

mod4j User Guide

**Jos Warmer
Eric Jan Malotaux
Johan Vogelzang**

mod4j User Guide

by Jos Warmer, Eric Jan Malotaux, and Johan Vogelzang

1.0 beta

Copyright © 2009 Ordina and other Mod4j committer

Table of Contents

1. Mod4j Principles and Patterns	1
1.1. Principles	1
1.1.1. Code Generation Principles	1
1.1.2. Modeling Principles	1
1.2. Code Generation Patterns	2
1.2.1. Generation Gap Pattern or Empty Subclass Pattern	2
2. Business Domain DSL Reference	4
2.1. Business Class	5
2.1.1. Definition	5
2.1.2. Generated code	6
2.2. Inheritance	7
2.2.1. Generated code	7
2.3. Attributes	8
2.3.1. Definition	8
2.3.2. Generated code	8
2.4. Enumerations	11
2.4.1. Generated code	11
2.5. Business Rules	11
2.5.1. Generated code	12
2.6. Associations	12
2.6.1. Generated code	12
3. Data Contract DSL Reference	15
3.1. Data Transfer Object (DTO)	15
3.2. CustomDto	15
3.2.1. Generated code	15
3.3. DTO Attributes	15
3.3.1. Definition	15
3.3.2. Generated code	16
3.4. Business Class Dto	16
3.4.1. Generated code	17
3.5. ListDto	17
3.5.1. Generated code	17
3.6. EnumerationDto	18
3.6.1. Generated code	18
4. Service DSL Reference	19
4.1. Service Method	19
4.1.1. Generated code	19
4.2. Create, Read, Update, Delete service methods	19
4.2.1. Generated code	20
4.3. Reference methods	21
4.3.1. Definition	21
4.3.2. Generated code	21

List of Figures

1.1. Empty subclass pattern	3
2.1. Business Domain DSL generation for each Business Class	4
2.2. Business Domain DSL generation for a Business Domain model.	4
2.3. Business Domain DSL generation for a Business Domain model.	5
2.4. Empty subclass pattern applied to the Domain Layer	6
2.5. Empty subclass pattern applied to the Data Layer	7

Chapter 1. Mod4j Principles and Patterns

1.1. 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.

1.1.1. 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 never overwritten**, in other words *the model is always leading*.

To ensure that generated code never needs to be overwritten the generated code is designed and generated with *specific extension points*. These extension points are the only places where handwritten code may be placed. In most cases an extension point is in a separate file than the generated code. To help the developer a first empty extension point file is usually created, but only when the extension point file does not exist.

- **Within Eclipse Mod4j support incremental code generation.** This means that 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 n 'build automatically' is set this will happen. 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).
- **Code generation through Maven.** Mod4j includes a Maven plugin that support the code generation process in batch mode. This is used to run automatic 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. We support two ways of generating code.

1.1.2. Modeling Principles

Modeling in Mod4j is:

- **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. In-

stead 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.

1.2. Code Generation Patterns

The code generated by mod4j is designed to be extensible by handwritten code,. This allows the developer 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 a number of code generation patterns that are used in the various code generators. These patters are applied in the code generators of the various DSLs.

1.2.1. Generation Gap Pattern or Empty Subclass Pattern

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

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.

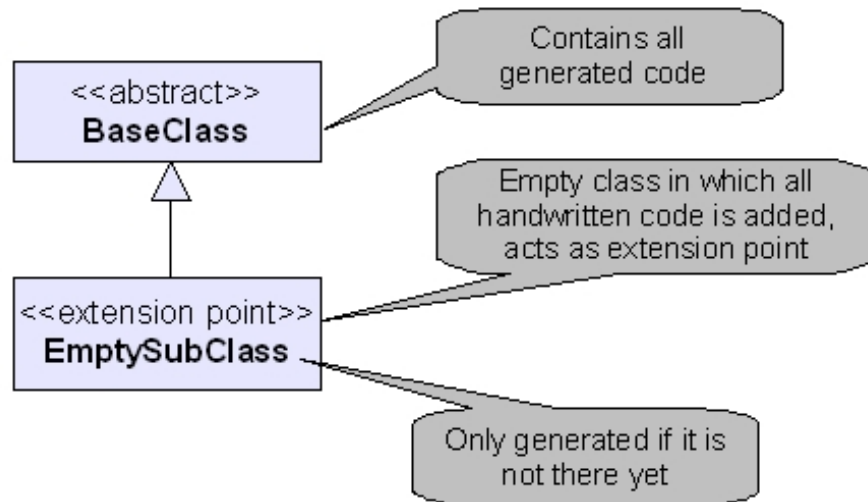
The empty-subclass pattern is used at several places within Mod4j. It is a well known pattern in Java code generation and allows a clear separation of generated and handwritten code. Code generated for a Java class or java interface is done in two parts. An abstract base class is generated which con-

tains all the generated code. An empty concrete subclass is generated. Handwritten code should be put into this subclass.

The generator will always regenerate the abstract base class. The generator will only generate the empty subclass once, if it is not there. In this way handwritten code is never overwritten by the code generator.

The solution is shown in Figure 1.1, "Empty subclass pattern".

Figure 1.1. Empty subclass pattern



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 call 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.

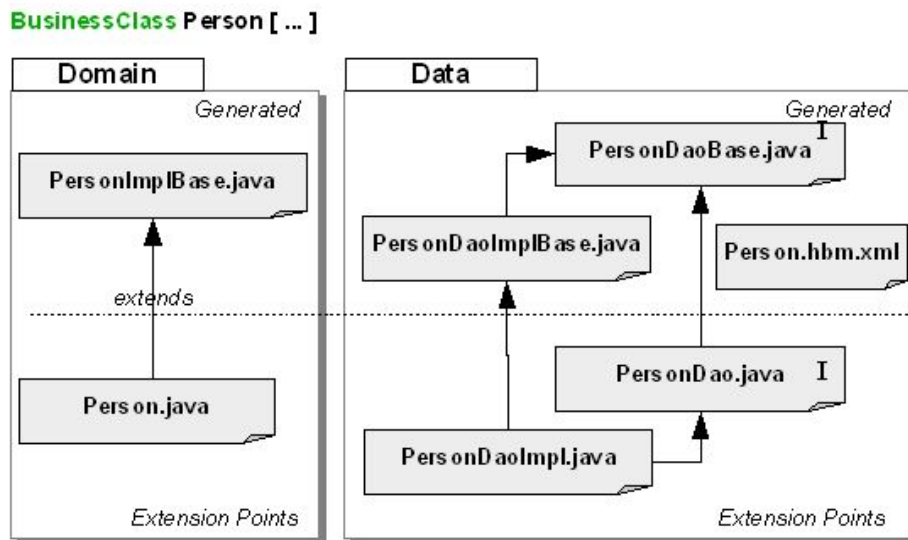
Chapter 2. Business Domain DSL Reference

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 use for modeling the domain model of an application. De Business Domain model consists of all domain classes, as understood by the business expert. Technical aspects do not belong in Business Domain models.

From a business domain model code is generated, conforming to the reference architecture as describe in REF. As a consequence, the code will always conform perfectly to the architecture, ensuring a high quality application.

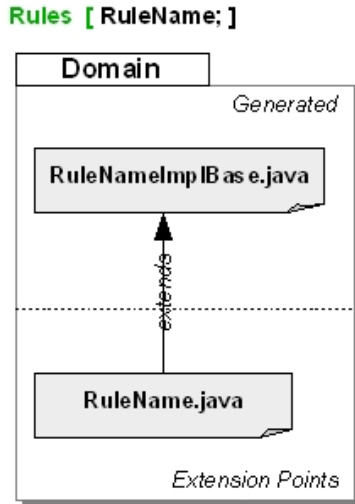
Figure 2.1. Business Domain DSL generation for each Business Class



As you can see in the figure, seven files are generated for each business 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. YTo 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.

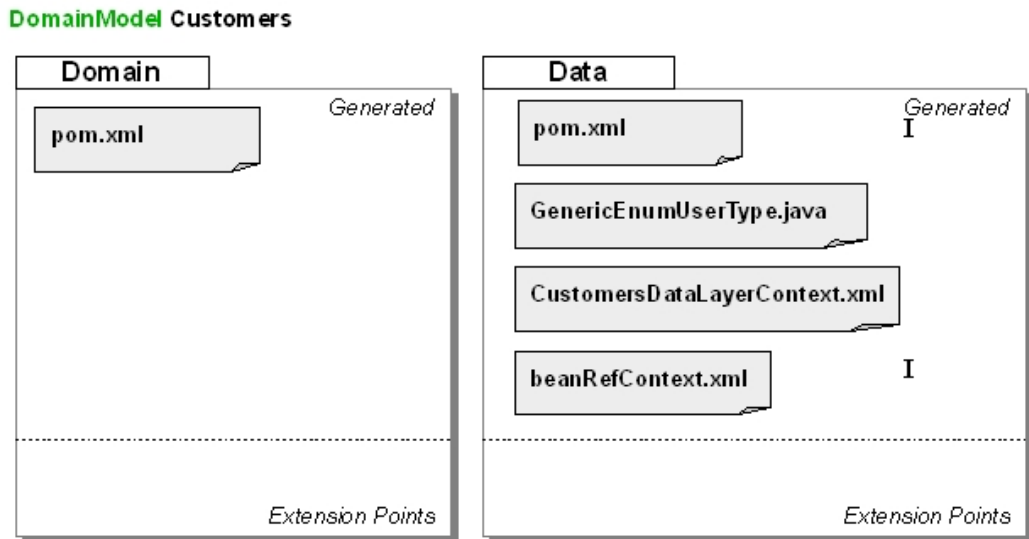
For a business rule two files, following the generation gap pattern are generated. This is shown in figure.

Figure 2.2. 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 businessclasses in the model. This is shown in the following figure.

Figure 2.3. Business Domain DSL generation for a Business Domain model.



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

2.1. Business Class

2.1.1. Definition

```

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

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

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

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

2.1.2. Generated code

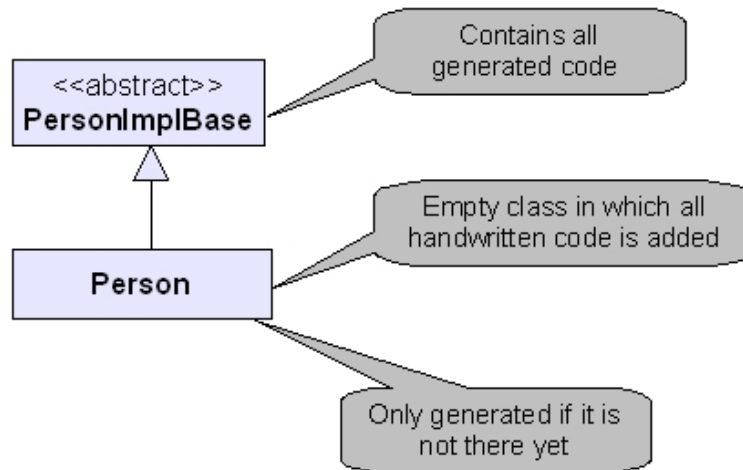
For a business class code is generated for both the domain and the data layer of the architecture.

2.1.2.1. Domain Layer

In the domain code is generated according to the *empty base class pattern* (Section 1.2.1, “Generation Gap Pattern or Empty Subclass Pattern”). For each business class two Java classes are generated. The first Java class is an abstract base class containing all generated code for the business class. All code generated for properties, associations and validation of attribute aconstraints 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 2.4. Empty subclass pattern applied to the Domain Layer



2.1.2.2. Data Layer

In the data layer a number of files is generated.

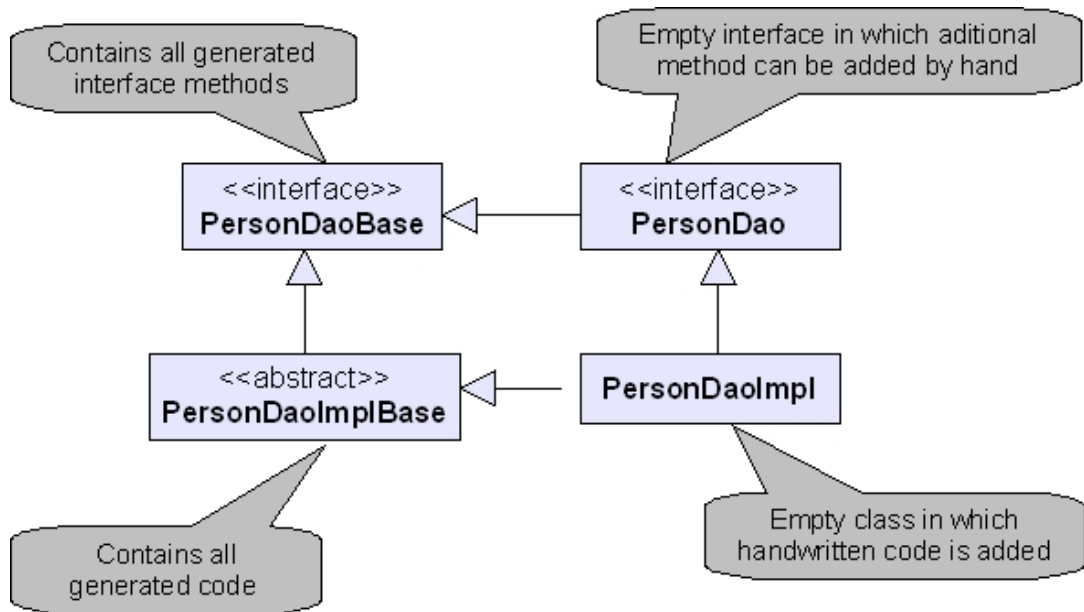
A hibernate mapping file is generated for each business class.

One Hibernate configuration file (how do you call this DatalaagContext.xml file) file is generated

for the application as a whole.

A Dao (Data Access layer) 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 empty subclass pattern. The result for a class called Person is shown in figure Figure 2.5, “Empty subclass pattern applied to the Data Layer”. Note that this is another instance of the empty subclass pattern as described in Section 1.2.1, “Generation Gap Pattern or Empty Subclass Pattern”.

Figure 2.5. Empty subclass pattern applied to the Data Layer



2.2. Inheritance

```

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

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

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

2.2.1. Generated code

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

2.2.1.1. 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*.

2.2.1.2. Data Layer

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

heritance.

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.

2.3. Attributes

2.3.1. Definition

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

```
class Person [  
    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 ;  
    integer numberOfEars; // optional constraints  
    datetime birthdate; // optional constraints  
    decimal length; // optional constraints  
    "Description of the isMale attribute"  
    boolean isMale; // optional constraints  
    // rules go here ...  
]
```

A property describes a characteristic of a business class. Properties always have a defined type. A property can have a predefined type, which can be *integer*, *string*, *decimal*, *boolean* or *datetime*.

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 the above example *minlength*, *maxlength* and *regexp* are such constraints. These constraints must always be true for the property.

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

2.3.2. Generated code

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

As described before (REF) code is generated in the domain layer according to the Empty base class pattern. For each business class an abstract base class is generated. This abstract base class contains a field for each property and getter and setter methods to access the property.

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

If there are constraints specified for the property in the model, validators for the property are also generated. Code is also generated to automatically call these validators whenever the value of the property has changed, e.g. when calling the setter method. Whenever an attempt is made to change the value of a property 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.

2.3.2.1. 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 REF.
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..

2.3.2.1.1. Generated code

Apart from the generic code generated for each property no additional code is generated for a string property.

2.3.2.2. 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.

2.3.2.2.1. Generated code

A boolean attribute that is not *nullable* is implemented by a Java *boolean* field, if the attribute is nullable this is done by a *Boolean* object field.

2.3.2.3. 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.

2.3.2.4. Decimal Attribute

A 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.

2.3.2.5. Datetime Attribute

A datetime attribute is completely defined as shown below

```
"The day on which it all started"  
datetime day started  
    default 4  
    accuracy day  
    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.

2.4. Enumerations

An enumeration defines a data type which can be used as the type of an attribute of a business 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.

2.4.1. Generated code

To Be Done

2.5. Business Rules

A business 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 business class.

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

```
class SampleClass [  
    // properties 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.

2.5.1. 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 empty-subclass pattern on how this is done in the code

2.6. Associations

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

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

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.

TODO: ordered associations and many-to-many associations

2.6.1. 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 updates consistently whenever one of the sides is changed.

2.6.1.1. 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.

```
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 3. 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.

3.1. 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. A `ListDto` defines a collection of other DTOs. All these variants are described in detail below.

3.2. CustomDto

```
custom ExampleDto [  
    // properties go here ...  
]
```

A custom dto describes

A custom dto has properties of different types.

3.2.1. Generated code

For a custom dto

3.3. DTO Attributes

3.3.1. Definition

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

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

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

3.3.2. Generated code

For an attribute code is generated for both the service layer of the architecture.

For each DTO a Java class is generated. This class contains a protected field for each attribute and getter and setter methods to access the field.

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

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

3.4. Business Class Dto

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

```
from RecordShopDomainModel import Customer ;

class SimpleCustomerDto represents Customer [
    // attributes go here
]
```

A *BusinessClassDto* always declares the business class that it represents as its base class. This business class must be defined in a Business Domain model. To ensure that the business 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 *BusinessClassDto* must have the same names as attributes in the base business class. Not every attribute has to be there, allowing a *BusinessClassDto* to contain a subset of attributes from its base business class.

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

```
class SimpleCustomerDto represents Customer [
    // attributes go here

    references [
        orders as OrderNumberAndDateList ;
    ]
]
```

A business class may take part in associations as defined in the business domain model. A *BusinessClassDto* may represent such an association as well. There are restrictions on represented associ-

ations. The main restriction is that only one side of an association is represented as a reference. A Dto does not support bi-directional associations.

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. There are two possibilities.

The association has multiplicity of maximum one. In this case the type of the reference must be a `BusinessClassDto` based on the `BusinessClass` in the association.

If the association has a multiplicity larger than one, the type must be a `ListDto` based on a `BusinessClassDto` based on the `BusinessClass` in the association.

Shorthand

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 `BusinessClassDto`.

```
class FullCustomerDto represents Customer [
```

3.4.1. Generated code

For a `BusinessClassDto` several files are generated.

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

Secondly a Translator class is generated. This class contains two translator methods to translate the `BusinessClass Dto` to a Domain class and vice versa. For example for the `BusinessClassDto 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);
```

3.5. ListDto

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

```
list CustomerList base FullCustomerDto ;
```

This example shows the `ListDto` named `CustomerList` which contains objects of type `FullCustomerDto`.

3.5.1. Generated code

For a `ListDto` no specific classes are generated.

At all places where a `ListDto` is used in any DSL model , the Java `List` class is used in the code.

3.6. EnumerationDto

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 shown 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.

3.6.1. Generated code

To Be Done

Chapter 4. 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).

The

4.1. 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;
```

4.1.1. 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.

4.2. 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

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

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 Business-ClassDto.

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

```
from RecordShopDataContract import SimpleCustomerDto ;

for SimpleCustomerDto crud ;
```

In this case the names of the crud methods are derived from the name of the Dto and the business class it represents. In the above example the names will be createCustomer, readAsSimpleCustomerDto, updateCustomer, 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 business-

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.

4.2.1. Generated code

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

4.2.1.1. Service Layer

Within the service class of the service model the crud methods are generated with the correct parameters and return values. For the example above this will be

```
/**
 * Create a new SimpleCustomerDto.
 *
 * @param object
 *         The SimpleCustomerDto to create.
 * @return the SimpleCustomerDto created, possibly modified during creation.
 */
public SimpleCustomerDto 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 corresponding
 * @return the SimpleCustomerDto updated, possibly modified during update.
 */
public SimpleCustomerDto updateCustomer(SimpleCustomerDto object);

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

Additionally an implementation of the crud methods is generated. In this implementation the Dto parameters is first translated into a business object, 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 business 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.

4.2.1.2. 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.

4.3. Reference methods

4.3.1. Definition

A reference method allows you to handle changing links between objects. There are two 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 ;  
for SimpleCustomerDto reference orders remove OrderDto;
```

An .

4.3.2. Generated code

For a reference method full code is generated to code is generated.