    testRenderRangeInputElement_WhenBoundedRange_ThenRendersInputWithRange (0,007 s)
    testCompileParameterLink_WhenOptionalParameter_ThenListItemIsCollapsible (0,005 s)
    testRenderBodyPart_rendersBodyElementAnd (0,006 s)
    testConfigDescription_rendersAsMainSection (0,002 s)
    testCompileParameterLink_WhenGroup_ThenRendersAsListItem (0,005 s)
    testRenderLocalValidatonMessage_WhenParameterGroup_ThenRendersWithDataboundGroupName (0,003 s)
    testRenderGroupPage_rendersPageDiv (0,006 s)
    testCompileParameterLink_WhenEnumeratedRange_ThenRendersAsSelectInput (0,004 s)
    testRenderBooleanInputElement_RendersAsBooleanInputElement (0,005 s)
    testParameterLink_RendersAsListItem (0,005 s)
```

7.4.3 Test of static html generation



```
Properties  Problems  Console  Search  Error Log  JUnit
Finished after 1,089 seconds

Runs: 2/2    Errors: 0    Failures: 0

dk.itu.smdp2015.church.configurator.syntax.tests.jqmGenerator.TestStaticJqmHtmlGenerator [Runner: JUnit 4] (0,000 s)
    testRenderHeadHtmlPart (0,000 s)
    testRenderValidatonSummary (0,000 s)
```

Then an explanation of how the testcode is built up.

Sample code showing some sample dynamic html test case. We make heavily use of Xtends ability to do chained method calls, increasing readability of the code.

All tests are build using the same pattern:

1. Arrange part
An input DSL string, just containing the essential part for test case at hand.

This test input are processed by adding boilerplate prefix ('configuration someConfiguratorName'), and then parsed into an instance of our metamodel.

Then we select the part of the metamodel instance that are relevant for the test case (the parameter object in this case)

2. Act part
Here we call the .compileParameterLink which is the method being tested here.

3. Assert part
We test the returned string with an expected, string. Since it is html we cant compile it. The test case asserts not only that the semantics of the html is correct, but also that the generated is human readable and indented correctly.

```
@Test
	def void testParameterLink_RendersAsListItem(){
		
		'{parameter test values (0;10)}'.addPrefix.parse.firstParam
		.compileParameterLink
		.assertHtmlWithExpectedOutput(
		'''<li>
		  <label for="test-param" >test:</label>
		      <select id="test-param" data-bind="options: test.choices,
selectedOptions: test.value,optionsCaption:'Choose...'"></select>
		        <p class="validationMessage" data-bind="validationMessage:
test.value"></p>
		</li>
		''')
		
	}
```

All boiler plate code are put in an abstract base class, and the the actual test class contains only a reference to class under test, and the test cases.

```
class TestDynamicJqmHtmlGenerator extends BaseTestJqmGenerator{
	
	@Inject extension JqmHtmlGenerator //Sut
	
	@Test
	def void testConfigDescription_rendersAsMainSection(){
		
		'configurator app "main app description"{}'.parse
		.renderAppDescription
		.assertHtmlWithExpectedOutput(
		'''<section class="main-description">
		    main app description
		  </section>
		  ''')
		
	}
```