MDSD Individual Portfolio,
External DSL for generating Spring Boot projects

Frederik Ralskov Holm, fholm16@student.sdu.dk

Source code and DSL program example available on GitHub:
`https://github.com/hrHolm/mdsd-individual-portfolio`

June 15, 2020

Contents

1 Introduction

This individual portfolio is made as part of the course, Model-Driven Software Development Project, and describes extensions made on a group project. The focus is to make a Domain Specific Language (DSL) which is a specialized language used for a specific purpose, in this case being generating Spring Boot projects based on user provided domain models.

Spring is a Java framework that helps developers make quick, easy and safe applications, whereas Spring Boot focuses on streamlining this process further, allowing the user to easily create stand-alone Spring applications. Spring applications often require a similar structure for the program flow (controller→ service→ repository) for each domain model class. The advantage of a DSL here, is that the language can focus on generating this structure, which can ease the development further and ensure consistency of both design and legal framework code.

The project is made with Xtext, which is a language workbench for creating External DSLs, which takes care of lexing, parsing and even name resolving, via declarative specifier rules or new scoping rules, effectively easing DSL development.

Parts of this report have been adapted from the group report, including the description of the original starting point in section 2 and the metamodel seen in appendix B.

2 Starting Point from Group Project

This section briefly explains the starting point from the original group project. The original capabilities, grammar, metamodel, and scoping/validation/generation are described.

In the original project is is possible to specify the package name of the generated project, its domain model and corresponding service methods. Each model can have a number of fields of different types, and inheritance is also supported between Models. Since the models will act as a representation of a database table, an ID must be present (however not for subclasses). An invariant (something which must hold true) can be set on fields of a model, which in the group project only consists of supporting restricting lengths of strings. A repository interface (database query abstraction) is generated for each model.

Services are defined separately, so every model does not necessarily have a corresponding service. For each service a controller is generated, which opens up the use of the service methods via a REST API. The user is also able to easily create CRUD (Create, Read, Update and Delete) operations by specifying "[C R U D]" as part of their service, the user is able to choose any combination of letter they want, and in any order. In addition to this, they can also define their own unique methods. For unique methods, an auto-generated comparison can be made (implemented in an abstract class), such that a certain argument variable should be less-than, bigger-than or equal to a certain field variable (of the specific Service's model). The user can override the methods and provide their own implementations (the generation gap pattern). Furthermore, the user can specify whether or not the method should be a local method (that is, not made available as an REST API endpoint), a GET, POST, PUT or DELETE HTTP request.

An abstract grammar which is based on these capabilities can be seen on Figure 3 in Appendix A. As can be seen, only one project is defined per DSL program. The original metamodel was modeled with UML based on the generated classes of the Xtext grammar and can be seen on Figure 4 in Appendix B. Here, it can be noticed how a list of models and services are unnecessarily put into a new object, which could be simpler.

Regarding implementation, a single scoping rule was created to handle access to field variables from a super class. If the user specifies a comparison restriction on a field variable under the services, Xtext needs to know how to access both the field variables of the class and any of the fields on its super class. The scoping rule looks at the current context, and in the case of a comparison, it looks at the right "Field", find the "Model" that contains that field, and adds all the fields of that model to a list of valid candidates, together with the parent class's fields. This allows the user to reference field variables from a parent object with correct highlighting and syntax in the IDE.

Furthermore, multiple validation rules were created; ensuring correct usage of the CRUD list, ensuring no cycling in entity hierarchy for models, ensuring each model has an ID field (or its superclass if present), lastly ensuring type compatibility in the comparison for services. Lastly, generation of the different parts of a project was split into multiple generators; SpringBoardGenerator, ModelGenerator, RepositoryGenerator, ServiceGenerator and ControllerGenerator.

3 Individual Extension

This section explains the work done individually, by first introducing an overview of the extensions/improvements done, followed by details of the new grammar, metamodel and design considerations and implementation when working in Xtext.

This individual extension adds support for defining abstract Spring Boot projects from which sub-projects can inherit and alter if necessary. These abstract projects, which we will call 'templates', defines partial data models and services, and should be referable by sub-projects if imported. This is valuable, since we often see similar implementations of key parts in web application e.g. user management, product management, among others.

In addition to this, some improvements to the original project is done by remedying shortcomings, i.e. introducing a more general approach to invariants on models (instead of only supporting string lengths) to allow combinations of boolean logic, comparisons and expressions. Lastly some small adjustments is made by cleaning up the metamodel and simplifying some generation steps.

3.1 Updated Grammar

Extended Backus-Naur Form (EBNF) is used to express the abstract grammar found on Figure 1. This extended form allows further meta symbols in the grammar, such as restrictions in multiplicity, optionals and parenthesis.

SpringBoard ::= (*Template* | *Project*)+
Template ::= 'template' ID 'models' ':' *Model*+ 'services' ':' *Service*+
Project ::= 'project' ID 'package' ':' *Package* ('uses' $ID^{Template}$)? 'models' ':' *Model*+ 'services' ':' *Service*+
Package ::= ID ('.' *Package*)?
Model ::= (('extension' 'of' ID^{Model})? | ID *Inherits*?) '{' *Field** '}'
Inherits ::= 'inherits' ID^{Model}
Field ::= ID ':' *Type Invariant*?
Type ::= 'string' | 'datetime' | 'long | 'int' | 'bool' | 'float' | ID^{Model} | 'list' 'of' *Type* | 'ID'
Invariant ::= '[' *BoolLogic* ']'
BoolLogic ::= *ExpComp* | '(' BoolLogic ')' | *BoolLogic* '||' *BoolLogic* | *BoolLogic* '&&' *BoolLogic*
ExpComp ::= *Exp Operator Exp*
Exp ::= *Prim* | *Exp* ('+' | '-') *Exp* | *Exp* ('*' | '/') *Exp*
Prim ::= *INT* | *STRING* | ('true' | 'false') | ID^{Field} | '(' Exp ')'
Operator ::= '<' | '>' | '=' | '<=' | '>=' | '<>'
Service ::= ('extension' 'of')? ID^{Model} '{' *CRUD*? *Method** '}'
CRUD ::= '[' 'C'? 'R'? 'U'? 'D'? ']'
Method ::= *Request* ID *Input* ':' *Type RES*
Request ::= 'local' | 'POST' | 'GET' | 'PUT' | 'DELETE'
RES ::= '{' *Comparison* '}'
Comparison ::= *ARGS Operator Field*
Input ::= *Input* '(' *ARGS*? ')'
ARGS :: = ID ':' *Type* (',' *ARGS*)?

Figure 1: The updated grammar written in Extended Backus-Naur Form (EBNF).

The new grammar allows a user to define multiple declarations, either of type 'Template' or 'Project', where 'Project' can declare which templates are imported with the 'uses' keyword, and the 'Model' and 'Service' rules can be started with the 'extension of' keyword to alter a template model/service. Furthermore, the 'Invariant' rule has been fully rewritten to allow boolean logic (AND/OR) defined as the 'BoolLogic' rule. You are also able to write comparisons and expressions, which can also be added/subtracted and/or multipled/divided. Supported types are integers, strings and booleans, and you can reference a variable that you wish to introduce an invariant in its setter-method.

Note, this grammar will not translate directly to Xtext since it has been written left-recursively, which Xtext cannot handle given its top-down parsing strategy. In addition to this, associativity and

precedence is not established here either. The solution to these issues during parsing will be expanded upon in the implementation section, 3.3.4.

3.2 Example of a DSL Program

The full DSL program example can be seen in Appendix C, however small snippets is used in this section to highlight the new additions, which can be seen on Listing 1. As can be seen, invariants support logical conjunctions, comparisons and expressions (see line 18). Furthermore, an example of extending a template can be seen, where the Library project makes alterations to the template UserManagement's User model and service. In the full example, the Shop project uses more than one template which is also supported.

Listing 1: Partial program example.

```
1  template UserManagement
2    models :
3      User {
4        id : ID
5        name : string [name <> "" && name <> "admin"]  //example of new requirement support
6      }
7    services :
8      User {
9        [ C R D U ]
10       POST auth() : bool
11       local makeAdmin( user : User ) : bool
12     }
13 project Library
14   package : dk.sdu.mmmi.library
15   uses UserManagement
16   models :
17       extension of User {
18         age : int [(age > 13 + 1 && age < 109 - 1) || age < 100 / 1 && age * 1 > 3 && age <> 0] //example of new requirement support
19         loans : List of Loan
20       }
21       //...
22   services :
23       extension of User {
24         [ R D ] // example of shadowing
25         local makeAdmin() : bool // example of shadowing
26         local updateUser(user : User) : bool
27       }
28       //...
```

3.3 Design Considerations and Implementation

This section initially details the metamodel and is afterward divided into subsections describing the design and implementation of the different extensions made on the project.

3.3.1 Metamodel

A view of the metamodel based on the Xtext grammar can be seen on Figure 2. The main difference compared to the old metamodel lies in the new hierarchy of the SpringBoard holding declarations of both Projects and Templates, and the new Exp and all of its sub-classes. When working with the metamodel, the 'declarations'-list can be filtered when iterating through it, in order to get instances of Templates and Projects. Furthermore, by having 'Exp' as a super-class, when generating expressions, Xtend's dispatch method feature can be used extensively, to allow recursive use of the same method for different sub-classes, this is further expanded upon in section 3.3.4. The Xtext grammar that this metamodel is based on, can be found in Appendix D.

3.3.2 Cleaning up the Metamodel and Generation

As mentioned in section 2, the classes 'Models' and 'Services' could be removed and just have the lists inside the project object instead, which have been done by removing the terminal rules of Models and Services and instead pasting their definitions up to the 'Project' rule instead, removing the unnecessary step.

Furthermore, the ControllerGenerator originally had both a service and a model as parameters, and would be used when iterating through models, and then have its own loop iterating through services until a pair was seen where the service model was the same as the current model - this is

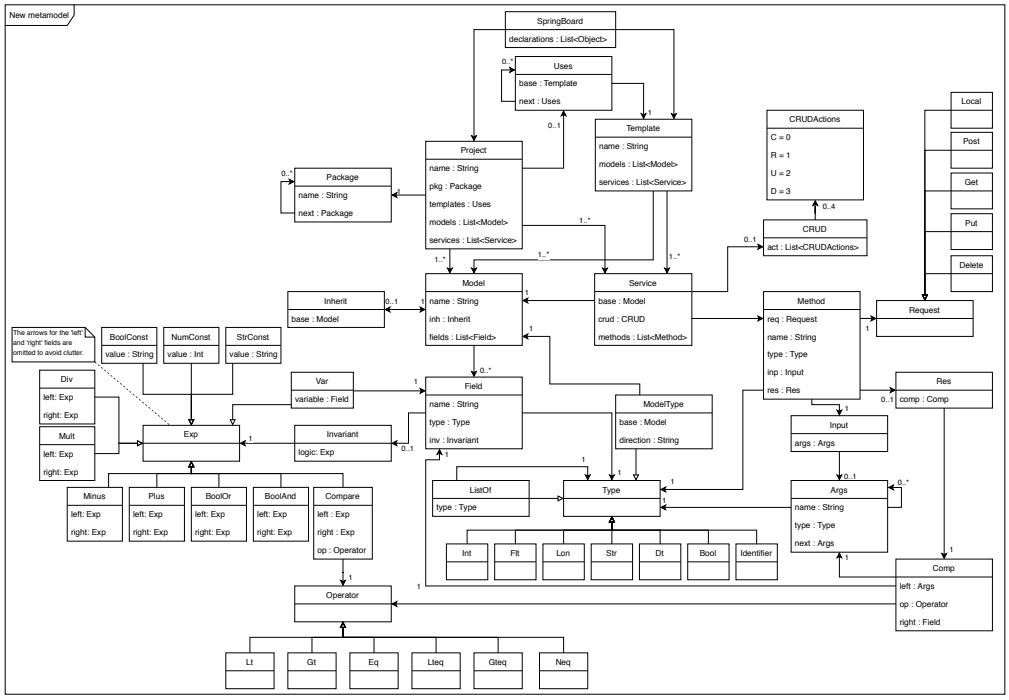

Figure 2: UML representation of the new metamodel.

unnecessary since a service holds a reference to the model it is based on in the metamodel. This has been corrected, which has simplified iteration of the metamodel. The ControllerGenerator is now called as part of running through a the service list, which can be seen in Appendix G in Listing 10 on line 42. The updated code of the ControllerGenerator has been included in Listing 13.

3.3.3 Adding Abstract Projects (Templates)

As mentioned in the introduction in section 3, the user is able to define abstract partial spring projects and use them (and extend them, if necessary) in other projects.

The introduction of templates meant sub-projects must be able to refer to them, which lead to a new scoping method, given Xtext cannot automatically provide object references for different nested contexts (templates and projects). The scoping should ensure that references to template models only are present if the template is imported. The new scoping method can be seen on Listing 2. The return statement on line 12 has two parameters, where the first is what is considered an inner scope, and the second parameter is the outer scope. It should be noted that official documentation on the behavior is lacking, but having the template models as outer scope ensured that the projects own definitions came first when looking for a reference and omitted template references if the template was not imported. This is important when you don't want to use a template, but want to use the same name of the models, and makes sense in the context of projects, since their namespace should not intertwine unlike typical java classes. The overwritten getScope-method can be seen in Appendix E in Listing 7, which based on model reference, will call the specific scope method when we are in a Project context.

Listing 2: The scoping method called when referencing a model, which makes references to models from templates visible, if and only if the template is imported.

```
1  def protected IScope scopeForModelReference(EObject context, EReference reference) {
2    val springBoard = EcoreUtil2.getContainerOfType(context, SpringBoard)
3
4    val templateModels = new ArrayList<Model>
5    for (Template t : springBoard.declarations.filter(Template).toList) {
6      templateModels.addAll(t.models)
7    }
8
```

```
9     val project = EcoreUtil2.getContainerOfType(context, Project)
10    val innerModels = project.models.filter[e | e.name !== null]
11
12    return Scopes.scopeFor(innerModels, Scopes.scopeFor(templateModels))
13  }
```

This scoping rule actually avoids any potential illegal references to templates from a Project's perspective, since they will only be referable if imported. This enables the user to write their own Models with the same names, without the Xtext IDE linking to the wrong reference, e.g. if a new User model was made, if another model in that project had an attribute with the User type, it would pick the project's own definition.

When extending the new templates, multiple new validation rules were necessary in order to maintain a correct project inheritance. This is done following the principle of 'loose grammar, strict validation' (which is further described in section 3.3.4) since when referencing a model during extension, the name resolver itself merely looks for that model reference, but does not check if it's from a template specifically. Given how the scoping rule works, validation in this case focus on cases where templates are imported or if templates are missing if trying to extend something. All new validation rules can be found in appendix F, with added comments on their individual concern. Since we need to be aware of the context these references reside in, the helper method from xtext.EcoreUtil2, getContainerOfType(EObject) is used. This allows us to know which templates have been imported in the current Project-context, if any. By using validation over scoping regarding accessibility, we can provide better error information, since we avoid the default "couldn't resolve reference to..." when a reference is not in scope. As an example, this is done when when a user tries to extend a template they haven't imported - the validation rule iterates over potential templates and let's the user know they haven't imported them if present.

Because the templatesø definitions and any extensions to them should be considered the same object, the handling of this was chosen to be done in the generation part, where the metamodel can be iterated through, and any definitions can be combined or shadowed resulting in model-aware generation. During generation template definitions should be shadowed if the same attribute/method name is used in 'extension of' models or services. In order to do this, the model is changed at runtime, since we are combining templates with the projects, before passing it on to the generators. This meant almost no changes were needed on the individual generators, but only changes to the upper level (e.g. running through a list of projects, instead of only one project). Listing 3 showcases this, from lines 12 - 18 calls to separate methods which handles the aforementioned combination handling, the implemantation of these methods can be found in Appendix G in Listing 10.

Listing 3: Showcase of new generation steps.

```
1   package dk.sdu.mmmi.generator
2   // omitted imports
3   class SpringBoardGenerator extends AbstractGenerator {
4       // omitted setup
5     override void doGenerate(Resource resource, IFileSystemAccess2 fsa, IGeneratorContext context) {
6       val model = resource.allContents.filter(SpringBoard).next
7       for (Project springProject : model.declarations.filter(Project)) {
8         val projectName = springProject.name
9         val packName = createPackageName(springProject.pkg)
10        generateSpringProjectStructure(fsa, packName, projectName)
11
12        var projectModels = springProject.models.toList
13        var projectServices = springProject.services.toList
14        if (springProject.templates !== null) { // handling of the new templates are necessary, altering the metamodel at run-time
15          val usedTemplates = getTemplateList(springProject.templates)
16          projectModels = incorporateTemplateModels(projectModels, usedTemplates)
17          projectServices = incorporateTemplateServices(projectServices, usedTemplates)
18        }
19        // traversal of the new projectServices list to generate services and controllers
20        // traversal of the new projectModels list to generate models and repositories
21      }
22    }
23    // omitted methods
24  }
```

3.3.4 Implementing General Invariants on Models

As mentioned in the introduction in section 3, invariants on models must allow combinations of boolean logic, comparisons and expressions.

The abstract grammar from section 3.1 had to be refined in order for Xtext to be able to parse it - the full refinement of the grammar can be found in Appendix D. Firstly, the rules following 'Exp' have been left-factorized in order to avoid left recursion during parsing. This also implicitly describes the precedence in the language, given the thing that recurses (e.g. 'MultDiv' in 'PlusMinus') delegates to the rule with the next higher precedence. In the Xtext grammar, this example can be seen on line 52 - 56.

Regarding associativity, the 'BoolLogic' which in this case supports *Or* and *And*, is right associative which is fine in that context, however this is different for the expressions, since they have to be left-associative. In order to achieve this, we instruct Xtext to create a Node-object for each rule which are inside curly brackets. In the Xtext grammar, an example of this could be on line 46 - 50 where the Exp rule will create a 'Compare'-node object and set its left as the current object Xtext is parsing. This will in turn generate a tree structure that ensures left associativity when generating expressions, which is useful for transformer based generation.

Furthermore, the metamodel is changed by letting all the rules return as 'Exp'. This enables simple type-checking for all combinations of expressions, in addition to generate whole invariants via dispatch methods.

After writing the Xtext rules for the invariants, the first step was to generate code from it. As mentioned earlier, given the way the rules were set up, this was easily achievable with a transformer-based generation approach, relying on the structure of the model, via dispatch methods. The result can be seen on Listing 4, where a single call to the dispatch method, genExp, will result in the compiler inferring a synthetic dispatcher method, combining all dispatch methods of that name, which can be seen in Appendix G in the class, InvariantGenerator, in Listing 11. Given the grammar allows multiple types (integers, strings and booleans) the user can potentially generate illegal code, e.g. multiplying an integer with a boolean. Therefore new validation rules were necessary.

Listing 4: This method is called from the model-generation if an invariant is present on a Model. Model traversal is defined as a single call to dispatch methods, providing transformer generation.

```
1 // located in ModelGenerator.xtend
2 def CharSequence generateInvariant(Field f)'''
3   if ( f.inv.logic.genExp ) { // call to dispatch methods
4     this._ f.name  =  f.name ;
5   } else {
6     throw new IllegalArgumentException("Requirement not satisfied.");
7   }
8   '''
```

Domain-specific (i.e. domain of logic and expressions in this case) validation rules were created where each Exp-node's (which can be any of the rules returning as Exp) left and right field are checked to see if the same type is used. This meant for BoolAnd, and BoolOr, sub-expressions must be of type boolean. For PlusMinus and MultDiv both have to be integer types. And lastly for comparisons (Compare), both have to be the same type, booleans cannot be compared and strings only support equality comparisons. All validation implementation made can be seen in Appendix F.

By having everything return as Exp, we can return its type by passing the object, and based on an 'instanceof' switch case, it returns an enum called ExpressionsType. This method is called typeFor, and can be used as an extension method simplifying the validation methods. All of this is located in the LogicTyping class in Listing 8. If the instance is a variable, it goes into a new switch case which does the same thing, figuring out the variable's type. With this extension method, the left and right field can be compared and if any of the rules mentioned earlier are broken an error is shown to the user - these validation methods can be seen in Listing 9.

As mentioned, this avoids generating any illegal code, which is a nice property to have as the user is guaranteed functioning invariants. The grammar could have been written differently to enforce legal types at grammar level, however as suggested by Bettini [1, page 207], the good practice of "loose grammar, strict validation" in Xtext DSL implementations have been followed (which is also the case regarding the template implementation). This meant the grammar ended up more simple, since any special cases were left for validation.

References

[1] L. Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing, 2016.

A Original Group BNF

The original abstract grammar can be seen on Figure 3.

SpringBoard ::= 'package' ':' *Package Models Services*
Package ::= ID ('.' *Package*)?
Models ::= 'models' ':' *Model+*
Model ::= ID *Inherits*? '{' *Field** '}'
Inherits ::= 'inherits' ID^{Model}
Field ::= ID ':' *Type Invariant*?
Type ::= 'string' | 'datetime' | 'long' | 'int' | 'bool' | 'float' | ID^{Model} | 'list' 'of' *Type* | 'ID'
Invariant ::= '[' *Property Operator INT* ']'
Property ::= 'length'
Operator ::= '<' | '>' | '=' | '<=' | '>=' | '<>'
Services ::= 'services' ':' *Service+*
Service ::= ID^{Model} '{' *CRUD*? *Method** '}'
CRUD ::= '[' 'C'? 'R'? 'U'? 'D'? ']'
Method ::= *Request* ID *Input* ':' *Type RES*
Request ::= 'local' | 'POST' | 'GET' | 'PUT' | 'DELETE'
RES ::= '{' *Comparison* '}'
Comparison ::= *ARGS Operator Field*
Input ::= *Input* '(' *ARGS*? ')'
ARGS :: = ID ':' *Type* (',' *ARGS*)?

Figure 3: Original abstract grammar

B Original Group Metamodel

The metamodel seen on Figure 4 is taken from original group report. The classes which have empty attribute lists are the result of using instanceof when iterating through the metamodel.

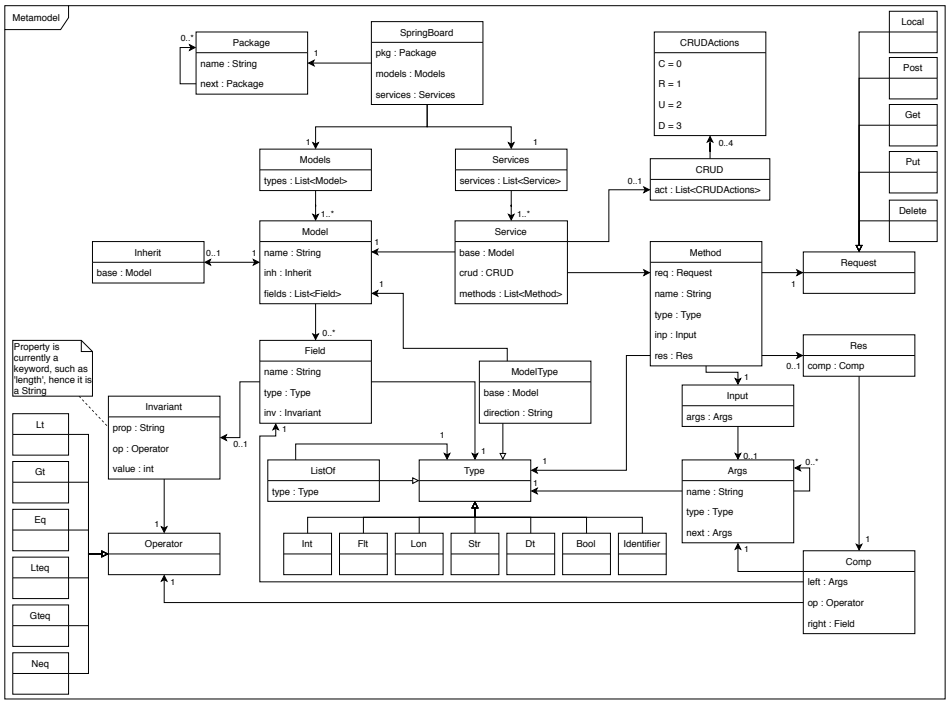

Figure 4: UML representation of the original group metamodel.

C Full Program Example of Updated DSL

The example program with legal code with the individual extension can be seen on Listing 5.

Listing 5: Full program example.

```
template UserManagement
  models :
    User {
      id : ID
      name : string [name <> "" && name <> "admin"]
    }
  services :
    User {
      [ C R D U ]
      POST auth() : bool
      local makeAdmin( user : User ) : bool
    }
template ProductSystem
  models :
    ProductLineItem {
      id : ID
      product : Product*
    }
    Product {
      id : ID
      name : string
      price : float
    }
  services :
    ProductLineItem {
      [ C R U D ]
    }
    Product {
      [ C R U D ]
    }
project Library
  package : dk.sdu.mmmi.library
  uses UserManagement
  models :
      extension of User {
        age : int [(age > 13 + 1 && age < 109 - 1) || age < 100 / 1 && age * 1 > 3 && age <> 0] //example of new requirement support
        loans : List of Loan
      }
      Loan {
          id : ID
          startDate : datetime
          endDate : datetime
          returned : bool
          user : User
          loaned : Media*
      }
      Media {
        id : ID
          name : string [name <> ""]
          published : datetime
      }
      Book inherits Media {
          isbn13 : string
          pageCount : int [pageCount > 0]
          language : string
      }
      Paper inherits Book {
        doi : string
      }
  services :
      extension of User { //example of both shadowing and new methods
        [ R D ]
        local makeAdmin() : bool
        local updateUser(user : User) : bool
      }
      Media { [ C R U D ] }
      Loan {
          [C R U ]
          GET overdueLoans ( currDate : datetime ): List of Loan  { currDate > endDate }
          GET specificMediaLoan ( media : Media ): List of Loan
      }
project Shop
  package : dk.sdu.mmmi.mdsd.shop
  uses UserManagement, ProductSystem
  models :
    Order {
      id : ID
      orderNumber : long
      products : List of ProductLineItem
      customer : User
      date : datetime
    }
    extension of User {
      orders : List of Order
      address : Address
      email : string
      phoneNumber : string
    }
    Address {
      id : ID
      streetName : string
      houseNumber : int
      apartmentDetails : string
      zipCode : int
    }
    services :
      Order {
```

```
    [ C R U ]
  }
```

D Xtext Grammar

The full Xtext grammar can be seen on Listing 6.

Listing 6: The full Xtext specification describing the grammar of the DSL.

```
grammar dk.sdu.mmmi.SpringBoard with org.eclipse.xtext.common.Terminals

generate springBoard "http://www.sdu.dk/mmmi/SpringBoard"

SpringBoard:
  declarations+=(Template | Project)+
;

Template :
  'template' name=ID 'models' ':' models+=Model+ 'services' ':' services+=Service+
;

Project:
  'project' name=ID 'package' ':' pkg=Package ('uses' templates=Uses)? 'models' ':' models+=Model+ 'services' ':' services+=Service+
;

Uses:
  base=[Template] (',' next=Uses)?
;

Package:
  name=ID ('.' next=Package)?
;

Model:
  (('extension' 'of' base=[Model]) | name=ID inh=Inherit?) '{' fields+=Field* '}'
;

Field:
  name=ID ':' type=Type inv=Invariant?
;

// Improved invariants START
Invariant:
  '[' logic=BoolLogic ']'
;

BoolLogic returns Exp:
  Conjunction ({BoolOr.left=current} '||' right=BoolLogic)?
;

Conjunction returns Exp:
  Exp ({BoolAnd.left=current} '&&' right=Conjunction)?
;

Exp:
  PlusMinus (
    {Compare.left=current} op=Operator right=PlusMinus
  )*
;

PlusMinus returns Exp:
  MultDiv (
    ({Plus.left=current} '+' | {Minus.left=current} '-') right=MultDiv
  )*
;

MultDiv returns Exp:
  Prim (
    ({Mult.left=current} '*' | {Div.left=current} '/') right=Prim
  )*
;

Prim returns Exp:
  {NumConst} value=INT |
  {StrConst} value=STRING |
  {BoolConst} value=('true'|'false') |
  {Var} variable=[Field] |
  '(' BoolLogic ')'
;
// Improved invariants END

Operator returns Operator:
  {Lt} '<' | {Gt} '>' | {Eq} '=' | {Lteq} '<=' | {Gteq} '>=' | {Neq} '<>'
;

Type returns Type:
  {Str} 'string' | {Dt} 'datetime' | {Lon} 'long' | {Int} 'int' | {Bool} 'bool' | {Flt} 'float' | {ModelType} (base=[Model]
        direction='*'?) | {ListOf} ('List' 'of' type=Type) | {Identifier} 'ID'
;

Inherit:
  'inherits' base=[Model]
;

Service:
  (extension?='extension' 'of')? base=[Model] '{' crud=CRUD? methods+=Method* '}'
;

```

```
CRUD:
  '[' act += CRUDActions* ']'
;

enum CRUDActions:
  C | R | U | D
;

Method:
  req=Request name=ID inp=Input ':' type=Type res=Res?
;

Request returns Request:
  {Local} 'local' | {Post} 'POST' | {Get} 'GET' | {Put} 'PUT' | {Delete} 'DELETE'
;

Res:
  '{' comp=Comp '}'
;

Comp:
  left=[Args] op=Operator right=[Field]
;

Input:
  {Input} '(' args=Args? ')'
;

Args:
  name=ID ':' type=Type (',' next=Args)?
;
```

E Xtend Scoping

The custom scoping implemenation can be seen on Listing 7.

Listing 7: Custom scoping implementation.

```
/*
 * generated by Xtext 2.20.0
 */
package dk.sdu.mmmi.scoping

// omitted imports

/**
 * This class contains custom scoping description.
 *
 * See https://www.eclipse.org/Xtext/documentation/303_runtime_concepts.html#scoping
 * on how and when to use it.
 */
class SpringBoardScopeProvider extends AbstractSpringBoardScopeProvider {

  override IScope getScope(EObject context, EReference reference) {
    switch reference {
      case reference == Literals.COMP__RIGHT: {
        return scopeForTypeReference(context, reference)
      }
      case reference == Literals.MODEL__BASE: {
        return scopeForModelReference(context, reference)
      }
      case reference == Literals.MODEL_TYPE__BASE: {
        if (EcoreUtil2.getContainerOfType(context, Project) !== null) { // we are in project context
          return scopeForModelReference(context, reference)
        }
      }
      case reference == Literals.SERVICE__BASE: {
        if (EcoreUtil2.getContainerOfType(context, Project) !== null) { // we are in project context
          return scopeForModelReference(context, reference)
        }
      }
    }
    return super.getScope(context, reference)
  }

  def protected IScope scopeForModelReference(EObject context, EReference reference) {
    val springBoard = EcoreUtil2.getContainerOfType(context, SpringBoard)

    val templateModels = new ArrayList<Model>
    for (Template t : springBoard.declarations.filter(Template).toList) {
      templateModels.addAll(t.models)
    }

    val project = EcoreUtil2.getContainerOfType(context, Project)
    val innerModels = project.models.filter[e | e.name !== null]

    return Scopes.scopeFor(innerModels, Scopes.scopeFor(templateModels))
  }


  def protected IScope scopeForTypeReference(EObject context, EReference reference) {
    var methods = EcoreUtil2.getContainerOfType(context, Method);
    val candidates = new ArrayList<Field>

    var type = methods.type;

    if (type instanceof ListOf) {
      type = (type as ListOf).type
```

```
    }
    if (type instanceof ModelType) {
      var model = (type as ModelType)
      candidates.addAll(model.base.getFields.filter(Field))
      if (model.base.inh !== null) {
        candidates.addAll(model.base.inh.base.getFields.filter(Field))
      }
    } else {
      return super.getScope(context, reference)
    }
    return Scopes.scopeFor(candidates)
  }

}
```

F Xtend Validation

Validation code is split into two classes, namely LogicTyping (see Listing 8) and the custom validator class (see Listing 9).

Listing 8: The LogicTyping class which takes care of returning the type of individual sub expressions.

```
package dk.sdu.mmmi.validation

// omitted imports

class LogicTyping {

  def ExpressionsType typeFor(Exp e) {
    switch (e) {
      StrConst: STRING_TYPE
      NumConst: INT_TYPE
      BoolConst: BOOL_TYPE
      Mult: INT_TYPE
      Div: INT_TYPE
      Plus: INT_TYPE
      Minus: INT_TYPE
      Compare: BOOL_TYPE
      BoolAnd: BOOL_TYPE
      BoolOr: BOOL_TYPE
      Var: variableType(e)
    }
  }

  def ExpressionsType variableType(Var v) {
    switch (v.variable.type) {
      Bool : BOOL_TYPE
      Str : STRING_TYPE
      Int : INT_TYPE
    }
  }
}

enum ExpressionsType {
  STRING_TYPE,
  INT_TYPE,
  BOOL_TYPE
}
```

Listing 9: The custom validator class including all new validations implemented, but omitting validation methods from the group project.

```
/*
 * generated by Xtext 2.20.0
 */
package dk.sdu.mmmi.validation

// omitted imports

/**
 * This class contains custom validation rules.
 *
 * See https://www.eclipse.org/Xtext/documentation/303_runtime_concepts.html#validation
 */
class SpringBoardValidator extends AbstractSpringBoardValidator {

  // omitted validations from group project

  /**
   * This rule from the group project was changed, since this check only needs to happen when a model is not an extension -
   * this way it is always ensured that an ID is present, since the template's model must have ID's
   */
  @Check
  def checkOnlySingleIdForModel(Model model) {
    if (model.base === null) { // new addition
      if (model.inh !== null) {
        if (!model.fields.filter[f|f.type instanceof Identifier].empty) {
          error("Subclasses must not have an ID field.", SpringBoardPackage.Literals.MODEL__FIELDS)
        }
      } else {
        if (model.fields.filter[f|f.type instanceof Identifier].size != 1) {
          error("A model must have a single ID field.", SpringBoardPackage.Literals.MODEL__NAME)
        }
```

```
32        }
33      }
34    }
35
36    /* ------------- Template Validations ------------- */
37
38    /**
39     * Make sure unique naming is done when importing templates
40     */
41    @Check
42    def checkUniqueNamingFromImportedTemplates(Model model) {
43      val contextProject = (model as EObject).getContainerOfType(Project)
44      if (contextProject.templates !== null && model.name !== null) {
45        for (t : contextProject.templates.templateList) {
46          for (m : t.models) {
47            if (m.name.equalsIgnoreCase(model.name)) {
48              error('''The name:  model.name  is already used by the imported template:  t.name ''',
                    SpringBoardPackage.Literals.MODEL__NAME)
49            }
50          }
51        }
52      }
53    }
54
55    /**
56     * Make sure unique naming is done when importing templates
57     */
58    @Check
59    def checkUniqueNamingFromImportedTemplates(Service service) {
60      val contextProject = (service as EObject).getContainerOfType(Project)
61      if (contextProject.templates !== null && !service.isExtension) {
62        for (t : contextProject.templates.templateList) {
63          for (m : t.models) {
64            if (m.name.equalsIgnoreCase(service.base.name)) {
65              System.out.println("what")
66              error('''The name:  service.base.name  is already used by the imported template:  t.name ''',
                    SpringBoardPackage.Literals.SERVICE__BASE)
67            }
68          }
69        }
70      }
71    }
72
73    /**
74     * Checks if the necessary templates have been imported when extending a model
75     */
76    @Check
77    def checkTemplateImport(Model model) {
78      if (model.base !== null && !model.base.isImported(model)) {
79        error('''The model:  model.base.name  has not been imported from a template.''', SpringBoardPackage.Literals.MODEL__BASE)
80      }
81
82    }
83
84    /**
85     * Checks if the necessary templates have been imported when extending a model
86     */
87    @Check
88    def checkTemplateImport(Service service) {
89      if (service.extension && !service.base.isImported(service)) {
90        error('''The model:  service.base.name  has not been imported from a template.''', SpringBoardPackage.Literals.SERVICE__BASE)
91      }
92
93    }
94
95    /**
96     * Checks if the necessary models are accessible if a model is used as a type
97     */
98    @Check
99    def checkIfModelIsAccessableAsType(ModelType modelType) {
100     val contextProject = (modelType as EObject).getContainerOfType(Project)
101     if (modelType.base !== null && !contextProject.models.contains(modelType.base) && !modelType.base.isImported(modelType)) {
102       error('''The model:  modelType.base.name  is not accessable.''', SpringBoardPackage.Literals.MODEL_TYPE__BASE)
103     }
104   }
105
106   protected def boolean isImported(Model base, EObject context) {
107     val contextProject = context.getContainerOfType(Project)
108     if (contextProject.templates === null) { // null-safety
109       return false
110     }
111     for (t : contextProject.templates.templateList) {
112       if (t.models.contains(base)) {
113         return true
114       }
115     }
116     return false
117   }
118
119   def List<Template> getTemplateList(Uses uses) {
120     var usesIter = uses
121     var List<Template> templateList = new ArrayList
122     templateList.add(usesIter.base)
123
124     while (usesIter.next !== null) {
125       usesIter = usesIter.next
126       templateList.add(usesIter.base)
127     }
128     return templateList
129   }
130
131   /* ------------- Logic Validations ------------- */
132
133   @Inject extension LogicTyping logicTyping
134
```

```
135   def private ExpressionsType getTypeAndCheckNotNull(Exp exp, EReference reference) {
136     var type = exp.typeFor
137     if (type === null)
138       error("Null type", reference)
139     return type;
140   }
141
142   def private checkExpectedType(Exp exp, ExpressionsType expectedType, EReference reference) {
143     val actualType = getTypeAndCheckNotNull(exp, reference)
144     if (actualType != expectedType) {
145       error("expected " + expectedType + " type, but was " + actualType, reference)
146       }
147   }
148
149   @Check
150   def checkType(BoolAnd and) {
151     checkExpectedType(and.left, ExpressionsType.BOOL_TYPE, SpringBoardPackage.Literals.BOOL_AND__LEFT)
152     checkExpectedType(and.right, ExpressionsType.BOOL_TYPE, SpringBoardPackage.Literals.BOOL_AND__RIGHT)
153   }
154
155   @Check
156   def checkType(BoolOr or) {
157     checkExpectedType(or.left, ExpressionsType.BOOL_TYPE, SpringBoardPackage.Literals.BOOL_OR__LEFT)
158     checkExpectedType(or.right, ExpressionsType.BOOL_TYPE, SpringBoardPackage.Literals.BOOL_OR__RIGHT)
159   }
160
161   @Check
162   def checkType(Minus minus) {
163     checkExpectedType(minus.left, ExpressionsType.INT_TYPE, SpringBoardPackage.Literals.MINUS__LEFT)
164     checkExpectedType(minus.right, ExpressionsType.INT_TYPE, SpringBoardPackage.Literals.MINUS__RIGHT)
165   }
166
167   /**
168    * String concatenation is not supported in this domain
169    */
170   @Check
171   def checkType(Plus plus) {
172     checkExpectedType(plus.left, ExpressionsType.INT_TYPE, SpringBoardPackage.Literals.PLUS__LEFT)
173     checkExpectedType(plus.right, ExpressionsType.INT_TYPE, SpringBoardPackage.Literals.PLUS__RIGHT)
174   }
175
176   @Check def checkType(Compare comparison) {
177     val leftType = getTypeAndCheckNotNull(comparison.left, SpringBoardPackage.Literals.COMPARE__LEFT)
178     val rightType = getTypeAndCheckNotNull(comparison.right, SpringBoardPackage.Literals.COMPARE__RIGHT)
179     if (leftType !== rightType) {
180       error('''Comparisons can only be done on the same type. Current types are  leftType  and  rightType ''',
              SpringBoardPackage.Literals.COMPARE__OP)
181     }
182     if (leftType == ExpressionsType.BOOL_TYPE) {
183       error('''Cannot be a boolean type''', SpringBoardPackage.Literals.COMPARE__LEFT)
184     }
185     if (rightType == ExpressionsType.BOOL_TYPE) {
186       error('''Cannot be a boolean type''', SpringBoardPackage.Literals.COMPARE__RIGHT)
187     }
188     if (leftType == ExpressionsType.STRING_TYPE) { // we can simply check the left here, since left and right will be same type
189       if (!(comparison.op instanceof Eq) && !(comparison.op instanceof Neq)) {
190         error('''Strings can only be compared based on equality''', SpringBoardPackage.Literals.COMPARE__OP)
191       }
192     }
193   }
194 }
```

G Xtend Generation

Only new or changed code regarding code generation is included in this appendix, for the full source code, see the GitHub Repository linked at the front page. Additions has been made into three classes, namely SpringBoardGenerator (see Listing 10), the InvariantGenerator class (see Listing 11) and the ModelGenerator (see Listing 12). Finally, ControllerGenerator was altered, which can be seen on Listing 13.

Listing 10: The SpringBoardGenerator including all new methods for combining templates into subprojects. Methods from the group project have been omitted

```
1   package dk.sdu.mmmi.generator
2
3   // omitted imports
4
5   class SpringBoardGenerator extends AbstractGenerator {
6
7     @Inject extension ServiceGenerator serviceGenerator
8     @Inject extension ModelGenerator modelGenerator
9     @Inject extension RepositoryGenerator repositoryGenerator
10    @Inject extension ControllerGenerator controllerGenerator
11
12    val mavenSrcStructure = "src/main/java/"
13    val mavenTestStructure = "src/test/java/"
14    List<Model> modelsWithSubClasses = new ArrayList<Model>();
15
16    override void doGenerate(Resource resource, IFileSystemAccess2 fsa, IGeneratorContext context) {
17      val model = resource.allContents.filter(SpringBoard).next
18
19      for (Project springProject : model.declarations.filter(Project)) {
20        val projectName = springProject.name
```

```
21        val packName = createPackageName(springProject.pkg)
22
23        generateSpringProjectStructure(fsa, packName, projectName)
24
25        var projectModels = springProject.models.toList
26        var projectServices = springProject.services.toList
27        if (springProject.templates !== null) {
28          val usedTemplates = getTemplateList(springProject.templates)
29          projectModels = incorporateTemplateModels(projectModels, usedTemplates)
30          projectServices = incorporateTemplateServices(projectServices, usedTemplates)
31        }
32
33        for (Model individualModel : projectModels) {
34          if (hasSubclasses(individualModel, springProject)) {
35            modelsWithSubClasses.add(individualModel)
36          }
37        }
38
39        projectServices.forEach [ element |
40          serviceGenerator.createService(fsa, packName, element, projectName);
41          serviceGenerator.createAbstractService(fsa, packName, element, projectName)
42          controllerGenerator.createController(element, fsa, packName, projectName, isASubClass(element.base))
43        ]
44        projectModels.forEach [ element |
45          modelGenerator.createModel(element, fsa, packName, hasSubclasses(element, springProject), projectName)
46          repositoryGenerator.createRepository(element, fsa, packName, modelsWithSubClasses, projectName)
47        ]
48      }
49
50    }
51
52    def List<Service> incorporateTemplateServices(List<Service> projectServices, List<Template> templates) {
53      var List<Service> incorboratedList = new ArrayList
54      val List<Service> allTemplateServices = new ArrayList
55      for (t : templates) {
56        allTemplateServices.addAll(t.services)
57      }
58
59      // to avoid java.util.ConcurrentModificationException
60      var List<Service> tsToRemove = new ArrayList
61      var List<Service> psToRemove = new ArrayList
62
63      // compare all models against each other, in order to determine if any shadowing is necessary
64      for (ts : allTemplateServices) {
65        for (ps : projectServices) {
66          if (ps.base == ts.base) { // an extension has been declared
67            val combinedService = createCombinedService(ps, ts)
68            incorboratedList.add(combinedService)
69            psToRemove.add(ps)
70            tsToRemove.add(ts)
71          }
72        }
73      }
74      allTemplateServices.removeAll(tsToRemove)
75      projectServices.removeAll(psToRemove)
76      incorboratedList.addAll(allTemplateServices)
77      incorboratedList.addAll(projectServices)
78
79      return incorboratedList
80    }
81
82    def createCombinedService(Service extensionService, Service templateService) {
83      var Service combinedService = templateService
84
85      var List<Method> methodsToRemove = new ArrayList
86      var List<Method> methodsToAdd = new ArrayList
87
88      // if CRUD is defined in an extension, it should always overwrite
89      if (extensionService.crud !== null) {
90        combinedService.crud = extensionService.crud
91      }
92
93      for (em : extensionService.methods) {
94        for (tm : templateService.methods) {
95          if (em.name == tm.name) { // shadowing is necessary
96            methodsToRemove.add(tm)
97            methodsToAdd.add(em)
98          } else {
99            methodsToAdd.add(em)
100         }
101       }
102     }
103     combinedService.methods.removeAll(methodsToRemove)
104     combinedService.methods.addAll(methodsToAdd)
105     return combinedService
106   }
107
108   /**
109    * Takes care of combining templates with the project's own models - when a project makes extensions to templates
110    * this method takes care of appending anything new, and shadow the templates' model if the same name is used
111    */
112   def List<Model> incorporateTemplateModels(List<Model> projectModels, List<Template> templates) {
113     var List<Model> incorboratedList = new ArrayList
114     val List<Model> allTemplateModels = new ArrayList
115     for (t : templates) {
116       allTemplateModels.addAll(t.models)
117     }
118
119     // to avoid java.util.ConcurrentModificationException
120     var List<Model> tmToRemove = new ArrayList
121     var List<Model> pmToRemove = new ArrayList
122
123     // compare all models against each other, in order to determine if any shadowing is necessary
124     for (tm : allTemplateModels) {
125       for (pm : projectModels) {
```

```
126         if (pm.base == tm) { // an extension has been declared
127           val combinedModel = createCombinedModel(pm, tm)
128           incorboratedList.add(combinedModel)
129           pmToRemove.add(pm)
130           tmToRemove.add(tm)
131         }
132       }
133     }
134     allTemplateModels.removeAll(tmToRemove)
135     projectModels.removeAll(pmToRemove)
136     incorboratedList.addAll(allTemplateModels)
137     incorboratedList.addAll(projectModels)
138
139     return incorboratedList
140   }
141
142   def Model createCombinedModel(Model extensionModel, Model templateModel) {
143     var Model combinedModel = templateModel
144
145     var List<Field> fieldsToRemove = new ArrayList
146     var List<Field> fieldsToAdd = new ArrayList
147
148     for (ef : extensionModel.fields) {
149       for (tf : templateModel.fields) {
150         if (ef.name == tf.name) { // shadowing is necessary
151           fieldsToRemove.add(tf)
152           fieldsToAdd.add(ef)
153         } else {
154           fieldsToAdd.add(ef)
155         }
156       }
157     }
158     combinedModel.fields.removeAll(fieldsToRemove)
159     combinedModel.fields.addAll(fieldsToAdd)
160     return combinedModel
161   }
162
163   // the template list are defined as a recursive rule
164   def List<Template> getTemplateList(Uses uses) {
165     var usesIter = uses
166     var List<Template> templateList = new ArrayList
167     templateList.add(usesIter.base)
168
169     while (usesIter.next !== null) {
170       usesIter = usesIter.next
171       templateList.add(usesIter.base)
172     }
173     return templateList
174   }
175
176   // omitted methods
177 }
```

Listing 11: The new InvariantGenerator including all dispatch methods for generating expressions.

```
1  package dk.sdu.mmmi.generator
2
3  // omitted imports
4
5  class InvariantGenerator {
6
7    def dispatch CharSequence genExp(BoolAnd logic) '''( logic.left.genExp  &&  logic.right.genExp )'''
8
9    def dispatch CharSequence genExp(BoolOr logic) '''( logic.left.genExp  ||  logic.right.genExp )'''
10
11   def dispatch CharSequence genExp(Compare logic) '''( logic.left.genExp   logic.op.genOp   logic.right.genExp )'''
12
13   def dispatch CharSequence genExp(Plus exp) ''' exp.left.genExp  +  exp.right.genExp '''
14
15   def dispatch CharSequence genExp(Minus exp) ''' exp.left.genExp  -  exp.right.genExp '''
16
17   def dispatch CharSequence genExp(Mult exp) ''' exp.left.genExp  *  exp.right.genExp '''
18
19   def dispatch CharSequence genExp(Div exp) ''' exp.left.genExp  /  exp.right.genExp '''
20
21   def dispatch CharSequence genExp(Var exp) ''' exp.variable.name '''
22
23   def dispatch CharSequence genExp(NumConst exp) ''' exp.value '''
24
25   def dispatch CharSequence genExp(StrConst exp) '''" exp.value "'''
26
27   def dispatch CharSequence genOp(Lt operator) '''<'''
28
29   def dispatch CharSequence genOp(Gt operator) '''>'''
30
31   def dispatch CharSequence genOp(Eq operator) '''=='''
32
33   def dispatch CharSequence genOp(Lteq operator) '''<='''
34
35   def dispatch CharSequence genOp(Gteq operator) '''>='''
36
37   def dispatch CharSequence genOp(Neq operator) '''!='''
38
39 }
```

Listing 12: The changed generateInvariant-method in the ModelGenerator. Again, all unchanged methods are omitted.

```
1  package dk.sdu.mmmi.generator
```

```
2
3  // omitted imports
4
5  class ModelGenerator {
6
7    @Inject extension InvariantGenerator invariantGenerator
8
9    // omitted methods
10
11   def CharSequence generateInvariant(Field f)'''
12   if ( f.inv.logic.genExp ) {
13     this._ f.name  =  f.name ;
14   } else {
15     throw new IllegalArgumentException("Requirement not satisfied.");
16   }
17   '''
18 }
```

Listing 13: The changed ControllerGenerator, such that it only depends on a Service, and not also a Model object.

```
1  package dk.sdu.mmmi.generator
2
3  // omitted imports
4
5  class ControllerGenerator {
6
7    val mavenSrcStructure = "src/main/java/"
8
9    def CharSequence generateController(Service service, String packName, boolean isASubClass) {
10     '''
11 package  packName .controllers;
12 import  packName .models.*;
13 import org.springframework.web.bind.annotation.*;
14 import  packName .services.I service.base.name ;
15 import javax.validation.Valid;
16 import java.util.List;
17 import java.time.LocalDateTime;
18 @RestController
19 public class  service.base.name Controller {
20
21   private I service.base.name   service.base.name.toFirstLower Service;
22
23   public  service.base.name Controller(I service.base.name   service.base.name.toFirstLower Service) {
24       this. service.base.name.toFirstLower Service =   service.base.name.toFirstLower Service;
25   }
26
27    IF service.crud !== null
28   generateCRUDMethods(service)
29    ENDIF
30   generateServiceMethods(service)
31 }
32 '''
33   }
34
35   def createController(Service service, IFileSystemAccess2 fsa, String packName, String projectName, boolean isASubClass) {
36     fsa.generateFile(projectName + "/" + mavenSrcStructure + packName.replace('.', '/') + "/controllers/" + service.base.name +
           "Controller.java",
37       generateController(service, packName, isASubClass)
38     )
39   }
40
41   def generateCRUDMethods(Service service) {
42     '''
43       FOR a : service.crud.act
44         IF a == CRUDActions.C
45          @PostMapping("/api/ service.base.name.toLowerCase ")
46          public  service.base.name  create service.base.name (@Valid @RequestBody  service.base.name
                  service.base.name.toFirstLower ) {
47            return  service.base.name.toFirstLower Service.create( service.base.name.toFirstLower );
48          }
49
50          ENDIF
51          IF a == CRUDActions.R
52          @GetMapping("/api/ service.base.name.toLowerCase /{id}")
53          public  service.base.name  find(@PathVariable Long id) {
54            return  service.base.name.toFirstLower Service.find(id);
55          }
56
57          @GetMapping("/api/ service.base.name.toLowerCase /all")
58          public List< service.base.name > findAll() {
59            return  service.base.name.toFirstLower Service.findAll();
60          }
61
62          ENDIF
63          IF a == CRUDActions.U
64          @PutMapping("/api/ service.base.name.toLowerCase ")
65          @ResponseBody
66          public void update(@RequestBody  service.base.name   service.base.name.toFirstLower ) {
67             service.base.name.toFirstLower Service.update( service.base.name.toFirstLower );
68          }
69
70          ENDIF
71          IF a == CRUDActions.D
72          @DeleteMapping("/api/ service.base.name.toLowerCase /{id}")
73          @ResponseBody
74          public void delete(@PathVariable Long id) {
75               service.base.name.toFirstLower Service.delete(id);
76          }
77
78          ENDIF
79       ENDFOR
```

```
    '''
  }

  def generateServiceMethods(Service service)'''
       FOR m : service.methods.filter[m | !(m.req instanceof Local)]
      @ m.req.showReq Mapping("/api/ service.base.name.toLowerCase / m.name.toLowerCase ")
       m.type.show   m.name ( IF m.inp.args !== null  m.inp.args.show  ENDIF ){
        return   service.base.name.toFirstLower Service. m.name ( IF m.inp.args !== null  m.inp.args.showName  ENDIF );
      }

       ENDFOR
      '''

  def dispatch CharSequence show(Dt dt) '''LocalDateTime'''
  def dispatch CharSequence show(ListOf lo) '''List< lo.type.show >'''
  def dispatch CharSequence show(Str st) '''String'''
  def dispatch CharSequence show(Int in) '''Integer'''
  def dispatch CharSequence show(Lon l) '''Long'''
  def dispatch CharSequence show(Bool b) '''Boolean'''
  def dispatch CharSequence show(Identifier id) '''Long'''
  def dispatch CharSequence show(ModelType m) ''' m.base.name '''
  def dispatch CharSequence show(Args a) '''@RequestParam  a.type.show   a.name  IF a.next !== null ,  a.next.show  ENDIF '''
  def CharSequence showName(Args a) ''' a.name  IF a.next!==null ,  a.next.showName  ENDIF '''
  def CharSequence showType(Args a) ''' a.type.show '''

  def dispatch CharSequence showReq(Post post)'''Post'''
  def dispatch CharSequence showReq(Get get)'''Get'''
  def dispatch CharSequence showReq(Put put)'''Put'''
  def dispatch CharSequence showReq(Delete del)'''Delete'''
}
```