

Mod4j User Guide

**Jos Warmer
Eric Jan Malotaux
Johan Vogelzang**

Mod4j User Guide

by Jos Warmer, Eric Jan Malotaux, and Johan Vogelzang

1.3.0

Copyright © 2009 Ordina and other Mod4j committers

Table of Contents

1. Introduction	1
1.1. Introduction	1
1.2. References	1
2. Mod4j Principles and Patterns	2
2.1. Principles	2
2.1.1. Code Generation Principles	2
2.1.2. Modeling Principles	2
2.2. Code Generation Patterns	3
2.2.1. Generation Gap Pattern	3
3. The Mod4j Eclipse environment	5
3.1. Mod4j Project	5
3.1.1. Mod4j New Project Wizard	5
3.1.2. Mod4j New Model Wizard	5
3.1.3. Mod4j Model Editors	5
3.1.4. Mod4j FileTracker View	5
3.1.5. Mod4j CrossX View	5
4. Mod4j Code Generation	7
4.1. Structure of Generated Code	7
4.2. Spring configuration	8
4.3. Tuning the Code Generation Properties	8
5. CrossX	11
6. Business Domain DSL Reference	12
6.1. Domain Class	13
6.1.1. Definition	13
6.1.2. Generated code	14
6.2. Inheritance	15
6.2.1. Generated code	15
6.3. Attributes	16
6.3.1. Definition	16
6.3.2. Generated code	17
6.4. Enumerations	20
6.4.1. Generated code	20
6.5. Business Rules	20
6.5.1. Generated code	21
6.6. Associations	21
6.6.1. Generated code	21
7. Data Contract DSL Reference	24
7.1. Data Transfer Object (DTO)	24
7.2. Custom Dto	24
7.2.1. Attributes	24
7.2.2. Generated code	24
7.3. Class Dto	25
7.3.1. Generated code	26
7.4. Enumeration Dto	26
7.4.1. Generated code	26
8. Service DSL Reference	27
8.1. Service	27
8.1.1. Generated code	27
8.2. Service Method	27
8.2.1. Generated code	28
8.3. CRUD: Create, Read, Update, Delete service methods	28
8.3.1. Generated code	28
8.4. ListAll and Find Methods	29
8.4.1. Definition	29
8.4.2. Generated code	30
8.5. Reference methods	30
8.5.1. Definition	30

8.5.2. Generated code	30
9. Presentation DSL Reference	31
9.1. Dialogues	31
9.1.1. Definition	31
9.1.2. Generated code	31
9.2. ContentForm	31
9.2.1. Definition	31
9.2.2. Generated code	32
9.3. Processes	32
9.3.1. Generated code	32

List of Figures

2.1. Generation Gap Pattern applies to Java Classes	4
6.1. Business Domain DSL generation for each Domain Class	12
6.2. Business Domain DSL generation for a Business Domain model.	12
6.3. Business Domain DSL generation for a Business Domain model.	13
6.4. Generation Gap pattern applied to the Domain Layer	14
6.5. Generation Gap pattern applied to the Data Layer	15

Chapter 1. Introduction

1.1. Introduction

Mod4j (*Modeling for Java*) is an open source DSL-based environment for developing administrative enterprise applications. It uses a collection of DSL's to model different parts of the architecture, combined with manually written code. Currently Mod4j consists of four DSLs: the Business Domain DSL, Service DSL, Data Contract DSL and Presentation DSL. The modeling environment is seamlessly integrated into the Eclipse IDE which gives the developers one environment where they can easily switch back- and forth between models and code. The different DSL's used in Mod4j can be used independently, but if they are used in collaboration they will be fully validated with each other. Apart from integration in the Eclipse IDE, Mod4j also supports the use of Maven. That is, using the DSL models as the source, the complete code generation process can be run automatically on a build server without the need for Eclipse. This is a must for professional development and fits well in current ways of working.

This document is the user guide for Mod4j. Chapter 2, *Mod4j Principles and Patterns* describes the guiding principles that are used within Mod4j.

1.2. References

Bibliography

[mod4j-app-arch] *Mod4j Application Architecture*. Eric Jan Malotaux. 2009.

Chapter 2. Mod4j Principles and Patterns

2.1. Principles

It is important to understand the principles behind the Mod4j project. These principles have guided how the DSL's are designed and how they should be used.

2.1.1. Code Generation Principles

Code generation has been around for a long time. Mod4j takes in account many of the lessons learned. The following principles are guiding the Mod4j development.

- **Generated code must be clear and readable.** Using Mod4j should not be a lifetime commitment. Therefore the source code must be at least as good as handwritten code. This allows projects to continue development with the generated code instead of the DSL models. Note that this surely isn't the recommended way, but we like the freedom this gives.

Another reason for generating clear and readable code is that we do not have debuggers at the model level. This would be great, but the tools to develop model-level debuggers are not available (yet, we hope). Therefore code needs to be debugged at the source code level and having readable generated code really helps.

The third reason for generating clear code is that we do not generate everything. We support a mixed-mode development where developers write both models and code to extend the generated code. It is easier to extend generated code when this is readable.

- **Generated code is strictly separated from handwritten code**, in other words *the model is always leading* and code is always regenerated from the model whenever the model changes.

To ensure that generated code never needs to be overwritten the generated code is designed and generated with so-called *extension points*. These extension points are the only places where handwritten code may be placed. In most cases an extension point is in a file separate from the generated code. To help the developer a first empty extension point file is usually created. Extension points are regenerated as long as no handwritten code has been added. Once handwritten code has been added an extension point is not regenerated anymore.

- **Mod4j supports incremental code generation.** Code is generated on a per model file basis. Effectively whenever a model file is saved the code for this model file is generated automatically in the background. There is no need for a separate code generation step. This code generation step uses the standard Eclipse build structure. If the option 'build automatically' is set code generation will take place automatically. If a full build is requested all code will be regenerated. This allows Eclipse users to work with models in the same way as with source code (i.e. Java class files).
- **Automatic builds through Maven.** Mod4j includes a Maven plugin that support the code generation process in batch mode. This is used to run automatics builds based on the models. There is no need to checkin generated source code, just as there is no need to checkin compiled Java classes.

2.1.2. Modeling Principles

Modeling in Mod4j is guided by a set of principles:

- **The goal of a model is to generate code.** Thus, everything in a model is used in some way to generate code. We don't do modeling for the sake of modeling.

- **Model and code live side by side.** That is, we do not focus on 100 percent code generation. Instead we focus on modeling aspects from which as much code as possible can be generated. At the same time we keep using code for aspects that are as much work in code as they are in a model. Given the time constraints that we all have, we start with the modeling concepts that give the best ratio of model-to-code.
- **DSL's and DSL models are independent of each other.** They may reference through soft references, but are never hard-linked to each other. This allows both DSL models and DSLs to be developed separately. It also allows version and multi-user management by common tools as CVS or subversion.
- **DSL Models are and remain small.** We do not need to handle huge models because models are always divided into small, independent parts which we call DSL models. The relationship between those DSL models is done by symbolic integration of the models. See the section on Crossx.

2.2. Code Generation Patterns

The code generated by Mod4j is designed to be extensible by handwritten code. This provides the developer with maximum flexibility. These so-called extension points are designed in such a way that a developer never needs to overwrite generated code, he always extends it. This ensures that code can always be regenerated keeping the model as the source for development during the lifetime of an application.

This section describes code generation patterns that are used in the various code generators.

2.2.1. Generation Gap Pattern

The Generation Gap pattern is described at <http://www.research.ibm.com/designpatterns/pubs/gg.html> by John Vlissides. The main aspects of this pattern are:

2.2.1.1. Intent

The intent of the pattern is to be able to modify or extend generated code just once no matter how many times it is regenerated.

2.2.1.2. Motivation

Having a computer generate code for you is usually preferable to writing it yourself, provided the code it generates is * correct, * efficient enough, * functionally complete, and * maintainable. The problem is twofold:

- handwritten code may get lost when re-generating code from a model
- generated code may sometimes not do exactly what is needed

2.2.1.3. Solution

Separate the handwritten and generated code in separate classes / files. Generate an abstract base class that contains all code that is generated. Generate an empty subclass in a separate file. In this way the generator may always regenerate the base class with all generated code, while it will never regenerate the subclass file with handwritten code.

The handwritten subclass file is used for two purposes:

- Extend the generated code in the base class with things that cannot be generated from the model.

In this case the base class usually defines something like an abstract method such that this must be defined in the handwritten file

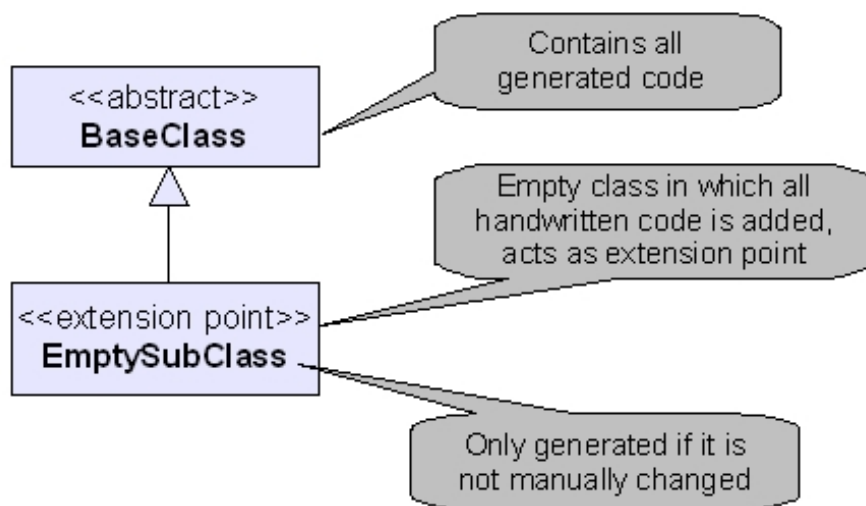
- Overwrite a method in the generated base class. This allows a developer to overrule the generated code. The system depends to a great extent on assumptions about the generated code. Therefore the developer should be intimately aware of the consequences and only use this in exceptional cases.

The generation gap pattern is used at several places within Mod4j. It is a well known pattern in Java code generation and allows for a clear separation of generated and handwritten code. Code generation for a Java class or java interface is done in two parts. An abstract base class is generated which contains all the generated code. An empty concrete subclass is generated, in which handwritten code should added.

The generator will always regenerate the abstract base class. The generator will only generate the empty subclass if it has not been manually changed. In this way handwritten code is never overwritten by the code generator.

The solution is shown in Figure 2.1, “Generation Gap Pattern applies to Java Classes”.

Figure 2.1. Generation Gap Pattern applies to Java Classes



There are a few properties of this solution that we need to be aware of:

1. The empty subclass isn't completely empty. It needs to include the public constructors from the base class with a calls to super(). This means that whenever the generated constructors change, the constructors in the subclass will be outdated.
2. When the abstract base class is not generated anymore, the empty subclass will become superfluous. It refers to a non-existing base class. The java compiler will flag this as an error, and the developer needs to remove the subclass by hand.

Chapter 3. The Mod4j Eclipse environment

3.1. Mod4j Project

Mod4j models reside within an Eclipse Mod4j project. A Mod4j project has several characteristics. First of all, such a project has a Mod4j nature, which tells Eclipse that this project contains Mod4j DSL models. This nature triggers the Eclipse build system to run on the model units in this project. Mod4j model files must reside in a Mod4j project

3.1.1. Mod4j New Project Wizard

To create a Mod4j project you can use the Mod4j Wizard. You can find this under the menu option **File ==> New ==> Project ... ==> Mod4j ==> Mod4j Project**. You can then define the project name and the project location. Within a Mod4j project the folder `src/model` is special. All Mod4j models must reside within this folder. The default structure below this folder is to make sub folders for business domain models, data contract models and service models.

3.1.2. Mod4j New Model Wizard

Once a Mod4j project has been created you can create a new Mod4j model through the menu option **New ==> Other... ==> Mod4j ==> Mod4j <DSL name> Model File**. Make sure you create the model files inside the `src/model` folder, or they will be ignored.

3.1.3. Mod4j Model Editors

Once a Mod4j model is created it can be opened in the special model editor by double clicking it. The model editors use syntax highlighting for keywords, where the color of the keywords matches the color of the layer in the architectural overview. Each editor has a different keyword color. This helps you to quickly know which type of model you are editing.

The model editors also have code completion, which is of course triggered by the default Eclipse key combination **Ctrl-SPACE**. Errors in the models are displayed in the Eclipse Problem View, thus working as expected in Eclipse.

3.1.4. Mod4j FileTracker View

For each model unit code is generated in a number of different files that may reside inside different folders in projects. It is rather cumbersome to navigate through the different projects to find the files that are generated. The FileTracker view, which can be shown by the menu option **Window ==> Show View ==> Other ==> Mod4j ==> Mod4j FileTracker**. The FileTracker view shows a tree with all model files. Under each model file the list of files generated from the specific model is shown. A generated file may be shown with a small G in the right lower corner, which indicates that this is a generate3d file that will always be overwritten. Files without the G are extension points and may be changed by hand. As long as these files are not changed by hand they will be regenerated. When such a file is changed, manually the Mod4j builder will recognize this and will not overwrite such a file.

Double clicking on the model file or on a generated file will directly open the file, provided that the project in which the file resides is available as an Eclipse project in your workspace. For new projects you first need to import the different projects as described in Chapter 4, *Mod4j Code Generation*.

3.1.5. Mod4j CrossX View

As described in Chapter 5, *CrossX*, the connection between the different models in different files is supported by the CrossX component. To view the contents of the CrossX broker you can select **Window ==> Show View ==> Other ==> Mod4j ==> CrossX**. For normal usage of Mod4j, the CrossX View is not needed. It becomes useful when you are developing your own DSL and need to check whether the right symbols are exported to CrossX from your DSL.

Double clicking on the model file will directly open the file, provided that the project in which the file resides is available as an Eclipse project in your workspace. For new projects you need to import the different projects as described in Chapter 4, *Mod4j Code Generation*.

Chapter 4. Mod4j Code Generation

4.1. Structure of Generated Code

Code generation in Mod4j follows a well-defined structure, which is derived from the layers in the application architecture.

From the various models in Mod4j code is generated to so-called *modules*. Each module represents a layer of the architecture. The names of the modules are derived from the application name. For example, if the application name is *RecordShop* the modules used in Mod4j are:

- *RecordShop-dslModels*: this is the project that contains all of the Mod4j models., No code is generated into this project.
- *RecordShop-domain*: this project contains code for the domain layer of the architecture. Code is generated from the business domain DSL.
- *RecordShop-data*: this project contains code for the data layer of the architecture. Code is generated from the business domain DSL.
- *RecordShop-business*: this project contains code for the business layer of the architecture. Code is generated from the service DSL.
- *RecordShop-service*: this project contains code for the service layer of the architecture. Code is generated from the data contract and service DSLs.
- *RecordShop-presentation*: this project contains code for the presentation layer of the architecture. Code is generated from the presentation DSL. Note that the presentation DSL is not part of the 1.0.0 release of Mod4j.

Each modules is a Maven module and includes a Maven *pom.xml* file that describes the build structure of the module. The location of the modules is defined in the Mod4j.properties files which is described in Section 4.3, “Tuning the Code Generation Properties”.

The structure of the modules in the file system is important, because all of the modules need to use the same parent pom.xml file. The structure is as follows:

```
main-folder
  pom.xml                    -- the parent pom for all modules
  RecordShop-dslModels
    pom.xml
  RecordShop-domain
    pom.xml
  RecordShop-data
    pom.xml
  RecordShop-business
    pom.xml
  RecordShop-service
    pom.xml
  RecordShop-presentation
    pom.xml
```

The project can be built by running maven (mvn clean install) in the main-folder.

To work with the generated modules in Eclipse Mod4j uses the Eclipse plugin Q4E. The q4e plugin allows an Eclipse user to import maven projects as Eclipse projects. It takes care of keeping all Eclipse settings in sync with the pom.xml file. The *getting started* guide on the Mod4j project site describes how to install q4e and how to run maven on a newly created project.

Warning

Within Eclipse the default location of projects is directly in the workspace folder. The consequence of this is that the parent pom for a Mod4j project will reside in the workspace folder. This becomes a problem when you are working with more than one Mod4j project. The parent pom for each of the Mod4j projects will have the same location. To avoid these problems, we advise to create Mod4j projects outside the Eclipse workspace.

4.2. Spring configuration

As explained in the [mod4j-app-arch], each maven module corresponding with an architectural layer, except the domain layer, has its own Spring configuration. The contexts are created and tied together by a `ContextSingletonBeanFactoryLocator`, configured by a set of `beanRefContext.xml` files. The configuration is separated into parts that are always generated, and parts that are generated just once as a convenience for the developer, and after that never overwritten because they are intended to be maintained manually. The generated parts are generated into a file named `applicationContext.xml` in the directory corresponding to the root package for each module in the source folder `generated-resources`. The manually maintained files are generated just once into the directory corresponding to the root package for each module in the source folder `src/main/resources`. The manually maintained configuration files contain those configuration items that depend on the environment where the application will be deployed and therefore cannot be known in advance. Currently there are three such configuration files:

1. `dataSourceContext.xml` in the data module.
2. `sessionFactoryContext.xml` in the data module.
3. `transactionManagerContext.xml` in the service module.

It is quite probable that in the future more specialised environment dependent configuration files will be introduced, for instance for JMS queues et cetera.

This separation of Spring configuration files between modules, and between generated and manual parts tries to benefit from knowledge available in the model on the one hand, and flexibility needed to adapt to the environment on the other hand. The generated setup will be sufficient for simple cases. When a more complicated configuration is needed, the developer will have to create his own, but he can still use most of the generated parts. In that case he will probably substitute his own `beanRefContext.xml` for the generated ones and assemble the contexts there. Just one `beanRefContext.xml` will probably be easier to maintain by hand than the several generated ones.

4.3. Tuning the Code Generation Properties

Inside the `src/model` folder of the Mod4j project a file named `mod4j.properties` is generated by the new Mod4j project wizard. This file contains a number of properties that can be used to tune the Mod4j code generation. This file has a number of sections which are explained below.

Application properties

Default encoding for generated XML files

```
fileEncoding=UTF-8
```

The name of the application

```
applicationName=RecordShop
```

The version of the application, this is used by Maven

```
applicationVersion=1.0-SNAPSHOT
```

The main location of the project, this has only been tested with the value ".", be aware of any other value you use.

```
applicationPath=.
```

The name of the root package for the generated code.

```
rootPackage=org.company.recordshop
```

The folder where Java sources are (re)generated. Files generated in this folder are always regenerated, thus they should never be changed by hand. Also, there is no need to checkin these generated files in SVN or other version control system. The files are regenerated anyway by both the Eclipse and the Maven builder.

```
srcGenPath=generated-sources
```

The folder where resources are (re)generated. This has the same function as the *srcGenPath*, but for non Java files.

```
resourceGenPath=generated-resources
```

The folder where Java extension points are generated. These are the Java files where code may be added by hand. Any file changed after generation will not be regenerated. From this folder all manually changed files need to be checked in, unchanged files should not be checked in.

```
srcManPath=src/main/java
```

The folder where non-Java extension points are generated. Works identically to *srcManPath*.

```
resourceManPath=src/main/resources
```

Dsl Models properties

The name of the module that contains the Mod4j models.

```
dslModelsModuleName=RecordShop-dslModels
```

Domain module properties

The name of the module that contains the code for the domain layer.

```
domainModuleName=RecordShop-domain
```

The name of the root package for this module

```
domainRootPackage=org.company.recordshop.domain
```

Data module properties

The name of the module that contains the code for the data layer.

```
dataModuleName=RecordShop-data
```

The name of the root package for this module

```
dataRootPackage=org.company.recordshop.data
```

Indicated whether a new database schema should be created. The value create that a new database schema is always created. The value update means that only changes in the database schema are generated.

```
hibernate.hbm2ddl.auto=update
```

The Hibernate mapping strategy for inheritance. There are two possible values.

```
#hibernate-mapping.inheritance.strategy=[table.per.concrete.class|table.p  
hibernate-mapping.inheritance.strategy=table.per.concrete.class
```

The name of the class that generates the id for objects. Value *native* indicated that the id generator of the underlying database is used.

```
hibernate-mapping.class.id.generator.class=native
```

Business module properties

The name of the module that contains the code for the business layer.

```
businessModuleName=RecordShop-business
```

The name of the root package for this module.

```
businessRootPackage=org.company.recordshop.business
```

Service module properties

The name of the module that contains the code for the service layer.

```
serviceName=RecordShop-service
```

The name of the root package for this module.

```
serviceRootPackage=org.company.recordshop.service
```

Presentation module properties

The name of the module that contains the code for the presentation layer

```
presentationModuleName=RecordShop-presentation
```

The name of the root package for this module.

```
presentationRootPackage=org.company.recordshop.presentation
```

Chapter 5. CrossX

Each model file is edited and processed separately. However information from one model file is often needed in another model file. To support the inter model referencing and validation we developed a specific component called CrossX. CrossX acts as a symbol table for model information. Each model exports symbolic information about a subset of its model elements to CrossX. Whenever a reference to a model element in another model is needed this is looked up in CrossX.

CrossX is integrated into the Mod4j Eclipse builder. When the CrossX builder is called on a model file it will extract the CrossX information from the model file. This information is stored in memory in the CrossX broker. It keeps track of which resource information has come from. If a model file changes, the information from that model files is replaced by the newly exported information.

The CrossX information is also stored in a .crossx file. These files are used to initialize the CrossX broker at Eclipse startup, to ensure that not all models need to be processed again.

The CrossX View can be used to examine the contents of the CrossX broker inside Eclipse. If it shows empty you need to rebuild your project (see issues below) CrossX will only process projects that have a Mod4j nature.

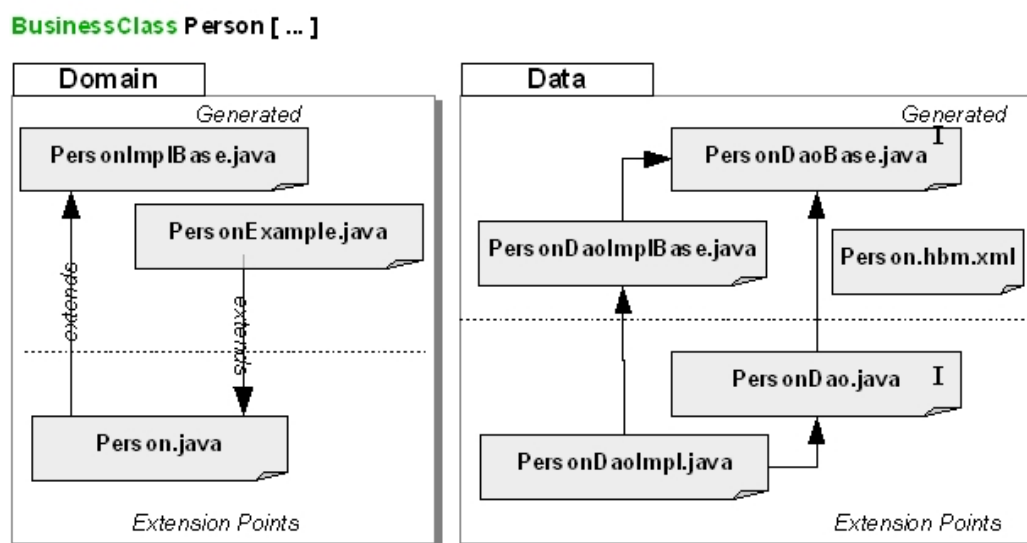
Users of Mod4j will usually not need to do anything with CrossX, it quietly does its work in the background. For developers of CrossX it is important to understand the inner working, which will be described in the developer guide.

Chapter 6. Business Domain DSL Reference

This chapter provides a description of all the concepts in the Business Domain DSL. Each concept is described by a definition which explains the business meaning of the concept, and a description of the code generated for the concept.

The Business Domain DSL is intended to be used for modeling the domain model of an application. The Business Domain model consists of all domain classes, as understood by the business expert. Technical aspects do not belong in Business Domain models.

Figure 6.1. Business Domain DSL generation for each Domain Class

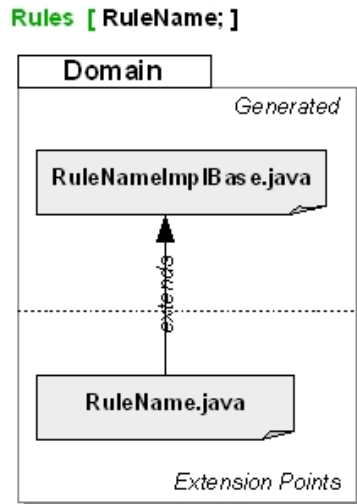


As you can see in the figure, eight files are generated for each domain class. At the domain level the generation gap pattern is used to generate two files. At the data layer a DAO class always has a separate interface. To allow manual additions, a double generation gap pattern is used where both the interface and the class can be extended. Next to the Java files a Hibernate mapping file is generated containing the mapping for the specified class.

To support the Hibernate query by example functionality the PersonExample class is generated. The Person.java class has its validation turned on, even when the constructor is called. This supports the rule in the application architecture that an object must always be valid. For the query by example functionality the example object does not need to be a valid domain object. The purpose of the PersonExample.java class is to turn off the validation in the constructor. This example class is only to be used for the query by example functionality.

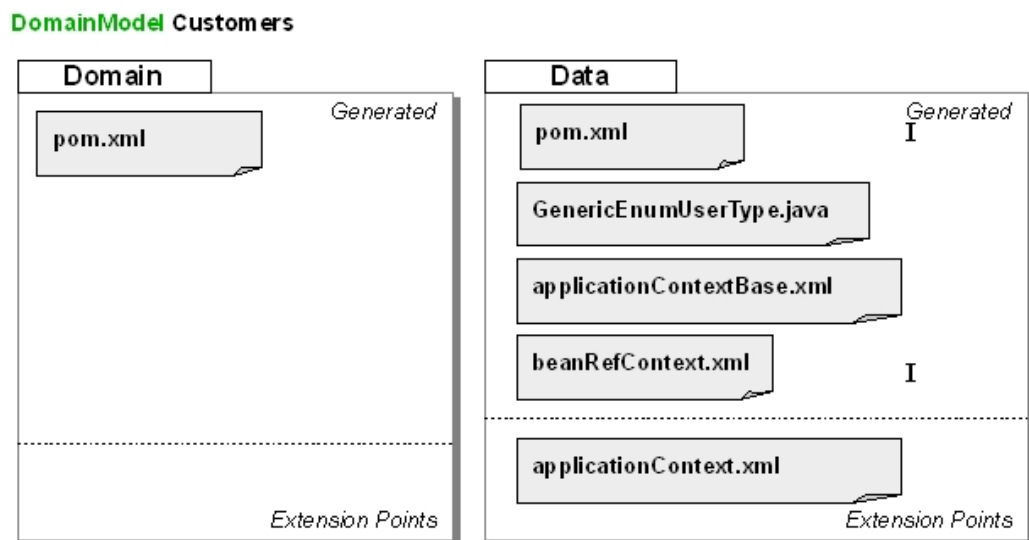
For a business rule two files, following the generation gap pattern are generated. This is shown in figure Figure 6.2, “Business Domain DSL generation for a Business Domain model.”

Figure 6.2. Business Domain DSL generation for a Business Domain model.



For the model as a whole, several files are generated as well, independent of the number of business classes in the model. This is shown in the following figure.

Figure 6.3. Business Domain DSL generation for a Business Domain model.



The remainder of this chapter describes the concepts as they are defined in the language and how they can be used to specify a business domain model.

6.1. Domain Class

6.1.1. Definition

```
class Person [
    // properties go here ...
    // rules go here ...
]
```

A Domain Class describes something that has a specific meaning in the business domain being modeled. In the system being modeled instances of domain classes will be created.

A domain class has properties (also called attributes in e.g. UML) of different types.

A domain class also contains business rules, that specify the conditions that an instance of the class needs to conform to.

6.1.2. Generated code

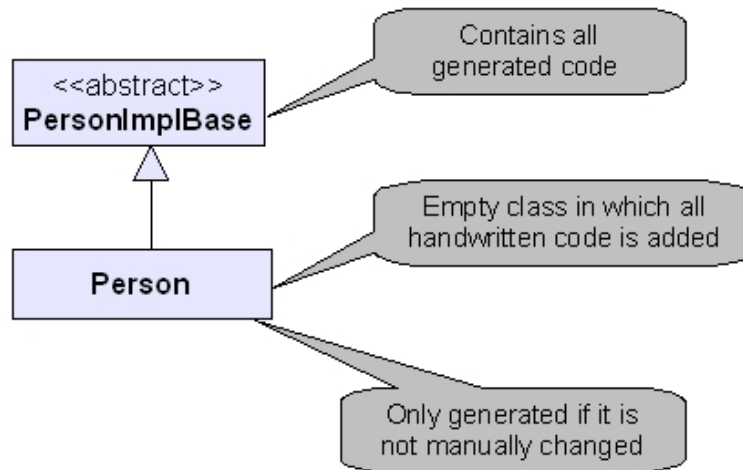
For a domain class code is generated for both the domain and the data layer of the architecture.

6.1.2.1. Domain Layer

In the domain layer code is generated according to the *generation gap pattern* (Section 2.2.1, “Generation Gap Pattern”). For each domain class two Java classes are generated. The first Java class is an abstract base class containing all generated code for the domain class. All code generated for properties, associations and validation of attribute constraints and business rules is placed inside this class.

The second class generated is an empty (except for constructors calling `super()`) class which subclasses the first class. This empty class is what we call an *extension point* file, where manual code may be added. This extension class is generated once and not overwritten, which ensures that all handwritten code is maintained at all times.

Figure 6.4. Generation Gap pattern applied to the Domain Layer



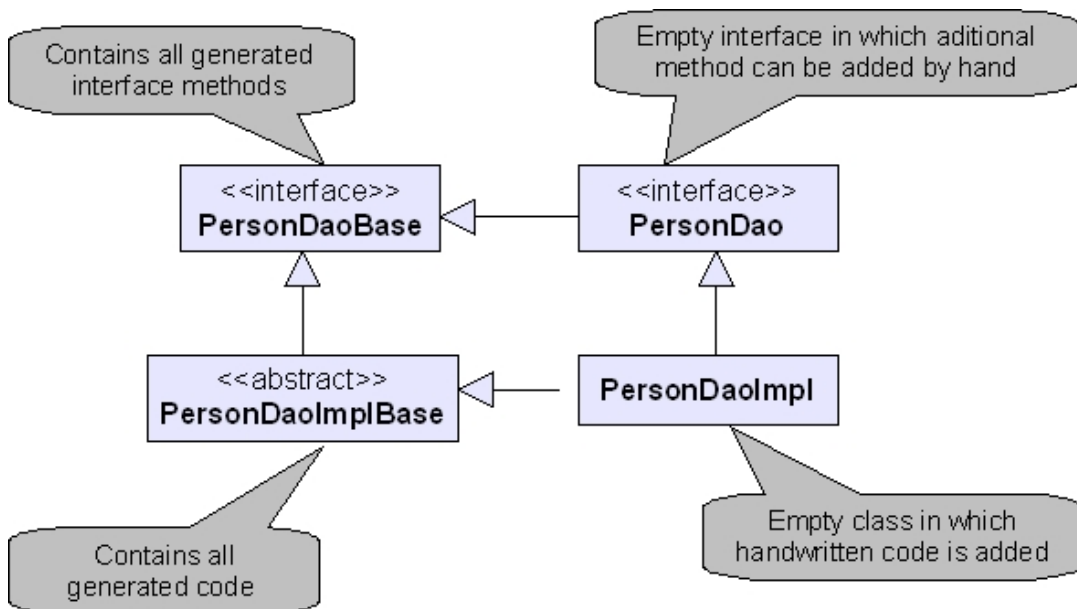
6.1.2.2. Data Layer

In the data layer a number of files is generated.

- A hibernate mapping file is generated for each domain class.

- A DAO (Data Access Object) is generated. The DAO normally consists of one interface and an implementing class. To get a clean separation of generated and handwritten code both of these use the generation gap pattern. The result for a class called Person is shown in figure Figure 6.5, “Generation Gap pattern applied to the Data Layer”. Note that this is another instance of the generation gap pattern as described in Section 2.2.1, “Generation Gap Pattern”.
- Spring configuration files are generated (beanRefContext.xml and applicationContext.xml) for the DAOs mentioned in the previous bullet. These files are generated in the generated-resources directory. Two other Spring configuration files are generated just once in src/main/resource and never overwritten: dataSourceContext.xml, containing a JNDI data source definition suitable for deployment in an application server, and sessionFactoryContext.xml, containing a definition of a Hibernate session factory bean. These files are generated just once to help developers get up and running quickly. They will never be overwritten, and are intended to be maintained manually by the developer. The reason for this is that data sources and session factories are generally dependent on the environment where the application will be deployed, and so cannot be known beforehand. The generated beanRefContext.xml contains references to these files.

Figure 6.5. Generation Gap pattern applied to the Data Layer



6.2. Inheritance

```

class Customer inherits Person [
    // additional attributes go here ...
    // additional rules go here ...
]
  
```

A domain Class may inherit from another domain class. All attributes, associations and rules defined for the class from which it inherits are applicable to the subclass as well.

A domain class may inherit from one class at most, multiple inheritance is not supported.

6.2.1. Generated code

For inheritance domain class code is generated for both the domain and the data layer of the archi-

tecture.

6.2.1.1. Domain Layer

In the domain code is generated to ensure that inheritance is part of the Java class structure as well. That is, inheritance is mapped onto the Java *extends* .

6.2.1.2. Data Layer

In the data layer code is generated in the hibernate mapping files using the hibernate support for inheritance.

In the DAO package, a regular DAO Java file is generated. There is no inheritance relationship between the class and its superclass in the DAO code. The DAO files contains methods for accessing the database as shown below.

```
public interface CustomerDaoBase {
    Customer retrieve(Long id);
    Customer add(Customer object);
    Customer update(Customer object);
    void delete(Customer object);
    List<Customer> listAll();
    List<Customer> listPage(final int firstResult, final int maxResults);
    long count();
    List<Customer> findByExample(CustomerExample example);
}
```

The generated code includes full Javadoc, which has been left out in the above example to save space.

6.3. Attributes

6.3.1. Definition

Attributes are part of a domain class and defined within the scope of their class. The example class person looks as follows.

```
class Person [
    string name
        derived [ writable ]
        default "the default name"
        minlength 3 maxlength 10
        regexp "^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}$"
        nullable ;
    integer numberOfEars; // optional constraints
    datetime birthdate; // optional constraints
    decimal length; // optional constraints
    "Description of the isMale attribute"
    boolean isMale; // optional constraints
    // rules go here ...
]
```

An attribute describes a characteristic of a domain class. Attributes always has a defined type, which can be *integer*, *string*, *decimal*, *boolean*, *datetime* or an enumeration defined in the domain model.

Each attribute can have a description, which is placed directly before the attribute definition within double quotes. In the example only *isMale* has a description.

For each attribute additional constraints can be specified. In th above example *minlength*, *maxlength* and *regexp* are such constraints. For each attribute type the possible constraints are defined in the following sections.

Each instance of a domain class has its own values for each attribute. These values always need to conform to all the rules specified for the domain class.

An attribute can be defined as *derived*, which means that its value is not set directly, but derived from the values of other attributes. The derivation algorithm must be provided by manually implementing the corresponding *getter* method in the extension point Java class. Normally the derivation algorithm only works one way, so that the derived property is read-only, which means that there is no setter for it. A subset of all possible derivation algorithms work both ways, so that the values of the attributes can be derived from the value of the derived attribute. This can be specified by adding the keyword *writable*. For derived attributes, the reverse derivation algorithm must be provided by manually implementing the corresponding *setter* method in the extension point for the enclosing class.

6.3.2. Generated code

For an attribute code is generated for both the domain and the data layer of the architecture.

As described before code is generated in the domain layer according to the generation gap pattern. For each domain class an abstract base class is generated. This abstract bases class contains a field for each attribute and getter and setter methods to access the attribute.

The description of an attribute is used to generate Javadoc comments in the code for the attribute.

If the there are constraints specified for the attribute in the model, validators for the attribute are also generated. Code is also generated to automatically call these validators whenever the value of the attribute has changed, e.g. when calling the setter method. Whenever an attempts is made to change the value of a attribute that will break a constraint, a *BusinessRulevalidation* exception is thrown and the value of the attribute is not changed.

If an attribute is not nullable and has no default value (see next sections for an explanation of nullable and default value), the attribute is added as a parameter to the constructor of the class. This ensures that an object of this class will always be correct.

For *derived* attributes the getter in the generated abstract base class is generated *abstract*, which forces the developer to provide a concrete implementation in the extension point. A abstract getter method is only generated when the derived attribute is also declare *writable*, which forces the developer to also provide a concrete implementation in the extension point.

6.3.2.1. String Attribute

A string attribute is completely denoted as shown below. Note that all constraints (after *string* name) are optional.

```
"A readable description of this attribute
string name
    default "the default name"
    minlength 3
    maxlength 10
    regexp "^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}$"
    nullable ;
```

Each string attribute has a number of optional properties,. The order of the properties is fixed as shown above. These properties act as constraints on the value of the attribute and are explained below..

default	Defines the default value of this attribute. This value is assigned when the object is being created. The value may be changed later on in the objects lifetime.
---------	--

minlength	Defines the minimal number of characters of this attributes
-----------	---

	value.
maxlength	Defined the maximum number of characters of this attributes value. The value of <i>maxlength</i> must always be higher or equal to value of <i>minlength</i> ..
regexp	Defines the format of the attributes value through a regular expression. The syntax of the regular expression is defined by the Java regular expressions.
nullable	The nullable property specifies that this attribute does not have to have a value. If nullable is not specified, the attribute must always have a defined value..

6.3.2.1.1. Generated code

A String attribute is implemented by a Java *String* field. Apart from the generic code generated for each property no additional code is generated for a string attribute.

6.3.2.2. Boolean Attribute

A boolean attribute is completely denoted as shown below

```
"A readable description of this attribute"  
boolean fixed  
    default "true"  
    nullable ;
```

Each attribute can have a description, which is places directly before the attribute definition. Each boolean attribute has a number of optional properties, as shown above. The order of the properties is fixed. These properties act as constraints on the value of the attribute and are explained below..

default	Defines the default value of this attribute. This value is assigned when the object is being created. The value may change later on in the Objects lifetime.
nullable	The nullable property specifies that this attribute does not have to have a value. If nullable is no specified, the attribute must always have a defined value.

6.3.2.2.1. Generated code

A boolean attribute is implemented by a java Boolean field. If the boolean attribute is not *nullable* the getter and setters use a Java *boolean*, if the attribute is nullable the getter and setters use a *Boolean*.

6.3.2.3. Integer Attribute

An integer attribute is completely defined as shown below

```
"A readable description of this attribute"  
integer numberOfSteps  
    default 12  
    min 4  
    max 32  
    nullable ;
```

An integer attribute has a number of optional properties, as shown above. The order of the properties is fixed. These properties act as constraints on the value of the attribute and are explained below..

default	Defines the default value of this attribute. This value is assigned when the object is being created. The value may change later on in the Objects lifetime. If the <i>min</i> and/or <i>max</i> properties are specified the default value must conform to these properties.
min	Defines the minimal value of this attribute.
max	Defines the maximum value of this attribute. The value of <i>max</i> must always be greater or equal to the value of <i>min</i> .
nullable	The nullable property specifies that this attribute does not have to have a value. If nullable is no specified, the attribute must always have a defined value.

6.3.2.3.1. Generated code

An integer attribute is implemented by a java *Integer* field. If the attribute is not *nullable* the getter and setters use a Java *int*, if the attribute is nullable the getter and setters use an *Integer*.

6.3.2.4. Decimal Attribute

An decimal attribute is completely defined as shown below

```
"A readable description of this attribute"  
decimal percentage  
    default 12  
    min 4  
    max 32  
    nullable ;
```

A decimal attribute has a number of optional properties, as shown above. The order of the properties is fixed. These properties act as constraints on the value of the attribute and are explained below..

default	Defines the default value of this attribute. This value is assigned when the object is being created. The value may change later on during the objects lifetime.
min	Defines the minimal value of this attribute.
max	Defines the maximum value of this attribute. The value of <i>max</i> must always be greater or equal to the value of <i>min</i> .
nullable	The nullable property specifies that this attribute does not have to have a value. If nullable is no specified, the attribute must always have a defined value.

6.3.2.4.1. Generated code

A decimal attribute is implemented by a java *Float* field. If the attribute is not *nullable* the getter and setters use a Java *float*, if the attribute is nullable the getter and setters use a *Float*.

6.3.2.5. Datetime Attribute

A datetime attribute is completely defined as shown below

```
"The day on which it all started"  
datetime day started  
    default 4  
    nullable ;
```

A datetime attribute has a number of optional properties, as shown above. The order of the properties is fixed. These properties act as constraints on the value of the attribute and are explained below..

accuracy	Defines the accuracy of the datetime attribute. The possible values are year, month, day, hour, minute, second, millisecond. In the above example the accuracy is defined as day.
default	Defines the default value of this attribute.
nullable	The nullable property specifies that this attribute does not have to have a value. If nullable is no specified, the attribute must always have a defined value.

6.3.2.5.1. Generated code

A datetime attribute is implemented by a java *org.joda.time.DateTime* field.

6.4. Enumerations

An enumeration defines a data type which can be used as the type of an attribute of a domain class. The enumeration definition includes a list of all enumeration values. The example below shown an enumeration definition.

```
"The severity level of a customer problem"
enumeration SeverityLevel [
    Low    = 1;
    Middle = 2;
    High   = 3;
]
```

The "= 1" defines the numeric value that is used for the enumeration literal in the code and the database. This is mandatory, because it is often necessary to retain previous values when you need to address existing encoding schemes for enumerations, or when you are adding enumeration literals during development.

6.4.1. Generated code

For an enumeration a Java *enum* type is generated.

6.5. Business Rules

A domain class may include business rules. Note that the concept of business rule here is limited. In this context a business rule is a rule that should be true at all times for every object of the given domain class.

Other rules, for example one that trigger something when a certain condition is met, are not covered.

```
class SampleClass [
    // attributes go here ...
    // rules go here ...
    rules [
        "A first rule"
        mustBeValid;
        "A second rule"
        isOriginal;
        "a unique rule"
        unique namesUnique [ firstname, lastname ]
    ]
]
```

There are two types of business rules that can be modeled.

The first type of business rule has a name only in the model. The actual implementation of the rule must be written in Java as an extension to the generated code. This type business rules are validated in the order they were declared. So in the Sample class above, first the "mustBeValid" rule will be validated and after that the "isOriginal" rule.

The second type of rule is a uniqueness rule, which is indicated by the *unique* keyword. A uniqueness rule specifies that the combination of values of one or more attributes of the class must be unique at all times. The names of the attributes are specified between brackets after the name of the rule.

6.5.1. Generated code

For a business rule an abstract method with the name of the rule returning a boolean value is generated in the base class. This ensures that the developer must provide an implementation in the empty subclass.

The first time the empty subclass is generated an empty method implementing the abstract method from the base class is generated. This method returns true and contains a *TODO comment* to remind the developer he needs to implement the rule.

The handling of the rules is completely generated. This means that the rule will be called and validated whenever the object changes. If the rule is broken, a `BusinessClassException` is created and thrown. Developers do not need to handle these errors themselves.

See the generation gap pattern on how this is done in the code

6.6. Associations

The final part of a domain model defines all the relationships between the domain classes.

```
association Customer    customer    one    <-> many Order      orders      ;  
association Order      order      one    -> many OrderLine  orderLines ordered ;  
association OrderLine  orderline  many   -> one Record    record      ;  
association Record    records   many   <-> many Artist    contributors ;
```

This example shows one bidirectional association, which uses the <-> syntax and one unidirectional association using the -> syntax.

The qualifiers **one** and **many** denote the multiplicity of the association at the side of the object closest to it.

The **ordered** keyword states that the association is ordered (not sorted!). The **ordered** keyword can only occur at the target (right hand side) of an association. This means that each element has an index in the relation to the source (left hand side) element. Ordered associations are currently restricted to unidirectional associations only.

6.6.1. Generated code

Associations are a powerful mechanism and require careful handling in the code. For bidirectional associations the generated code ensures that both sides will always be updated consistently whenever one of the sides is changed.

6.6.1.1. Generated Code for the Domain Layer

On the domain layer the code is generated in the Java base class to store the references to objects at the opposite side of an association. For associations with multiplicity 0 or 1 this will be a field of the type at the opposite side, for associations with multiplicity greater than 1 this will be a field of type

Set<opposite type>.

Next to these fields methods are generated to change these fields. For a single valued field a simple setter and getter is generated.

For multivalued fields adding or deleting elements from the collection should always be done by the object owning the collection. We cannot simply generate getter and setter methods, because changes in the collection can be done without the object knowing it, and the object cannot guarantee consistency with the opposite objects.

To resolve this problem we generate methods *addTo...*, *removeFrom...* for the collection in the owning class. For the Customer class in the above example we generate the *addToOrders()* and *removeFromOrders()* methods. The getter methods returns an Unmodifiable collection to guard the client from changing the collection.

For an ordered association a List is used instead of a Set, and additional *addTo...* and *removeFrom...* methods are generated to allows additions and removal based on the index.

```
public abstract class CustomerImplBase {
    /**
     * orders: One 2 Many Bidirectional: The customer has a number of orders
     */
    private Set<Order> orders = new HashSet<Order>();

    /**
     * @return orders (Set<Order>)
     */
    public Set<Order> getOrders()

    /**
     * Implements adding single element to a collection
     */
    public void addToOrders(Order element)

    /**
     * Implements removal of a single element from feature
     *
     * @param element
     */
    public void removeFromOrders(Order element)
```

To support the consistency of the two sides of an association we always have to update the other side whenever something changes. We cannot simply call the other sides update methods, because they will call their opposite side in turn and we end up in an infinite loop.

To solve this problem we use special that are only used to update the opposite side. All the methods are generated with 'z_internal' as prefix. These methods are public because they must be callable from the opposite class. These methods should never be used for anything else.

```
public abstract class CustomerImplBase {

    /**
     * This operation should NOT be used by clients. It implements the correct
     * addition of an element in an association.
     *
     * @param element
     */
    public void z_internalAddToorders(Order element) {

    /**
     * This operation should NOT be used by clients. It implements the correct
     * removal of an element in an association.
     *
     * @param element
     */
```

```
public void z_internalRemoveFromorders(Order element)
```

Chapter 7. Data Contract DSL Reference

This chapter provides a description of all the concepts in the Data Contract DSL. Each concept is described by a definition which explains the meaning of the concept, and a description of the code generated for the concept.

7.1. Data Transfer Object (DTO)

The central concept in the Data Contract DSL is the Data Transfer Object, or DTO. A DTO is a value object that carries data only. The purpose of a DTO is to be able to send data to back and forth to and from clients.

Clients communicate with domain objects through services. The services and the service methods are defined by using the Service DSL. The DTO objects define the in- and out parameters for the service methods.

There are several types of DTOs. A `customDto` is a DTO that may have any attributes. A `BusinessClassDto` represents the data of a domain object as defined in the Business Domain DSL. An `EnumerationDto` defines an Enumeration. All these variants are described in detail below.

7.2. Custom Dto

```
custom ExampleDto [  
    string description;  
    integer amount;  
    datetime deliverydate;  
    decimal price;  
    "Description of the isOk attribute"  
    boolean isOk;  
]
```

A custom dto has attributes of different types, corresponding to the types of attributes in the business domain DSL.

7.2.1. Attributes

DTO Attributes are part of a DTO and defined within the scope of their DTO. The example custom DTO `personDto` looks as follows.

An attribute describes a field of a data transfer object. Attributes always have a defined type, which can be *integer*, *string*, *decimal*, *boolean* or *datetime*.

Each attribute may have a description, which is placed directly before the attribute definition within double quotes. In the example only *isOk* has a description.

Contrary to domain classes in the Business Domain DSL, attributes of a DTO have no constraints associated with it.

7.2.2. Generated code

For a custom dto a Java class is generated which contains the attributes as protected fields with getters and setters.

The fields in data transfer objects are always nullable, therefore the field types in the generated code are always real objects. This means that *Integer*, *Float*, *Boolean* are used as field types for the attrib-

utes of type *integer*, *decimal* and *boolean*.

The description of an attribute is used to generate Javadoc comments in the code for the attribute.

7.3. Class Dto

A class DTO is a DTO that represents the data of a domain class from the Business Domain DSL. As a consequence, the attributes that are allowed within a class dto must map 1-to-1 to attributes of the corresponding domain class.

```
from RecordShopDomainModel import Customer ;  
  
class SimpleCustomerDto represents Customer [  
    // attributes go here  
]
```

A class dto always declares the domain class that it represents as its base class. This domain class must be defined in a Business Domain model. To ensure that the domain class is in scope it must be imported first from its business domain model. Import statements are always at the top of a data contract model.

```
class SimpleCustomerDto represents Customer [  
    name;  
    numberOfEars;  
    birthdate;  
    length;  
    "Description of the isMale attribute"  
    isMale;  
]
```

Attributes of a class dto must have the same names as attributes in the represented domain class. Not every attribute has to be there, allowing a class dto to contain a subset of attributes from the domain class it represents.

The type of the attributes is derived from the type of the corresponding attribute in the domain class in the business domain model. Therefore it is not specified in the data contract model.

If you need a BusinessClassDto that represents all attributes from a BusinessClass, the following shorthand will achieve this. It will by default add all attributes to the class dto.

```
class FullCustomerDto represents Customer ;
```

Using this shorthand only the attributes of the represented domain class are added, not the associations.

Apart from the attributes, a business class dto may also include references to other DTOs as shown below.

```
class SimpleCustomerDto represents Customer [  
    // attributes go here  
  
    references [  
        orders as OrderNumberAndDateDto ;  
    ]  
]
```

A domain class may take part in associations as defined in the business domain model. A class dto may represent such an association as well. There are restrictions on represented associations. The main restriction is that only one side of an association is represented as a reference. A Dto does not support bi-directional associations.

It is possible and supported to simulate bi-directional associations by defining a reference to another class dto that in turn references back to the first class dto, but this is discouraged. Because the generated dto's do not "know" that such references are each others inverse, it is possible to create inconsistent dto's. For instance, a dto instance may refer to another dto that refers to a different dto instance than the first one. A single reference is sufficient to update a corresponding bi-directional association in the domain model. The direction of the reference is not important; either one will work.

Generally it is better to avoid very complicated dto graphs, especially structures containing cycles or different path to the same referenced dto. Mod4J does support those and the generated code processes them correctly. But if the dto instance graph is inconsistent, the results may be unexpected.

The references are contained in the references block. The same rule that applied to attributes also applies to associations. Only associations that are defined in the business domain model may be used. The name to be used is the role name of the opposite class in the association definition.

The type of a reference must conform to the type defined in the association in the business domain model. The type of the reference must be a class dto that represents the domain class in the association. Multiplicity of the reference is derived from the definition of the association in the business domain model and therefore it does not need to be specified.

7.3.1. Generated code

For a class dto several files are generated.

First of all there is a straightforward Java class representing the attributes and references defined in the class dto .

Secondly a Translator class is generated. This class contains two translator methods to translate the class dto to a domain class and vice versa. For example for the class dto *FullCustomerDto* the following two translator methods are generated.

```
public synchronized Customer fromDto(final FullCustomerDto source,  
                                     Customer target) throws TranslatorException  
  
public synchronized FullCustomerDto toDto(final Customer source);
```

In the *fromDto()* method the *FullCustomerDto* object must represent a valid domain object, otherwise a *TranslatorException* is thrown.

7.4. Enumeration Dto

An enumeration defines a data type which can be used as the type of an attribute of a business class. The EnumerationDto definition includes a list of all enumeration values. As with the BusinessClassDto, this dto is connected to an Enumeration in the business class model. The example below shows an enumeration definition.

```
enumeration SeverityLevelDto represents SecurityLevelEnum [
```

The SecurityLevelEnum must be defined in a Business Class model and imported defines the numeric value that is used for the enumeration literal in the code and the database. This is optional, but useful when you need to address existing encoding schemes for enumerations.

7.4.1. Generated code

For an enumeration dto a Java *enum* class is generated. Also, a translator class is generated with method to translate the enumeration dto to its domain enumeration and vice versa.

Chapter 8. Service DSL Reference

This chapter provides a description of all the concepts in the Service DSL. Each concept is described by a definition which explains the meaning of the concept, and a description of the code generated for the concept.

The central concept in the Service DSL is the Service Method. A service method uses DTOs, defined in a Data Contract Model (see chapter on Data Contract DSL).

8.1. Service

A service model defines a service, which may include service methods. There are multiple types of service methods which are described in the following sections.

```
service Orders ;  
  
// methods go here
```

8.1.1. Generated code

For a service two Java classes are generated. One service class in the service layer, which contains all the service methods defined in the service model. In the business layer a corresponding class is generated, which includes the same method.

The class at the service layer uses DTO types as its parameters, thus making its interface completely independent of the domain layer. The class at the business layer uses domain types. Typically the implementation of a service method in the service layer translates its arguments to domain objects, then calls the corresponding business layer method and translates the result back to DTO objects.

The generation of these service classes uses the generation gap pattern as described in Section 2.2.1, "Generation Gap Pattern". This allows the developer to add manual code when this is needed.

For the service and business layers in the architecture three Spring configuration files are generated:

1. `applicationContext.xml` containing bean definitions for the services including transaction handling. Generated in `generated-resources` and always overwritten.
2. `transactionManagerContext.xml` containing a transaction manager bean. Generated just once in `src/main/resources` and never overwritten. The transaction manager is generally dependent on the environment where the application is deployed and therefore is the responsibility of the developer.
3. `beanRefContext.xml` containing a definition of a context assembled from the above two files.

8.2. Service Method

A service method is a method that is defined in the service model, but will be implemented by hand. A method is defined by the keyword "method", followed by the name of the method. A method may have zero or more input parameters and zero or one output parameter.

```
method addToOrders in [ OrderDto order;  
                        SimpleCustomerDto customer  
out OrderDto;
```

If the parameters or the result of a service method is a list of objects that can be defined by using the keyword "list" as in:

```
method addToOrders in [ list OrderDto order;
                        SimpleCustomerDto customer
                      out list OrderDto;
```

8.2.1. Generated code

For a service method code is generated for service layer in the architecture. A service method results in the generation of a corresponding Java method inside the service class.

As there is no knowledge of the functionality of a custom method, the implementation of a custom method must be done by hand. The generation gap pattern is used on the service class to introduce an extension point where this can be done.

8.3. CRUD: Create, Read, Update, Delete service methods

In a layered architecture certain types of service methods often occur in applications. Methods for creating, reading, updating and deleting (CRUD) business objects are often needed.

For this purpose the service DSL includes special methods which are indicated by their respective keywords.

```
from RecordShopDataContract import SimpleCustomerDto;

create createCustomer for SimpleCustomerDto ;
read readCustomer for SimpleCustomerDto ;
update updateCustomer for SimpleCustomerDto ;
delete deleteCustomer for SimpleCustomerDto ;
```

Because the Dto is defined in a data contract model it needs to be imported first. As these methods are used for defining CRUD functionality for business objects the DTO used must be a class dto.

If you need all of the CRUD methods, you can use a shortcut to specify all four methods as follows:

```
from RecordShopDataContract import SimpleCustomerDto ;

crud SimpleCustomerDto ;
```

In this case the names of the crud methods are derived from the name of the Dto and the domain class it represents. In the above example the names will be *createCustomer*, *readAsSimpleCustomerDto*, *updateCustomer*, and *deleteCustomer*. The read method has a slightly unappealing name, but that's for a reason. If multiple CRUD's are defined with different Dto's representing the same domain class the name *readCustomer* would become ambiguous. The create, delete and update methods have the *SimpleCustomerDto* as one of their parameters and having these with different Dto's of the same type will result in correctly working overloading. The read method only has the id (of type long) as parameter and cannot be overloaded.

8.3.1. Generated code

The CRUD methods have clearly defined meaning and are connected with the domain class they work on through the class dto used. This allows the code generator to generate the full implementation of these methods on both the service and the business layer.

8.3.1.1. Service Layer

Within the service calls of the service model the crud methods are generated with the correct parameters and return values. Only the read method returns the complete dto with all its referenced objects. The update and delete methods return void. The create method returns just the id of the object created. This id can then be used to read the object and then update or delete it. For the example above the interface of the generated methods will be:

```
/**
 * Create a new SimpleCustomerDto.
 *
 * @param object The SimpleCustomerDto to create.
 * @return unique id of the new customer created.
 */
public Long createCustomer(SimpleCustomerDto object);

/**
 * Read an existing SimpleCustomerDto.
 *
 * @param object
 *         The id of the SimpleCustomerDto to read.
 * @return
 */
public SimpleCustomerDto readCustomer(Long id);

/**
 * Update an existing SimpleCustomerDto.
 *
 * @param object
 *         The SimpleCustomerDto containing the modifications for the corres
 */
public void updateCustomer(SimpleCustomerDto object);

/**
 * Delete an existing SimpleCustomerDto
 *
 * @param id
 *         The id of the SimpleCustomerDto to delete.
 */
public void deleteCustomer(SimpleCustomerDto object);
```

Additionally a full implementation of the crud methods is generated. In this implementation the Dto parameters are first translated into domain objects, using the translator methods that have been generated by the data contract DSL. Then the service method on the business layer is called with the domain object as its argument. The result of the business layer service call is then transformed back into a Dto and returned.

The actual crud functionality is performed at the business layer, see next section.

8.3.1.2. Business Layer

In the business layer a method is generated for a crud method which implements the functionality by calling the appropriate method on the Dao in the data layer.

8.4. ListAll and Find Methods

8.4.1. Definition

A listall method allows you to get a list of all objects of a certain type.

```
listall listOrders for OrderDto;
```

The above *listall* method defines three service methods that support retrieving lists of objects of type OrderDto.

1. A service method named **listOrders()** that will get a list of all Order objects represented as OrderDto objects. Be carefull when using this method, it may result in very long lists.
2. A service method **listOrders(int firstResult, int maxResults)** that will get a list of OrderDto objects within a specified range. The range, also called page, is specified by the two parameters. You will typically use this method when you work wiht large sets of objects and want to retrieve them in chunks (pages).
3. A service method **countOrders()** that will return the total number of existing Order objects.

```
find findOrders for OrderDto;
```

The above *find* method defines a service method that will find all Order objects that look like the OrderDto parameter. This is a find by example method. The resutl is a List of OrderDto objects that corresponds to the parameter.

8.4.2. Generated code

For both the *listall* and *find* methods a method in both the service layer and the business layer is generated. As these methods deal directly with domain classes the functionality is fully generated.

8.5. Reference methods

8.5.1. Definition

A reference method allows you to handle changing links between objects. The links between objects are defined in de business class DSL as associations. A reference method can only be defined for a DTO that represents a business class. There are several types of reference methods. The *add* method allows adds an object to an association, the *remove* removes an object from an association.

```
for SimpleCustomerDto reference orders add OrderDto ;
```

The above *add reference method* defines a service method that will add an Order object, represented by an OrderDto, to a Customer object, represented as a SimpleCustomerDto.

```
for SimpleCustomerDto reference orders remove OrderDto;
```

The above *remove reference method* defines a service method that will remove an Order object, represented by an OrderDto, from a Customer object, represented as a SimpleCustomerDto.

```
for SimpleCustomerDto reference orders get OrderDto;
```

The above *get reference method* defines a service method that will get a collection of all Order objects, represented by a list of OrderDto objetcs, of a Customer object, represented as a SimpleCustomerDto.

8.5.2. Generated code

For reference methods a method in both the service layer and the business layer is generated. As these methods deal directly with domain classes the functionality is fully generated.

Chapter 9. Presentation DSL Reference

To Be Done

This chapter provides a description of all the concepts in the Presentation DSL. Each concept is described by a definition which explains the meaning of the concept, and a description of the code generated for the concept.

The Presentation DSL has two central concepts: Dialogue and Process. Dialogues are the things shown to the user in the user interface, while processes represent all of the actions that can be performed. Both processes and Dialogues have a context object on which they work. In the MVC paradigm, the Dialogue is the view, the Process the Controller. and the context object the model.

9.1. Dialogues

9.1.1. Definition

A dialogue describes something that interacts with the user. Usually this is a page in a web browser, but it might be rendered using an audio voice interface as well. In Mod4j a dialogue always defines a part of a page.

Each Dialogue has a context, which is the type of object that this dialogue uses to get its information from.

There are many subtypes of Dialogue that denote specific types of dialogues like ContextForm, MasterDetail, CompoundDialogue etc. These subtypes are all described in the following sections.

9.1.2. Generated code

To Be Done.

9.2. ContentForm

9.2.1. Definition

A ContentForm defines a panel that shows the values of properties of the context object of the ContentForm. It may show all properties, but can also define a subset of the available properties to show.

A ContentForm may be editable allowing the user to change values of the properties.

```
form ShowCustomercontext CustomerDto [  
  element name label Name ;  
  element number label CustomerNr;  
  element title label Title;  
  processes [  
    doSomething;  
  ]  
]
```

In the above example *name*, *number* and *title* must be properties of *CustomerDto*. The string after the *label* keyword is the name shown to the user on the panel. This allows the naming of the properties in the models to be different from the naming on the panels.

The *processes* section shows all the processes that a user can start from this form. These processes

are shown as buttons or links on the panel.

9.2.2. Generated code

TBD: Swap + View + dit panels, property files. .

9.3. Processes

A process consists of a sequential or paralel list of either DialogueCalls or ProcessCalls. A process that cointains only other process calls is called an Automated Process, and a process that calls at least one dialogue is called an InteractiveProcess.

```
InteractiveProcess processname;  
    // methods go here
```

9.3.1. Generated code

For a sprocess one Java class is generated which represents the process. This class conains a method called nextPage(...) which returns the next Wicket page according to the process definition..

For each DialogueCall in a process a dialogePage is created. This dialoguePage is the one returned by the nextPage(...) operation in the process class. The dialogue page only contains the Wicket panel that is generated from the referred Dialogue..

For the service and business layers in the architecture three Spring configuration files are generated:

1. `applicationContext.xml` containing bean definitions for the services including transaction handling. Generated in `generated-resources` and always overwritten.
2. `transactionManagerContext.xml` containing a transaction manager bean. Generated just once in `src/main/resources` and never overwritten. The transaction manager is generally dependent on the environment where the application is deployed and therefore is the responsibility of the developer.
3. `beanRefContext.xml` containing a definition of a context assembled from the above two files.