The Lunifera Entity DSL

1 Purpose

The “Lunifera Entity DSL” facilitates the handling of persistence entities. Defining entities using the Lunifera Entity DSL efficiently creates a clean entity model that contains all relevant semantic elements of the model text file. This semantic model is used to automatically transform the semantic information into proper Java code with the respective annotations for a persistence provider.

2 Overview

The main semantic elements of the Lunifera Entity DSL are:

- “Package” – the root element that contains all the other elements. A model can contain multiple packages.
- “Import” declarations – used to import external models or even Java classes.
- “Datatype” declarations – a way to define datatypes that can be used in entities and beans.
- “Entity” – the abstraction of a business model entity. It contains further elements such as properties and references.
- “Bean” – does not compile to a JPA Entity but to a Java Bean (POJO with getter and setter and PropertyChange-Support). Beans may be used as temporary containers in entity operations or can be embedded into JPA Entities.
- “Enum” – the abstraction for Java enums.
- “Property” – a reference to an embedded Bean, an Enum, a Java class or a “simple datatype” (as defined in the datatype declaration). Offers multiplicity.
- “Reference” – a reference to another Entity (or to another Bean in the case of a Bean). Offers multiplicity.
- “Operations” – similar to Java methods. The Xbase expression language can be used to write high-level code.
- “Annotations“ can be placed on Entity, Property and Reference.
- “Comments” can be added to all elements.

3 Package

Packages are the root element of the Lunifera Entity DSL grammar. Everything is contained in a package: Imports, Entities, Beans and Enums have to be defined inside the Package definition. One document can contain multiple packages with unique names.

The elements a package can contain are Entities, Beans and Enums.

Additionally, a package allows Import statements and the declaration of datatypes.

```
1  package org.lunifera.entitydsl.documentation.demo {
2
3      entity Foo {
4
5      }
6
7      bean Bar {
8
9      }
10
11 }
```

Package.png: A package is the topmost element and contains other items.

Imports

The “Entity DSL” allows to reference entities defined in different packages. The import statement is a way to address these elements by their fully qualified name.

Import statements allow the use of the *-wildcard.

```
1  package org.lunifera.entitydsl.documentation.demo {
2
3      import org.lunifera.entitydsl.documentation.common.*;
4
5      entity Foo extends Bar {
6
7      }
8
9  }
```

```
1  package org.lunifera.entitydsl.documentation.common {
2
3      entity Bar {
4          id long id;
5      }
6
7  }
8
9
```

Import.png: Items contained in another package can be addressed if the package is imported.

Datatypes

The “Entity DSL” allows the definition of datatypes. These are translated by the inferrer into their standard Java presentation. The behaviour of the generator can be controlled by the datatype definitions.

There are three types of datatype definitions:

- **jvmTypes**

  Datatpye definitons that map types to jvmTypes take the basic form of

  `datatype <name> jvmType <type>.`

  Specifying datatypes in this manner uses an appropriate wrapper class in the

generated Java code; adding the keyword “as primitive” enforces the use of primitive datatypes where applicable:

`datatype foo jvmType Integer` compiles to Integer whereas `datatype foo jvmType Integer as primitive` results in “int”.

```
1 package org.lunifera.entitydsl.documentation.demo {
2
3     datatype foobar jvmType java.lang.Integer;
4
5     entity Foo  {
6         id foobar id;
7     }
8 }
```

```
14 public class Foo {
15     @Transient
16     private boolean disposed;
17
18     @Id
19     @GeneratedValue
20     private Integer id;
21
```

Datatypeinteger.png: The defined datatype is translated to a wrapper class.

```
1 package org.lunifera.entitydsl.documentation.demo {
2
3     datatype foobar jvmType java.lang.Integer as primitive;
4
5     entity Foo  {
6         id foobar id;
7     }
8 }
```

```
15 public class Foo {
16     @Transient
17     private boolean disposed;
18
19     @Id
20     @GeneratedValue
21     private int id;
22
```

Datatypeint.png: By adding the “as primitive” keyword, the datatype is translated to a primitive datatype.

- **dateTypes**

  The datatypes for handling temporal information can be defined by the following statement:

  `datatype foo dateType <date|time|timestamp>`

  Datatypes that have been defined in this manner can be used as property variables in entities and beans.

```
1 package org.lunifera.entitydsl.documentation.demo {
2
3     datatype foobar dateType date;
4
5     entity Foo  {
6         id long id;
7         var foobar sometime;
8     }
9 }
10
```

```
date
time
timestamp
```

Datetype.png: Defining datatypes for handling temporal information. Content assist is available.

- **Blobs**

  Binary blobs can be handled by defining a datatype with the “as blob” keyword. The Java implementation of such a blob is a byte array. Appropriate persistence annotations are automatically added.

```
package org.example {

    datatype long jvmType Long as primitive;
    datatype blobtype as blob;

    entity FooBar {
        id long id;
        var blobtype myblob;
    }

}
```

```
public class FooBar {
  @Transient
  private boolean disposed;

  @Id
  @GeneratedValue
  private long id;

  @Column(name = "MYBLOB")
  @Lob
  @Basic(fetch = FetchType.LAZY)
  private byte[] myblob;
```

Blob.png: Including binary blobs by using a datatype with the “as blob” keyword.

4 Entities

Entities are the most complex elements in the Lunifera Entity DSL. An entity is an abstraction above a business object. Entities are defined by its name and properties, references and operations. Generally, an entity is an object which can keep a state about variables and references and can persisted.

For each entity that is defined in a package, a Java class and the corresponding persistence structure is automatically generated.

```
1  package org.lunifera.entitydsl.documentation.demo {
2
3      entity Foo  {
4          id long id;
5          var String name;
6          var String[*] info;
7      }
8  }
9
10
11
```

```
1  package org.lunifera.entitydsl.documentation.demo;
2
3  import java.util.ArrayList;
14
15 @Entity
16 @Table(name = "FOO")
17 @DiscriminatorValue(value = "FOO")
18 @SuppressWarnings("all")
19 public class Foo {
20   @Transient
21   private boolean disposed;
22
23   @Id
24   @GeneratedValue
25   private long id;
26
27   @Column(name = "NAME")
28   private String name;
29
30   @ElementCollection
31   @Column(name = "INFO")
32   private List<String> info;
```

Entity.png: The defined entity is translated to a Java class with getter and setter methods as well as the appropriate annotations for the persistence provider.

Entity Modifiers

The following modifiers can be placed before the “entity” keyword:

- abstract – marks the entity to be abstract. This generates an abstract Java class.

```
1  package org.lunifera.entitydsl.documentation.demo {
2
3      abstract entity Foo  {
4          id long id;
5          var String name;
6          var String[*] info;
7      }
8  }
9
10
11
```

```
1  package org.lunifera.entitydsl.documentation.demo;
2
3  import java.util.ArrayList;
14
15 @Entity
16 @Table(name = "FOO")
17 @DiscriminatorValue(value = "FOO")
18 @SuppressWarnings("all")
19 public abstract class Foo {
20   @Transient
21   private boolean disposed;
```

Abstract.png: The “abstract” keyword causes the translation into an abstract Java class.

- historized – marks the entity to be historized. Historized entities can have several entries in a database, but only one of them may be marked as current. The “historized” keyword adds an object ID (OID), a version field and a flag for the current version to the entity.

```
package org.example {

    datatype long jvmType Long as primitive;
    datatype String jvmType String;


    historized entity Foo {
        uuid String id;
    }


}
```

```
package org.example;

import javax.persistence.Column;

@Entity
@Table(name = "FOO")
@DiscriminatorValue(value = "FOO")
@SuppressWarnings("all")
public class Foo {
  @Transient
  private boolean disposed;

  @Id
  private String id = java.util.UUID.randomUUID().toString();

  @Column(name = "OBJ_ID")
  private String objId = java.util.UUID.randomUUID().toString();

  @Column(name = "OBJ_VERSION")
  private int objVersion;

  @Column(name = "OBJ_CURRENT")
  private boolean objCurrent;
```

Historized.png: The “historized” modifier triggers the creation of an OID, an object version and a flag for marking the current version.

- cacheable – marks an entity as cacheable. The appropriate annotation for the persistence provider is added to the generated Java code.

```
package org.example {

    datatype long jvmType Long as primitive;
    datatype String jvmType String;

    cacheable entity Foo {
        uuid String id;
    }
}
```

```
package org.example;

import javax.persistence.Cacheable;

@Entity
@Table(name = "FOO")
@DiscriminatorValue(value = "FOO")
@Cacheable
@SuppressWarnings("all")
public class Foo {
```

Cacheable.png: Declaring an entity to be cacheable adds the “@Cacheable” annotation.

- timedependent – marks the entity to be time-dependent. An object may have several entries in the database. Which entry is valid will be determined by “valid from” and “valid until” fields that are added to the entity. An object ID is created in order to tie the entries together. The “timedependent” keyword recognizes the modifiers “(DATE)” and “(TIMESTAMP)”. Default is timedependent(DATE).

```
package org.example {

    datatype long jvmType Long as primitive;
    datatype String jvmType String;


    timedependent(DATE) entity Foo {
        uuid String id;
    }


}
```

```
package org.example;

import java.util.Date;

@Entity
@Table(name = "FOO")
@DiscriminatorValue(value = "FOO")
@SuppressWarnings("all")
public class Foo {
  @Transient
  private boolean disposed;

  @Id
  private String id = java.util.UUID.randomUUID().toString();

  @Column(name = "OBJ_ID")
  private String objId = java.util.UUID.randomUUID().toString();

  @Column(name = "VALID_FROM")
  @Temporal(value = TemporalType.DATE)
  private Date validFrom;

  @Column(name = "VALID_UNTIL")
  @Temporal(value = TemporalType.DATE)
  private Date validUntil;
```

Timedependent.png: The “timedependent” keyword causes an object ID, a “validFrom” and a “validUntil” field to be created in order to support multiple database entries for an object.

- mapped superclass – marks a class that provides persistent entity state and mapping information for its subclasses, but which is not an entity itself. Typically, the purpose of a mapped superclass is to define state and mapping information that is common to multiple entities. All the mappings from the mapped superclass are inherited to its subclasses as if they had been defined there directly.

```
1 package org.lunifera.entitydsl.documentation.demo {
2
3     mapped superclass Foo  {
4         id long id;
5         var String name;
6         var String[*] info;
7     }
8 }
9
```

```
1  package org.lunifera.entitydsl.documentation.demo;
2
3  import java.util.ArrayList;
12
13 @MappedSuperclass
14 @SuppressWarnings("all")
15 public class Foo {
16   @Transient
17   private boolean disposed;
```

Mappedsuperclass.png: The “mapped superclass” keyword sets the appropriate annotation which causes the persistence provider to move the mappings to the derived subclasses.

The following modifier can be placed after the “entity” keyword:

- extends – marks an entity that is derived from another entity. That means that the properties and references of the parent entity are inherited.

```
1  package org.lunifera.entitydsl.documentation.demo {
2
3      entity Foo {
4          id long id;
5          var String name;
6      }
7
8      entity Bar extends Foo {
9          var long number;
10     }
11
12 }
```

```
1  package org.lunifera.entitydsl.documentation.demo;
2
3  import javax.persistence.Column;
8
9  @Entity
10 @Table(name = "BAR")
11 @DiscriminatorValue(value = "BAR")
12 @SuppressWarnings("all")
13 public class Bar extends Foo {
14   @Column(name = "NUMBER")
15   private long number;
16
```

Extends.png: The “extends” keyword causes a Java subclass to be created.

Persistence Settings

Apart from the “mapped superclass” setting that moves all property columns to the tables belonging to derived classes, the following settings for table inheritance can be chosen within an entity definition:

- inheritance per class – Causes a table to be created for each class; subclasses share this table using a discriminator value. This statement has to be followed by braces inside of which further details can be specified.

```
package org.example {

    datatype long jvmType Long as primitive;

    entity Base {
        inheritance per class {}
        id long id;
    }

    entity Derived extends Base {
        var long foo;
    }

}
```

```
package org.example;

import javax.persistence.Column;
import javax.persistence.DiscriminatorValue;
import javax.persistence.Entity;
import org.example.Base;

@Entity
@DiscriminatorValue(value = "DERIVED")
@SuppressWarnings("all")
public class Derived extends Base {
  @Column(name = "FOO")
  private long foo;
```

Inheritancebyclass.png: A single table for entity Base is created; the generated Java code for the “Derived” class shows that the “Derived” entity is added to this table by using a discriminator.

- inheritance per subclass – Causes a table to be created for each subclass. This statement has to be followed by braces inside of which further details can be specified. This is the default behaviour of no inheritance strategy is specified.

```
package org.example {

    datatype long jvmType Long as primitive;

    entity Base {
        inheritance per subclass {}
        id long id;
    }

    entity Derived extends Base {
        var long foo;
    }

}
```

```
package org.example;

import javax.persistence.Column;
import javax.persistence.DiscriminatorValue;
import javax.persistence.Entity;
import javax.persistence.Table;
import org.example.Base;

@Entity
@Table(name = "DERIVED")
@DiscriminatorValue(value = "DERIVED")
@SuppressWarnings("all")
public class Derived extends Base {
  @Column(name = "FOO")
  private long foo;
```

Inheritancebysubclass.png: An “@Table” annotation is added to the generated Java code, so the “Derived” entity is mapped to a table of its own.

The structure of the created database can be controlled by the following settings:

- schemaName – allows the specification of a name for the database schema to be used. This setting is translated to the appropriate JPA annotation “@Table(schema = xyz)”. The schemaName given is converted to snake case using capitals.
- tableName – allows the specification of a name for the table (within the database schema) which the entity is mapped to. This setting is translated to the appropriate JPA annotation “@Table(name = xyz)”. The tableName given is converted to snake case using capitals. The default value is the name of the entity.

```
package org.example {

    datatype long jvmType Long as primitive;

    entity Foobar {
        schemaName Foo;
        tableName Bar;
        id long id;
    }
}
```

```
package org.example;

import javax.persistence.DiscriminatorValue;

@Entity
@Table(schema = "FOO", name = "BAR")
@DiscriminatorValue(value = "BAR")
@SuppressWarnings("all")
public class Foobar {
  @Transient
  private boolean disposed;
```

Schematablename.png: Specifying the schemaName and tableName settings in an entity controls the name of the database schema and tables used for persistence.

- discriminatorColumn – allows the definition of a name for the discriminator column in the case of inheritance per class. Can be set within the braces after the “inheritance” statement and has to be followed by a semicolon. Defaults to “DISC”.
- discriminatorType – allows the definition of the datatype used as discriminator within the single table. Can be set to CHAR, INT, STRING or INHERIT. Can be set within the braces after the “inheritance” statement and has to be followed by a semicolon. Defaults to “STRING”.
- DiscriminatorValue – allows a custom value to be used as discriminator within the single table. Can be set within the braces after the “inheritance” statement and has to be followed by a semicolon. Defaults to the entity name converted to snake case.

my.entitymodel

```
package org.example {

    datatype long jvmType Long as primitive;

    entity Foo {
        inheritance per class {
            discriminatorColumn TYPE_DISCR;
            discriminatorType STRING;
            discriminatorValue BASE_ENTITY;
        }
        id long id;
    }

    entity Bar extends Foo {
        inheritance per class {
            discriminatorValue DERIVED_ENTITY;
        }

    }
}
```

Foo.java

```
package org.example;

import javax.persistence.DiscriminatorColumn;

@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "TYPE_DISCR", discriminatorType = DiscriminatorType.STRING)
@DiscriminatorValue(value = "BASE_ENTITY")
@SuppressWarnings("all")
public class Foo {
  @Transient
  private boolean disposed;
```

Bar.java

```
package org.example;

import javax.persistence.DiscriminatorValue;

@Entity
@DiscriminatorValue(value = "DERIVED_ENTITY")
@SuppressWarnings("all")
public class Bar extends Foo {
```

Discriminator.png: The inheritance statement allows the specification of discriminator column, type and value to be used in the case of a single table.

5 Beans

Beans are objects that are embedded in other entities, inheriting their persistence and lifecycle. Similar to Entities, Beans are characterised by their name, their properties and references. For each bean that is defined in a package, a Java class is automatically generated.

Beans can be embedded into entities by defining them as properties of the respective entity. The appropriate annotations (@Embeddable, @Embedded, @AttributeOverrides

etc.) are added in order to have beans persisted with their parent entities.

my.entitymodel

```
package org.example  {

    datatype long jvmType Long as primitive;

    entity Foo {
        id long id;
        var BarBean mybean;
    }

    bean BarBean {
        var long data;
    }

}
```

Foo.java

```
package org.example;

import javax.persistence.AttributeOverride;

@Entity
@Table(name = "FOO")
@DiscriminatorValue(value = "FOO")
@SuppressWarnings("all")
public class Foo {
  @Transient
  private boolean disposed;

  @Id
  @GeneratedValue
  private long id;

  @Embedded
  @AttributeOverrides(value = @AttributeOverride(name = "data", column = @Column(name = "MYBEAN_DATA")))
  @Column(name = "MYBEAN")
  private BarBean mybean;
```

BarBean.java

```
package org.example;

import java.io.Serializable;

@Embeddable
@SuppressWarnings("all")
public class BarBean implements Serializable {
  private boolean disposed;

  @Basic
  private long data;
```

Beans.png: Beans can be embedded in entities and are persisted with them.

6 Enums

Enums are an abstraction above the Java enum. They compile to enum classes and can be used as properties in Entities and Beans.

my.entitymodel

```
package org.example  {

    datatype long jvmType Long as primitive;

    entity Foo {
        id long id;
        var BarEnum status;
    }

    enum BarEnum {
        ON, OFF, ERROR
    }

}
```

Foo.java

```
package org.example;

import javax.persistence.Column;

@Entity
@Table(name = "FOO")
@DiscriminatorValue(value = "FOO")
@SuppressWarnings("all")
public class Foo {
  @Transient
  private boolean disposed;

  @Id
  @GeneratedValue
  private long id;

  @Column(name = "STATUS")
  private BarEnum status;
```

BarEnum.java

```
package org.example;

@SuppressWarnings("all")
public enum BarEnum {
  ON,

  OFF,

  ERROR;
}
```

Enum.png: Defining enums allows using them as variables in entities and beans.

7 Properties

Properties of Entities and Beans are references to datatypes or enums. They can be regarded as variables and are defined by a keyword followed by a datatype (Java type or datatype defined in the datatype section) and a name. By defining a Bean as a property of an entity, it is embedded in it.

Property Keywords

- var – The basic property; defines a variable that is persisted in a table column.

```
package org.example  {

    datatype long jvmType Long as primitive;
    datatype String jvmType String;

    entity Foo {
        id long id;
        var long number;
        var String name;
    }

}
```

```
package org.example;

import javax.persistence.Column;

@Entity
@Table(name = "FOO")
@DiscriminatorValue(value = "FOO")
@SuppressWarnings("all")
public class Foo {
  @Transient
  private boolean disposed;

  @Id
  @GeneratedValue
  private long id;

  @Column(name = "NUMBER")
  private long number;

  @Column(name = "NAME")
  private String name;
```

Var.png: Variables can be defined using the "var" keyword. The appropriate persistence settings are genereated automatically.

- id (deprecated) – defines an id-property (used as primary key by the JPA compiler). If no id is given, a warning is shown. Caution: ID autogeneration causes problems since the value is not set before the entity is persisted in the database, which can cause problems. It is advised to use the "uuid" keyword instead.

```
package org.example  {

    datatype long jvmType Long as primitive;
    datatype String jvmType String;

    entity Foo {
        id long id;
    }

}
```

```
package org.example;

import javax.persistence.DiscriminatorValue;

@Entity
@Table(name = "FOO")
@DiscriminatorValue(value = "FOO")
@SuppressWarnings("all")
public class Foo {
  @Transient
  private boolean disposed;

  @Id
  @GeneratedValue
  private long id;
```

Id.png: Entities are supposed to have an ID property that is used as primary key in the database. Caution: The generated IDs are not necessarily unique!

- uuid – allows the use of Universally Unique IDs as primary keys. A new UUID value is created for each object as soon as it is created, independently of database operations. This circumvents the problems with the "id" keyword. UUIDs have to be strings.

```
package org.example  {

    datatype long jvmType Long as primitive;
    datatype String jvmType String;

    entity Foo {
        uuid String id;
    }


}
```

```
package org.example;

import javax.persistence.DiscriminatorValue;

@Entity
@Table(name = "FOO")
@DiscriminatorValue(value = "FOO")
@SuppressWarnings("all")
public class Foo {
  @Transient
  private boolean disposed;

  @Id
  private String id = java.util.UUID.randomUUID().toString();
```

Uuid.png: By using the "uuid" keyword, entities are created with a reliably unique identifier.

- version – defines a version-property (used by the JPA-Compiler)

```
package org.example  {

    datatype long jvmType Long as primitive;
    datatype String jvmType String;

    entity Foo {
        uuid String id;
        version long build;
    }
}
```

```
@Entity
@Table(name = "FOO")
@DiscriminatorValue(value = "FOO")
@SuppressWarnings("all")
public class Foo {
  @Transient
  private boolean disposed;

  @Id
  private String id = java.util.UUID.randomUUID().toString();

  @Version
  private long build;
```

version.png: A version property can be added to entities and beans.

- transient – Marks the property to be transient. Instead of "@Column", an "@Transient" annotation is generated so the property is not persisted in the database.

```
package org.example  {

    datatype long jvmType Long as primitive;
    datatype String jvmType String;

    entity Foo {
        uuid String id;
        transient long discard;
    }
}
```

```
@Entity
@Table(name = "FOO")
@DiscriminatorValue(value = "FOO")
@SuppressWarnings("all")
public class Foo {
  @Transient
  private boolean disposed;

  @Id
  private String id = java.util.UUID.randomUUID().toString();

  @Transient
  private long discard;
```

Transient.png: The "transient" keyword allows properties to be excluded from persistence.

8 References

The Lunifera Entity DSL tracks relationships between entities by using the concept of references. References can exist between objects of the same nature (entities to entities, beans to beans). Where appropriate, back references ("opposite") are added.

References are defined by the "ref" keyword, a type and a name. The exact type of reference can be specified as follows:

Reference Modifiers

- Multiplicity and nullability: The Lunifera Entity DSL supports the specification of multiplicities in square brackets appended to the type:
  [1] defines a non-nullable one-to-one relationship.
  [0..1] defines a nullable one-to-one relationship (Default behavior).
  [1..*] defines a non-nullable one-to-many relationship. Needs an opposite reference.
  [0..*] or simply [*] defines a nullable one-to-many relationship. Needs an opposite reference.

```
package org.example {

    datatype long jvmType Long as primitive;
    datatype String jvmType String;

    entity Foo {
        uuid String id;
        ref Bar optionalBar;
        ref Bar[1] requiredBar;
    }

    entity Bar {
        id long id ;
        var String name ;
    }

}
```

```
package org.example;

import javax.persistence.DiscriminatorValue;

@Entity
@Table(name = "FOO")
@DiscriminatorValue(value = "FOO")
@SuppressWarnings("all")
public class Foo {
  @Transient
  private boolean disposed;

  @Id
  private String id = java.util.UUID.randomUUID().toString();

  @ManyToOne(fetch = FetchType.LAZY)
  @JoinColumn(name = "OPTIONAL_BAR")
  private Bar optionalBar;

  @ManyToOne(optional = false, fetch = FetchType.LAZY)
  @JoinColumn(name = "REQUIRED_BAR", nullable = false)
  private Bar requiredBar;
```

Refnullability.png: Adding multiplicity "[1]" causes the annotations for non-nullable database entries to be set.

- opposite reference: Lifecycle references need the specification of an opposite reference. Using the opposite reference, it is possible to navigate back to the original object after following the reference.
- cascade: The "cascade" specifier controls the behaviour of the database on delete operations. If an object with a "ref cascade" reference to other objects is deleted, those will be removed as well.

```
package org.example {

    datatype long jvmType Long as primitive;
    datatype String jvmType String;

    entity Foo {
        uuid String id;
        ref cascade Bar[0..*] optionalBars opposite foo;
        ref cascade Bar[1..*] requiredBars opposite foo;
    }

    entity Bar {
        id long id ;
        var String name;
        ref Foo foo;
    }
}
```

```
package org.example;

import java.util.ArrayList;

@Entity
@Table(name = "FOO")
@DiscriminatorValue(value = "FOO")
@SuppressWarnings("all")
public class Foo {
  @Transient
  private boolean disposed;

  @Id
  private String id = java.util.UUID.randomUUID().toString();

  @JoinColumn(name = "OPTIONAL_BARS")
  @OneToMany(mappedBy = "foo", cascade = CascadeType.ALL, orphanRemoval = true)
  private List<Bar> optionalBars;

  @JoinColumn(name = "REQUIRED_BARS", nullable = false)
  @OneToMany(mappedBy = "foo", cascade = CascadeType.ALL, orphanRemoval = true)
  private List<Bar> requiredBars;
```

Opposite.png: By using the "cascade" and "opposite" keywords, bidirectional associations can be achieved.

9 Operations

Operations are other important features. They are based on Xbase and offer a huge set of semantic features, extending the featureset of Java. Xbase additionally offers features like closures.

Operations can be declared by the “def” keyword.

```
package org.example {

    datatype long jvmType Long as primitive;
    datatype String jvmType String;

    entity Foo {
        uuid String id;
        var String name;
        def String capsName() {
            return name.toString.toUpperCase;
        }
    }

}
```

```
/**
 * Sets the name property to this instance.
 */
public void setName(final String name) {
  checkDisposed();
  this.name = name;
}

public String capsName() {
  String _string = this.name.toString();
  return _string.toUpperCase();
}
}
```

Def.png: The “def” keyword allows the inlining of custom methods in the Lunifera Entity DSL code. These methods are translated to the appropriate Java methods along with the auto-generated getter and setter methods.

10 Annotations

Annotations can be added to all elements except “import declarations”. Specifying annotations in the Lunifera Entity DSL works in a straightforward manner; content assist is available. The added annotations are taken over into the generated Java code.

```
package org.example {

    datatype long jvmType Long as primitive;
    datatype String jvmType String;

    entity Foo {
        uuid String id;
        @java.lang.Deprecated
        var String test;
    }

}
```

```
@Entity
@Table(name = "FOO")
@DiscriminatorValue(value = "FOO")
@SuppressWarnings("all")
public class Foo {
  @Transient
  private boolean disposed;

  @Id
  private String id = java.util.UUID.randomUUID().toString();

  @Deprecated
  @Column(name = "TEST")
  private String test;
```

Annotation.png: Annotating elements in the Lunifera Entity DSL causes an annotation to be inserted into the generated Java code.

11 Comments

Comments can be added anywhere in an entitymodel text file and are copied over into the generated Java code. Comments are enclosed in /* ... */.

```
package org.example {

    datatype long jvmType Long as primitive;
    datatype String jvmType String;

    /*
     * Base entity
     */
    entity Foo {

        /*
         * Unique ID
         */
        uuid String id;

        /* Count */
        var long number;
    }


}
```

```
package org.example;

import javax.persistence.Column;

/**
 * Base entity
 */
@Entity
@Table(name = "FOO")
@DiscriminatorValue(value = "FOO")
@SuppressWarnings("all")
public class Foo {
  @Transient
  private boolean disposed;

  /**
   * Unique ID
   */
  @Id
  private String id = java.util.UUID.randomUUID().toString();

  /**
   * Count
   */
  @Column(name = "NUMBER")
  private long number;
```

Comment.png: The comments added in the Entity DSL are transformed to Java-style comments in the generated code.

Comments before the “package” definition are copied over to all generated Java classes – this is the place for copyright notices.

```
1  /*
2   * This comment is copied over to all generated classes.
3   * Put copyright info here.
4   */
5
6  package org.lunifera.dsl.entitydsl.histtimevalidation {
7
8      datatype String jvmType String;
9
10     /* Comment for Test.java only */
11     entity Test {
12         inheritance per subclass{}
13         uuid String id;
14     }
15
16     /* Comment for SubTest.java only */
17     entity SubTest extends Test {
18         var String something;
19     }
20
21 }
```

*Commentcopyright.png: A comment before the “package” keyword ends up in **all** generated Java classes.*

12 Reserved words

The keywords of the Lunifera Entity DSL are syntactic features and can therefore not be used as semantic identifiers. In order to circumvent this, it is possible to escape them with the “^” character. During the generation of the Java code, the escape character is removed.

```
1  package org.lunifera.entitydsl.documentation {
2
3      datatype String jvmType String;
4
5      entity Foo {
6          uuid String ^uuid;
7      }
8
9  }
10
11
12
13
14
```

```
1  package org.lunifera.entitydsl.documentation;
2
3  import javax.persistence.DiscriminatorValue;
8
9  @Entity
10 @Table(name = "FOO")
11 @DiscriminatorValue(value = "FOO")
12 @SuppressWarnings("all")
13 public class Foo {
14   @Transient
15   private boolean disposed;
16
17   @Id
18   private String uuid = java.util.UUID.randomUUID().toString();
19
```

Escapecharacter.png: Using the escape character “^”, it is possible to achieve an identifier in the generated Java code with a name that would be a reserved word in the Lunifera Entity DSL. In this case, a variable name “uuid” is generated that cannot be specified in the entity model file.