Modeling: User's Guide

Release 0.9

Sébastien Bigaret

March 4, 2006

Email: sbigaret@users.sourceforge.net

CONTENTS

CHAPTER

ONE

Introduction

The Modeling framework intends to fill the gap between the python object world and relational databases. It relies on a model, based on Entity-Relationship Modelling, that describes how the two worlds map to each other. From your design of such a model, the database's schema and corresponding python classes are automatically generated. Thus, once you have designed how your classes should be stored in the RDBMS, you can focus on the real challenges – the logic of your business objects – while remaining in the object-oriented world of those objects and never having to worry about the SQL and RDBMS persistence layer below.

1.1 A historic preamble

This framework is intended to be an open-source port of the Enterprise Object Framework ™ [hereafter called EOF], a NeXT technology now owned by Apple Computer, Inc. and used in its WebObjects™ application server. I have worked for two years using this framework before I switched to an other company and a new technology (zope+python); and, of course, once I had seen the power of such a framework, that was pretty hard to go back to manually preparing the DB schema, the SQL statements, etc... Discovering python and zope, and since using the original framework was not an option for the company I work for, I began developing a few tools to help me with DB-integration into an object world and, after a few shaky tries, I realized that what I really missed and was eager for was the EOF and that, instead of trying to re-invent the wheel, I'd better go ahead and develop my own version of the framework.

Disclaimer: The EOF goes much farther than this framework: it is well tested and robust for many years and in any case should not be assimilated to what I have done. If you have the opportunity to buy and use the original EOF, go for it, as this is a really pleasant and instructive journey.

1.2 About this manual

tbd.

Part I

The essentials

CHAPTER

TWO

Entity-Relationship Models

Introduction

Models define how an object model maps to a relational model. The model used follows the Entity-Relationship Modelling. This discipline is still very active, even if it was initiated by M. Peter P. Chen in 1976 (one of the most cited and influent paper in computer science (rank 36), see here (at http://bit.csc.lsu.edu/~chen/display.html)). Some other references can be found at the end of this chapter.

Models may be specified in one of two equivalent formats – in python or in XML. Both ways are detailed in this chapter. The next section presents the sample models used in this manual. Section 2.2 then reviews the main concepts of Entity-Relationship Modelling, then each element in a model are examined in details.

The two sections coming after describe how such models map onto, respectively, the python description, or PyModel (sect. 2.4), and the XML description (sect. 2.5).

Section 2.6 gives some answers to common design-related questions.

Last, section 2.7 presents the tools that help you design and verify the models, and derive from them db-schema and python code.

Some included tools may help you with the design and management of your models: the ZModelizationTool (for Zope) or, alternatively, a collection of command-line scripts. Thus, the easiest way to define a model would be to use the ZModelizationTool inside a Zope instance. However, you do not *need* a Zope instance to design models, as both PyModels and XML models may be designed in your favorite editor.

Both tools includes some validation logic, so that the common mistakes made when designing a model are identified.

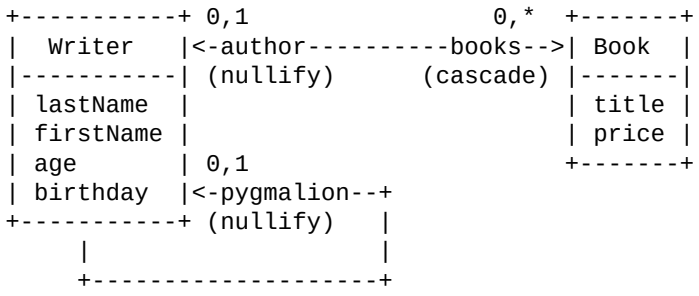
2.1 Sample models used in this manual

We expose here the two models that we will use as examples in this manual.

Both models ”AuthorBooks” and ”StoreEmployees” are models used in the test units shipped with the framework. You’ll find the associated packages in directory ‘Modeling/tests/testpackages’.

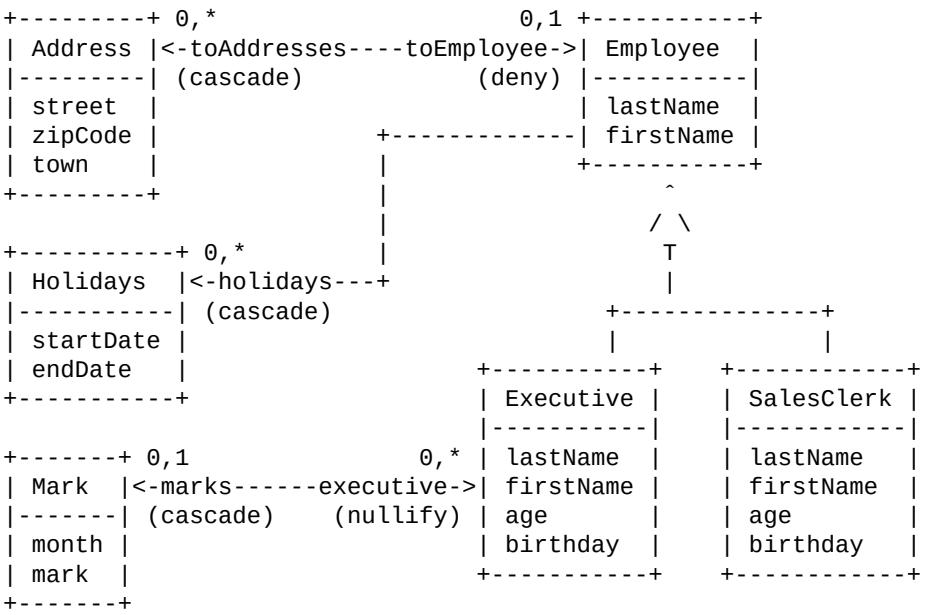
Notations: UML.

2.1.1 Model: AuthorBooks



Note: Notations *(cascade)* and *(nullify)* refer to the rule that should be applied when an object is deleted. Here, the deletion of a book removes its reference in its Writer's `books` relation, while the deletion of an author triggers the deletion of all its books in cascade. For a full explanation, please refer to section 2.3.4.

2.1.2 Model: StoreEmployees



2.2 Concepts

In this section we give a general idea on the Models and their elements. The details for the different parameters will be reviewed in detail in the next section.

E.-R. Modelling defines:

Entities: they map one-to-one with classes. An Entity holds all informations needed to make the instances persistent, including: Attributes, Relationships, and other informations (such as the name of the DB's table in which instances' values are ultimately stored)

Attributes: they can be of two kinds. Some map the attributes/fields of your classes to their column's name in their Entity's table, others have no existence at all at the object-level: they are artifacts, such as "primary keys" or "foreign keys", needed to design a relational database schema.

Relationships: relationships are, rougly speaking, half of an association. Relationships are defined in Entities. A Relationship is uni-directional, from the entity that holds it (the "source entity") to the "destination entity". A Relationship may have an inverse relationship. A relationship also holds its cardinality, and is said to belong to one of these two categories: **to-one** and **to-many** relationships. Let's illustrate this with a little example.

Say you have two entities, `Book` and `Writer` (this refers to the model definied in section 2.1.1); these two entities are in relation to each other: the `Book` defines a relationship named `toAuthor`, pointing to `Writer`, and `Writer` has in turn a relationship `toBooks` pointing to `Book`. Each of these relationships is the inverse of the other. The former, `toAuthor`, designates e.g. exactly 1 author, while the second one designates 0 or *n* books, with *n* being a positive integer.

In this case `1` is said to be both the lower and the upper bounds of the *toAuthor* relationship's multiplicity, while for *toBooks* `0` is the lower bound and `*` (meaning: any positive number of books) is the upper bound. The former is said to be a to-one relationship (1 is the upper bound), the latter, a to-many relationship (the multiplicity's upper-bound is –strictly– greater than 1).

Joins: Joins have no meaning at the object-level; you will not explicitly see or even define Joins in PyModels, or in the ZModelization tool interface –they are defined along with the relationship's definition. Joins are, again, artifacts defining **how** a relationship is made persistent into a database; more explicitly, a join has a source attribute and a destination attribute, both of which have no meaning at all at the object-level (they are not class' fields). Typically, a to-one relationship has Joins joining foreign keys to primary keys, while to-many relationships have joins whose source attributes are primary keys and destination attributes are foreign keys (see "Designing relationships", below, for details)

Additionally, we define Models and Model Sets.

A model groups entities together. It holds specific informations about the underlying database itself, such as the connection dictionary, and it gathers Entities which are to be made persistent within the same database.

Model Sets are, as the name suggest it, sets of models. The Framework itself only deals with one `ModelSet`, accessible by calling `defaultModelSet()` on module `Modeling.ModelSet`.

2.3 Full description

2.3.1 Model

A model element may have the following attributes:

`name`: The name of a model is just informative and is not used at run-time. It is however required. Note that models are intended to be loaded within a Model Set at runtime, and that a model set identifies its models by name. Hence, two models having the same name cannot be loaded at the same time within the framework.

adaptorName: the name of the database-adaptor, i.e. the back-end that you want to use for the entities in the model. Currently only `MySQL`, `Oracle` , `Postgresql` and `SQLite` (exact spelling) are valid `adaptorName`s (correspond respectively to the packages `MySQLAdaptorLayer`, `OracleAdaptorLayer`, `PostgresqlAdaptorLayer` and `SQLiteAdaptorLayer`).

packageName: the name of the package where your classes will be dropped. This will be used when you will ask for generation of code-skeletons for the whole model ; moreover, it implies that all classes mapped to the entities in a model must live within the same package.

Dotted-names are supported. This is used both when the code-templates are generated and at run-time, when objects of a certain type (entity) must be instanciated and populated with fetched values. The value stored in the model should therefore be kept in step with your current python installation.

See also: 2.3.2, Entity's properties `className` and `moduleName`, and B.1, FAQ.

comment: a comment, for maintainers of the model.

connectionDictionary: this may be the full connection dictionary, as required by the Adaptor chosen above, or a partial connection dictionary – in which case the equivalent keys will be read from the configuration file pointed to by the the enviroment variable *MDL_DB_CONNECTIONS_CFG* (see A.1).

A connection dictionary is always made of the following keys: `host`, `database`, `user`, and `password`. An additional key: `port` allows you to specify the port number to which the database server listens.

The adaptors in the framework make a different use of each of these keys to connect to a database. For example, the adaptor for SQLite only requires one key to be defined: `database`, pointing to the file containing the database. For an exact definition of how different adaptors use the connection dictionary, please refer to their respective documentation:

MySQL: MySQLAdaptorLayer (at ../API/Modeling-API/public/Modeling.DatabaseAdaptors.MySQLAdaptorLayer-module.html)

Oracle: OracleAdaptorLayer (at ../API/Modeling-API/public/Modeling.DatabaseAdaptors.OracleAdaptorLayer-module.html)

PostgresqlSQL: PostgresqlAdaptorLayer (at ../API/Modeling-API/public/Modeling.DatabaseAdaptors.PostgresqlAdaptorLayer-module.html)

SQLite: SQLiteAdaptorLayer (at ../API/Modeling-API/public/Modeling.DatabaseAdaptors.SQLiteAdaptorLayer-module.html)

2.3.2 Entity

An Entity corresponds to a Class in your (UML) model.

An entity may have the following attributes:

name: the name of the entity is required. It plays a fundamental role at run-time, making it possible for the framework to make classes correspond to DB tables and vice-versa.

className: this is the name of the class bound to the current entity. This is used both when the code-templates are generated and at run-time, when instances from a certain type are needed to be populated with fetched values. The value stored in the model should therefore be kept in sync. with your current python installation. See also: 2.3.1, Model's `'packageName'` property, `'moduleName'` (below), and B.1,"FAQ".

moduleName: this is the name of the module where the class is defined. This is used both when the code-templates are generated and at run-time, when objects a certain type needs to be instanciated and populated with fetched values. The value stored in the model should therefore be kept in step with your current python installation.

See also: 2.3.1, Model's `'packageName'` property, `'className'` (above), and B.1,"FAQ".

If it is not provided, it defaults to the `className`, since a class is often declared in a module which has the same name as its class.

externalName: this is the name of the underlying table in the database. A common habit consists in deriving it from the class name in a consistent way, for example, class `Book` becomes `externalName BOOK`, and `RegularCustomer` becomes `REGULAR_CUSTOMER`. (cf. `Modeling.Entity.externalNameForInternalName()` and its [1])

parentEntity: you can choose a parent entity (a *super class*) for your entity. Inheritance implies more than simply setting this tag, please refer to section 2.6.4 for a complete discussion.

primaryKeys: see the next subsection.

typeName, isAbstract, isReadOnly not used for now.

comment: a comment, for maintainers of the model.

Before detailing attributes and relationships, contained in entities, we first describe the special primary key attribute.

Primary keys

Primary keys are attributes/columns that an RDBMS uses to uniquely identify every row of a table. It is **required** that a non-abstract Entity[2] defines one of its attributes to be a primary key. Within an entity you define attributes, relationships, and primary keys as well as the attributes that are "used for locking" (this we will not explained for now, as it it addresses a feature, *optimistic locking*, that is not yet supported). However, for primary keys, you should remember that we only support **simple primary keys**, *i.e.* primary keys with only one attribute; hence, please make sure that you have one, and only one, primary-key per entity.

Entities know their primary keys. Their are defined differently in PyModels and in xml-model, see the respective sections **??** and 2.5.3.

2.3.3 Attribute

An attribute can be either a class field (and a database column), or simply just a database column.

You cannot have two attributes with the same name inside a single entity, but two entities can have distinct attributes with the same name. Similarly, you cannot have two attributes, two relationships or an attribute and a relationship that have the same name *within the same entity*, but it is okay if they each live in a different entity.

An attribute may have the following properties:

name: the name of the attribute

type: this is the associated python type, which can be either `string`, `int`[3] `float` or `DateTime` (mapping to egenix's `mxDateTime` –or to `DCOracle2.Timestamp`). You should keep the `externalType` (described below) and the python type in sync (python `string`: SQL `CHAR`, `VARCHAR` or `TEXT`[4], `int`: SQL `INTEGER`, etc.).

isClassProperty: indicates whether the Attribute correspond to a class' attribute. An example would be an `id` column serving as the primary key.

isRequired: indicates whether the attribute is **mandatory**, i.e. if it can have a value of `None`. If checked, this will be enforced both by the validation mechanism included in the framework AND by the database since the corresponding attribute will be created with a SQL statement indicating `NOT NULL`.

[1] as defined in module `Entity`, see documentation (at http://modeling.sf.net/API/Modeling-API/public/Modeling.Entity-module.html#externalNameForInternalName)

[2] abstract entities are not supported yet

[3] Note that python types `int` and `long int` are treated the same way. This is due to different DB-adaptors having different behaviours –some can also have a "non-deterministic" behaviour: sometimes returning `int`, sometimes `long int`, for the same db-field and the same value (e.g. 42); anyway it sounds reasonable to consider `int` and `long int` as equivalent within the framework's core.

[4] if you db-server supports `TEXT`

`columnName` the name of the column in the entity's DB table

`externalType`: the SQL type to which this attribute should be mapped. The supported SQL datatypes depend on the adaptor, please refer to their documentation for details:

- **MySQL:** MySQLAdaptorLayer (at http://modeling.sourceforge.net/API/Modeling-API/public/Modeling.DatabaseAdaptors.MySQLAdaptorLayer-module.html)
- **Oracle:** OracleAdaptorLayer (at http://modeling.sourceforge.net/API/Modeling-API/public/Modeling.DatabaseAdaptors.OracleAdaptorLayer-module.html)
- **PostgresqlSQL:** PostgresqlAdaptorLayer (at http://modeling.sourceforge.net/API/Modeling-API/public/Modeling.DatabaseAdaptors.PostgresqlAdaptorLayer-module.html)
- **SQLite:** SQLiteAdaptorLayer (at http://modeling.sourceforge.net/API/Modeling-API/public/Modeling.DatabaseAdaptors.SQLiteAdaptorLayer-module.html)

`precision`, `scale`, `width`: depending on the `externalType`, you may have to specify one or more of these elements to fully define the SQL datatype. Refer to any SQL reference manual for details (Postgresql 7.3 devel. documentation, chapter3: Data types is a good starting point).

Note: The field `width` is used at runtime when validating a string attribute's value (see 3.3, "Validation"), before objects are saved in the database. SQL data types `CHAR` and `VARCHAR` require that `width` is specified; on the other hand, `TEXT` –when supported by the database– does not accept that width is set. If you set it on an attribute, it will be ignored when the database schema is generated (i.e. a `TEXT` field my_text with width=30 won't be declared as `TEXT(30)`, but simply as `TEXT`), but it will be checked at runtime and validation for this attribute will fail if its value's length exceeds the given width. Note that the same goal can be achieved by writing a specific validation method for the attribute (see 3.3.2).

`defaultValue`: use this field to assign a default value to the class' attribute. Note that this is only used when python code is generated from a model. You enter it as a string, it will be converted to the correct type when the model is loaded.

Conversion details: an empty string converts to integer:`0`, float: `0.0`, string: `''`; special string value `'None'` evaluates to `None` for type string, and is invalid otherwise.

`displayLabel`: not used by the framework, but you can consider it valuable to fill it in and use it in your own applications (see also section 7 describing how a model can be introspected).

`comment`: a comment, for maintainers of the model.

`usedForLocking`: Unused yet, reserved field.

This flag will be used when optimistic locking is implemented. It will indicate which attributes optimistic locking should check when it is about to save changes: if value for attribute `lastname` has changed and `usedForLocking` is set for that attribute, then under the optimistic locking policy saveChanges() will raise; if the flag is unset, the attribute will be silently overriden (for example, you probably won't mark an object's timestamp as used for locking).

Primary keys

A primary key is an attribute that:

- is identified as a primary key in its entity (see 2.3.2),
- is required,
- is generally not defined as a class property (see B.1 for details).

Foreign keys

A foreign key is an attribute that:

- is used in a relationship's source or destination attributes (detailed in the next section)
- **should not** be defined as a class property (see B.1 for details).

2.3.4 Relationship

A relation is attached to an entity, and describes how an entity is related to another one.

Two kinds of relationships can be defined: **simple** and **flattened** relationship. However flattened relationshipsare not supported/tested yet.

Simple relationships join their (source) entity to a destination entity.

A relation may have the following properties:

`name`: the name of the relationship, unique within the entity.

`deleteRule`: this specifies how the object should react when it is deleted. Four options are possible options: `NULLIFY`, `DELETE_DENY`, `DELETE_CASCADE` and `DELETE_NOACTION`. We will use the model Author-Books defined in section 2.1.1, with entities `Writer` and `Books`, to highlight the differences between those 4 values:

- if `Book`'s relationship *toAuthor* has the delete rule `DELETE_NULLIFY`, then when a book is deleted it is automatically removed from the corresponding author's *toBooks* relationship.
- If `Author`'s relationship *toBooks* has the delete rule `DELETE_DENY`, then if an author is marked for deletion while it is still in relation with books, the deletion will be denied when you try to make it persistent.
- If `Author`'s relationship *toBooks* has the delete rule `DELETE_CASCADE`, then the deletion of an author will automatically delete the related books as well.
- You can also instructs the framework not to take any actions when an object is deleted, with respect to its relationships. In this case, you use the delete rule `DELETE_NOACTION`. This is however quite an advanced setting and should be used with great care, since inappropriate use of this setting can result in dangling references in your databases.

This will be enforced when an object is inserted/updated or when it is deleted. See sections 4.4 and 4.6 for further details.

`isClassProperty`: tells whether the Relationship's key (i.e. its name, for example: *toBooks*) is part of the class' API. It should be set in most cases.

`multiplicityLowerBound`, `multiplicityUpperBound`: the multiplicty lower bound of a relationship defines the minimum number of objects that should be in relation. If the lower bound is zero, none are required: this is an *optional* relationship; if the lower bound is one or more, this is a *mandatory* relationship.

The multiplicity upper bound defines the maximum allowed number of objects in relations. It can be any positive number, given that it is always greater or equal than the lower bound. As a special exception, values `-1` (minus one) or `'*'` serve to indicate an unconstrained upper bound.

The exact definition of to-one and to-many relationships (see 2.2) are then: to-one relationship: `multiplicityUpperBound<=1`, to-many relationship: `multiplicityUpperBound>1`.

`destinationEntity`: designates the name of the destination entity.

`joinSemantic`: the join's sematic can be either:

- (default) Inner join

- Full outer join
- Left outer join
- Right outer join

`displayLabel`: same as for Attributes, above.

`comment`: a comment, for maintainers of the model.

The full definition of a relationship implies that it defines at least one *join*. However, this will not appear in PyModels, just in xml models. It's not really part of Entity-Relationship Modelling, it is just a mean to identify the correspondance between primary and foreign keys that defines a relationship. For this reasons, we will wait until section 2.5.5 to define them.

See also:

- section 2.6.2, which describes how relationships should be designed, and why.
- section 2.3.3, which discusses namespace issues for relationships and attributes.

2.4 PyModels

Now that we know the different components of a model, it is time to examine how they can be practically defined. This section examines a python definition of a model, and the next section will show how this can be done equivalently in a xml-file.

A PyModel is a python module that defines a variable *model* that is of type `PyModel.Model`. Objects of type `Model` collect all information about their entities, as a list of objects of type `Entity`. Similarly, entities collect objects defining attributes and relationships. A *model* instance may also define bi-directional relationships as associations. The python classes for each of these is described below. To reduce boilerplate code, PyModels allow defaults to be specified for all values. Furthermore, sub-types for frequently used constructs are provided, and others may of course be defined as necessary.

2.4.1 Organization of this chapter

The next section, 2.4.2, will give a full example of a PyModel.

Section **??** will then discuss defaults: what they are, and how they will be changed. It should be read before referring to the components' details, since it explains important things that won't be repeated afterwards.

Then, for each component of a model (`Model`, `Entity`, etc.), a comprehensive list of available properties will be given, including their default values. These properties are those we saw in section 2.3. We suppose here that the content of this section is known, so if you need details about the meaning and effects of some component's property, please refer to the component's detailed description, above.

Last, please note that the section describing Models (2.4.5) exposes in details how these properties can be set; this information will not be repeated in the sections coming afterwards.

2.4.2 A sample PyModel

Before dealing with all the details, here is an overview of a PyModel. Note the the *id* default property for entities, as well as the predefined `Attribute` sub-types, `APrimaryKey` and `AString`.

```
from Modeling.PyModel import *

# Set preferred defaults for this model (when different from
# standard defaults, or if we want to make things explicit)

AFloat.defaults['precision'] = 10
AFloat.defaults['scale'] = 2
AString.defaults['width'] = 40

Association.defaults['delete']=['nullify', 'nullify']

Entity.defaults['properties'] = [
  APrimaryKey('id', isClassProperty=0, isRequired=1, doc='PK')
]

_connDict = {'database': 'AUTHOR_BOOKS'}
model = Model('AuthorBooks',adaptorName='Postgresql', connDict=_connDict)
model.version='0.1'
model.entities = [
  #
  Entity('Book',
         properties=[ AString('title', isRequired=1, columnName='title'),
                      AFloat('price'),
],
         ),
  Entity('Writer',
         properties=[ AString('lastName',isRequired=1, width=30,
                              displayLabel='Last Name', ),
                      AString('firstName', displayLabel='First Name', ),
                      AInteger('age', displayLabel='Age', ),
                      ADateTime('birthday', usedForLocking=0,
                                displayLabel='birthday', ),
                      ]
         ),
  ]
model.associations=[
  Association('Book', 'Writer',
              relations=['author', 'books'],
              delete=['nullify', 'cascade']),
  Association('Writer', 'Writer',
              relations=['pygmalion', None],
              delete=['nullify', None]),
  ]
```

A PyModel is typically made of two parts: a first part sets the defaults, then the second one defines the model. Defaults helps you keep the model definition tidy, by removing the repetitive parts from the model definition itself; in this example, the defaults add a primary key to each entity, sets the `width` for strings, etc.

2.4.3 Defaults

In the coming description of the components of a model, you'll see that every possible property is assigned a default value.

This default value will be used if you omit it in your definition. For example, a component `AString` representing a string attribute has a default value of `255` assigned to its `width`. This default value may, or may be not, what you need. To avoid the unnecessary overhead of overwriting the defaults in each component `AString`, you can set a different default for one, or more, of its properties:

```
AString.defaults['width']=30
AString.defaults['externalType']='TEXT'
```

This is a general mechanism that you can use for every single property defined by a component, simply by changing the corresponding entry in the component class's dictionary `defaults`.

2.4.4 Model

A model defines the following properties:

Prop.	Type	Default	Comment
`name`	`string`	*no default*	The `name` is the only mandatory arguments when instanciating a `Model`. Once set, it should not be changed
`packageName`	`string`	value of `name`	Dynamically
`adaptorName`	`string`	`''`	
`connDict`	`dict`	`{'host': '', 'database': '', 'user': '', 'password': '' }`	
`entities`	sequence	`[]`	Entities for the model are appended to this sequence.
`associations`	sequence	`[]`	Associations are appended to this sequence.
`doc`	`string`	`''`	this is the "comment" field
`version`	`string`	*no default*	currently `'0.1'` –see below

Note: For details about these properties and their meaning, please refer to 2.3.1.

When instanciated, a model should supply its name:

```
model = Model('AuthorBooks')
```

Properties can be set at instanciation time:

```
model = Model('AuthorBooks', adaptorName='Postgresql')
```

or by direct assignment:

```
model.adaptorName='Postgresql'
```

Model's `version`

A special property, `version`, should be set in every PyModel, and it should be equal to `PyModel.Model.VERSION`, currently `'0.1'` (a string value).

Its purpose is to track changes in the PyModel mechanisms, especially to warn you when some defaults have changed. When this happens, the file 'MIGRATION' shipped along with the source distribution, and located in the top-level directory of the distribution, this file will contain details about the changes, such as: the defaults that should be checked, how to update your existing PyModels, etc.

How does this work? When a PyModel is interpreted, its `version` is checked; if they do not correspond, you will get an exception `PyModel.IncompatibleVersionError` (a subclass of `RuntimeError`).

We highly recommend that you use the literate string value of `version` and that you do not directly assign `PyModel.Model.VERSION` to it; you'll supply it as a property, like in:

```
>>> model=PyModel.Model('ModelName')
>>> model.version='0.1'
```

or:

```
>>> model=PyModel.Model('<ModelName>', version='0.1')
```

This way, you are sure that you'll be warned when the default behaviour of PyModels changes.

2.4.5 Entity

The component `Entity` defines the following properties:

Prop.	Type	Default	Comment
`name`	`string`	*no default*	The `name` is the only mandatory arguments when instanciating a `Entity`. Once set, it should not be changed
`className`	`string`	value of `name`	
`moduleName`	`string`	value of `name`	
`externalName`	`string`	`externalNameForInternalName(name)`	
`isAbstract`	`int`	`0`	
`isReadOnly`	`int`	`0`	
`properties`	sequence	`[]`	Sequence of `Attribute` and `Relationship` objects
`parent`	`string`	`''`	The `name` of its parent entity (inheritance)
`typeName`	`string`	`''`	
`doc`	`string`	`''`	comment

Note: For details about these properties and their meaning, please refer to 2.3.2.

2.4.6 Attribute

Prop.	Type	Default	Comment
`name`	`string`	*no default*	The `name` is the only mandatory arguments when instanciating a `Attribute`. Once set, it should not be changed

Prop.	Type	Default	Comment
columnName	string	externalNameForInternalName(name) [5]	
type	string	'int'	
externalType	string	'INTEGER'	
isClassProperty	int	1	
isRequired	int	0	
precision	int	0	
scale	int	0	
width	int	0	
defaultValue		None	
usedForLocking	int	0	
displayLabel	string	''	
doc	string	''	comment

Note: For details about these properties and their meaning, please refer to 2.3.3.

Every attribute can be declared as a plain `Attribute` object. However, the framework provides convenience subclasses for standard attributes:

- `ADateTime` for mapping dates,
- `AFloat` for mapping floating-point numbers,
- `AInteger` for mapping integers,
- and `AString` for mapping strings.

We now examines those subclasses and their defaults.

ADateTime

The component `ADateTime` redefines the following defaults:

Prop.	Type	Default	Comment
name	string	*no default*	The `name` is the only mandatory arguments when instanciating a `ADateTime`. Once set, it should not be changed
type	string	'DateTime'	
externalType	string	TIMESTAMP	Be warned: not all database supports `TIMESTAMP`. If this is the case, you'll probably have to redefine this default value in your PyModel. Discussion on supported SQL datatypes can be found at section 2.3.3.
defaultValue		None	
usedForLocking	int	1	

[5] as defined in module `Entity`, see externalNameForInternalName (at http://modeling.sf.net/API/Modeling-API/public/Modeling.Entity-module.html#externalNameForInternalName)

Note: For details about these properties and their meaning, please refer to 2.3.3.

AFloat

The component `AFloat` redefines the following defaults:

Prop.	Type	Default	Comment
`name`	`string`	*no default*	The `name` is the only mandatory arguments when instanciating a `AFloat`. Once set, it should not be changed
`type`	`string`	`'string'`	
`externalType`	`string`	`'NUMERIC'`	
`precision`	`int`	`15`	
`scale`	`int`	`5`	
`defaultValue`		`0.0`	
`usedForLocking`	`int`	`1`	

Note: For details about these properties and their meaning, please refer to 2.3.3.

Of course, you can redefine the defaults for `AFloat` in your PyModel to fit your needs.

AInteger

The component `AFloat` redefines the following defaults:

Prop.	Type	Default	Comment
`name`	`string`	*no default*	The `name` is the only mandatory arguments when instanciating a `AInteger`. Once set, it should not be changed
`type`	`string`	`'int'`	
`externalType`	`string`	`'INTEGER'`	
`defaultValue`		`0`	
`usedForLocking`	`int`	`1`	

Note: For details about these properties and their meaning, please refer to 2.3.3.

AString

The component `AString` redefines the following defaults:

Prop.	Type	Default	Comment
name	string	*no default*	The name is the only mandatory arguments when instanciating a AString. Once set, it should not be changed
type	string	'string'	No matter whether you run python2.1 (where type('').__name__=='string') or python2.2 (where type('').__name__=='str'), you should always use 'string' for a character type.
externalType	string	'VARCHAR'	
width	int	255	
defaultValue		''	
usedForLocking	int	1	

Note: For details about these properties and their meaning, please refer to 2.3.3. Of course, you can redefine the defaults for AString in your PyModel to fit your needs. For example, if you prefer to use TEXT, you'll probably add this to your PyModel:

```
AString.defaults['externalType'] = 'TEXT'
AString.defaults['width'] = 0
```

APrimaryKey

The component APrimaryKey redefines the following defaults:

Prop.	Type	Default	Comment
name	string	*no default*	The name is the only mandatory arguments when instanciating a APrimaryKey. Once set, it should not be changed
isClassProperty	int	0	
isRequired	int	1	You should not change this default, nor overwrite it: a primary key should be mandatory.
defaultValue		None if isClassProperty is false, 0 otherwise	
doc	string	'Primary Key'	
usedForLocking	int	0	

Note: For details about these properties and their meaning, please refer to 2.3.3.

Warning: If you want to make your primary keys class properties, please be sure to read the dedicated "FAQ" entry (B.1).

Every entity must define a primary key. However, most of the times, you do not want to explicitly declare a primary key in each of your entities, since they'll basically be the same. In this case, you'll simply add a primary key to the `Entity`'s defaults:

```
Entity.defaults['properties'] = [
  APrimaryKey('id', isClassProperty=0, isRequired=1, doc='PK')
]
```

Every declared entity will then automatically get its own primary key –a clone of the default one.

AForeignKey

The component `AFloat` redefines the following defaults:

Prop.	Type	Default	Comment
`name`	`string`	*no default*	The `name` is the only mandatory arguments when instanciating a `AForeignKey`. Once set, it should not be changed
`isClassProperty`	`int`	`0`	
`isRequired`	`int`	`0`	
`defaultValue`		`None`	
`doc`	`string`	`'Foreign Key'`	
`usedForLocking`	`int`	`0`	

Note: For details about these properties and their meaning, please refer to 2.3.3.

Warning: Foreign keys should not be marked as class properties. For a full discussion on this topic, please refer to the dedicated "FAQ" entry (B.1).

2.4.7 Relationship

Relationships can be declared in two different manners:

- you can declare `RToOne` and `RToMany` relationships, joining an entity to an other one. Each of these objects defines a directional relationship, from a source entity (this is the entity where the relationship is defined) to a destination entity (pointed to by the object's field `destination`, see below).
- Or, instead of declaring two directional relationships, each being the inverse of the other, you can also declare a object `Association` which gives you the opportunity to declare a relationship and its inverse in a signle python statement.

In the next section, we'll see the defaults of `BaseRelationship`; while it's not a pymodel component *per se*, it defines the defaults for all other component defining relationships: `RToOne` (cf.2.4.7), `RToMany` (cf.2.4.7) and `Association` (cf.2.4.8).

BaseRelationship

A `BaseRelationship` describes how two entities relate to each other. It has the following attributes:

Prop.	Type	Default	Comment
`name`	`string`	*no default*	The `name` is the first mandatory argument when instanciating a `BaseRelationship`. Once set, it should not be changed
`destination`	`string`	*no default*	The destination entity's `name` is the second mandatory argument when instanciating a `BaseRelationship`. Once set, it should not be changed
`delete`	`string`	`'nullify'`	
`isClassProperty`	`int`	1	
`multiplicity`	sequence	`[0,1]`	
`joinSemantic`	`int`	0	see below
`src`	`string`	`''`	source attr.'s name
`dst`	`string`	`''`	destination attr.'s names
`displayLabel`	`string`	`''`	
`doc`	`string`	`''`	
`inverse`	`string`	`''`	a valid relationship's name in the destination entity –see 2.4.7, "Inverse relationships" for a full discussion.

You'll never need to use a `BaseRelationship`; in fact, it's not even legal in a PyModel. Instead, you'll declare `RToOne` (2.4.7) and `RToMany` (2.4.7) objects; or even better, you'll use `Association` (2.4.8) objects to create both a relationship and its inverse in a single declaration.

We presented it here because all three element `RToOne`, `RToMany` and `Association` use the `BaseRelationship`'s defaults for their own defaults (and override some of them).

Join semantic

The possible values for the attribute `joinSemantic`, and their respective meaning are:

- `0`: Inner join
- `1`: Full outer join
- `2`: Left outer join
- `3`: Right outer join

RToOne

`RToOne` objects describe a to-one relationship from an `Entity` to another. It derives from `Relationship` and overrides the following defaults:

Prop.	Type	Default	Comment
`name`	`string`	*no default*	The `name` is the only mandatory arguments when instanciating a `RToOne`. Once set, it should not be changed
`multiplicity`	sequence	`[0,1]`	
`joinSemantic`	`int`	`0`	see 2.4.7 for possible values

(All other defaults are `BaseRelationship`'s ones, cf.2.4.7)

Minimally, a `RToOne` needs a `name` and a the name of the destination entity, `destination`.

Source and destination attributes

Attributes `src` and `dst`, identifying source and destination attributes, are automatically calculated if they are not supplied:

- `dst` is the primary key of the destination entity identified by its name in attribute `destination`.
- `src` is calculated from the destination entity's name stored in the relationship's `destination` attribute. It is a string like: `'fk<sourceEntityName>'`, possibly followed by a integer (such as in `'fkEmployee1'`) if the name already exists in the destination entity. In fact, a PyModel does more than just computing a name: it also automatically creates the corresponding foreign key in the source entity.

  For example, a pymodel containing:

```
self.model.entities = [
  Entity('Employee',
         properties=[ RToOne('toStore', 'Store'),
  # ...
  ]
```

  automatically binds `dst` to the destination entity `'Store'` 's primary key, and creates a `AForeignKey` named `'fkStore'` in the source entity `'Employee'` (unless such a property –either an attribute or a relationship– already exists with this name, in which case it uses the first unused name among `'fkStore1'`, `'fkStore2'`, etc.

  Last, the `'inverse'` field of a `RToOne` (which designates the inverse relationship defined in the destination entity) has some effect in the automatic generation of `AForeignKey`: please refer to 2.4.7 for a full discussion on this topic.

Of course, you can specify your own source and destination attributes. In this case, it is requires that both are supplied, and that they corrspond to attributes (resp. `AForeignKey` and `APrimaryKey` attributes) explicitly declared in the source/destination entities.

RToMany

`RToOne` objects describe a to-many relationship from an `Entity` to another. It derives from `Relationship` and overrides the following defaults:

Prop.	Type	Default	Comment
`name`	`string`	*no default*	The `name` is the only mandatory arguments when instanciating a `APrimaryKey`. Once set, it should not be changed
`multiplicity`	sequence	`[0, None]`	`None` means *unconstrained*: no upper limit
`joinSemantic`	`int`	`0`	see 2.4.7 for possible values

(All other defaults are `BaseRelationship`'s ones, cf.2.4.7)

Minimally, a `RToMany` needs a `name` and a the name of the destination entity, `destination`.

Source and destination attributes

Attributes `src` and `dst`, identifying source and destination attributes, are automatically calculated if they are not supplied. The rules are the same than the ones given above for `RToOne`, you just need to swap name `'src'`/"source" and `'dst'`/"destination" in the above explanation.

As an example, suppose you have a pymodel declaring such a `RToMany`:

```
self.model.entities = [
  Entity('Store',
         properties=[RToMany('toEmployees','Employee')]
         ),
  # ...
  ]
```

`RToMany` then automatically binds `'src'` to the source entity `'Store'` 's primary key, and creates a `AForeignKey` named `'fkStore'` in the destination entity `'Employee'` (unless such a property –either an attribute or a relationship– already exists with this name, in which case it uses the first unused name among `'fkStore1'`, `'fkStore2'`, etc.

You'll also want to read the section 2.4.7 for a complete explanation on how automatic binding/generation of `APrimaryKey`/`AForeignKey` is handled when two relationships are inverse of each other.

Inverse relationships

Most of the times, a `RToOne` relationship is the inverse of a `RToMany`. That's why they both have a field `inverse`, which allows you to identify the inverse relationship.

When you define both `RToOne` and `RToMany` with *explicit* source- and destination attributes (using the fields, resp., `src` and `dst`), a PyModel does not need this information to know that the two relationships are inverse for each other: a simple inspection of the relationships makes it noticeable that their source/destination entity and attributes are the same, which allows it to deduce that they are inverse to each other.

However, when you use the automatic and *implicit* declaration of source and destination attributes, you must supply the `inverse` keyword, otherwise you won't get what you expect. Suppose you declare something like this:

```
self.model.entities = [
  Entity('Employee',
         properties=[ RToOne('toStore', 'Store'),
                    ] ),
   Entity('Store',
         properties=[RToMany('toEmployees','Employee'),
                    ] ),
   ]
```

Now see what happens (we suppose here that you have read how automatic binding of source and destination attributes is done, as described in 2.4.7):

1. the PyModel examine the first `RToOne`; given that neither `src` nor `dst` are supplied, its destination attribute is automatically bound to the `Store`'s primary key, and a `AForeignKey` is created, named `'fkStore'`, and assigned to the source attribute.
2. Now the PyModel examine the other `RToMany` relationship. The source attribute is automatically bound to `Store`'s primary key. What about the destination attribute? As expected, a foreign key should be created then bound; but since a attribute `'fkStore'` already exists (created at the previous step), a foreign key named `'fkStore1'` is created and bound to the destination attribute.

So: two foreign keys were created in entity `Store`, one for each relationship defined. As a consequence, and since the two relationships use their own foreign key, they cannot be considered as inverse to each other.

This is why you must supply the `inverse` attribute when designing a relationship and its inverse. When it is supplied, the automatic generation of foreign key detects it and, rather than re-creating an other foreign key such as above in step 2., re-uses the foreign key that was previously created in step 1. Hence, the following declaration is correct:

```
self.model.entities = [
  Entity('Employee',
         properties=[ RToOne('toStore', 'Store', inverse='toEmployees'),
                    ] ),
   Entity('Store',
         properties=[RToMany('toEmployees','Employee',inverse='toStore'),
                    ] ),
   ]
```

Note: It is not required that both relationships defines the `inverse` attribute: it is sufficient to declare it in one of the two relationships (either the `RToOne` or the `RToMany`). However and as a general rule, we think that it makes a pymodel clearer if you define it in both relationships, and we suggest that you do that.

2.4.8 Association

`Associations` are a pratical shortvut for defining a relationship and its inverse in a single python statement.

Suppose that we want to design the two relationships already discussed above between entities `Employee` and `Store`:

```
Employee <<-toEmployees------toStore-> Store
```

We could define the appropriate `RToOne` and `RToMany` objects in their respective entities. Now we can also define them like this:

```
model.entities = [ Entity('Employee'), Entity('Store') ]
model.associations = [
  Association('Employee', 'Store'),
  ]
```

(We've only left in the example the necessary declarations for demonstration –the full pymodel is exposed in 2.4.2).

This automatically creates the two relationships, along with the necessay foreign key.

IMPORTANT: `Association` objects *always* define a to-one association from the first entity to the second entity, and an inverse to-many relationship from the second entity to the first one.

Here is an equivalent declaration, where some of the defaults are explictly exposed:

```
Association('Employee', 'Store',
            relations=['toStore', 'toEmployees'],
            multiplicity=[ [0, 1], [0, None] ],
            delete=['nullify', 'deny'])
```

Here again and as a general rule, we suggest that you provide at least the names and the multiplicity of both relationships, so that it is clear which one is the to-one/to-many relationship.

Now here are the defaults that `Association` uses. As you see, it uses the same defaults then `RToOne` and `RToMany`:

Prop.	Type	Default	Comment
`src`	`string`	*no default*	The source entity's name. This parameter is mandatory when creating a `Association`
`dst`	`string`	*no default*	The destination entity's name. This parameter is mandatory when creating a `Association`
`multiplicity`	sequence	`[ [0,1], [0,None] ]`	The multiplicity for each rel.
`relations`	sequence	`['to<Dst>', 'to<Src>s']`	The names for the relationships
`keys`	`sequence`	`[None, None]`	The two attributes'names that both relationships refer to as source/destination attributes
`delete`	sequence	`[ RToOne.defaults['delete'], RToMany.defaults['delete'] ]`	the delete rule for each rel.
`isClassProperty`	sequence	`[ RToOne.defaults['isClassProperty'], RToMany.defaults['isClassProperty'] ]`	Whether each rel. is a class property
`joinSemantic`	sequence	`[ RToOne.defaults['joinSemantic'], RToMany.defaults['joinSemantic'] ]`	The join semantic for each rel.
`displayLabel`	sequence	`[ RToOne.defaults['displayLabel'], RToMany.defaults['displayLabel'] ]`	the `displayLabel` for each rel.
`doc`	sequence	`[ RToOne.defaults['doc'], RToMany.defaults['doc'] ]`	A comment assigned to each rel.

Last, an `Association` can be used to define a directional association (where only one of the two relationships is defined), by setting one of the `relations` to `None`, such as in:

```
Association('Writer', 'Writer',
            relations=['pygmalion', None],
            delete=['nullify', None],
            keys=['FK_Writer_id', 'id']),
```

(extracted from 'Modeling/tests/testPackages/AuthorBooks/pymodel_AuthorBooks.py', corresponding to the model defined in 2.1.1)

2.5 XML model

2.5.1 Overview of the xml

Before going into details, here is the skeleton of an xml-model:

```
<?xml version='1.0' encoding='iso-8859-1'?>
<model ...>
  <entity ...>
    <primaryKey attributeName='...'/>
    <attribute .../>
    <relation ...>
      <join .../>
    </relation>
  </entity>
  <entity ...>
    ...
  </entity>
</model>
```

A complete template is provided at the end of the chapter, in section 2.5.6.

Each of the xml elements `<model>`, `<entity>` etc. has its own set of attributes, as indicated by the ellipses found in the xml elements above. The following subsections describe each element in detail.

Note: every value associated to a xml attribute should be a string. It is automatically converted to the right type when the model is loaded.

2.5.2 Model

We already know that an xml-model holds one and only one model; `<model>` is, hence, the root element of the model.

The `<model>` element has the following attributes:

```
<model
    name        = ''             -- unique per model set
    packageName = ''             -- generate classes into, and import from, this package
    adaptorName = ( MySQL | Oracle | Postgresql | SQLite ) : Postgresql    -- a supported database adaptor
    comment = ''                 -- a comment
    connectionDictionary = "{}"  -- string representation of a python dictionary
>
```

All these attributes are described in section 2.3.1.

The connection dictionary should be a valid python expression for dictionaries; when other database adaptors come up for other database, they will be required to accept the very same set of required keys, namely `host`, `database`, `user`, and `password`.

2.5.3 Entity

An Entity corresponds to a Class in your (UML) model.

Before detailing attributes and relationships, contained in entities, we first describe the special primary key attribute.

Primary keys

You define a primary key like this:

```
<entity ...>
  <primaryKey attributeName='...'/>
  ...
</entity>
```

The `<primaryKey>` tag only has one attribute, `attributeName`, which needs to have the same value as one the entity's attributes' names. We will see how an attribute is described in the next section, however we can already tell that an attribute used as a primary key is often defined like:

```
<attribute name='id' type='int'
           isClassProperty='0' isRequired='1'
           columnName='ID' externalType='INTEGER'/>
```

Definition of an entity

The `<entity>` element is a child of the `<model>` element. It has the following attributes:

```
<entity
    name            = ''   -- relates class to db table
    className       = ''   -- python class name for this entity
    moduleName      = ''   -- class is generated into, and accessed from, this module
    externalName    = ''   -- db table name
    parentEntity    = ''   -- name of entity to "inherit" from
    typeName        = ''   -- not yet used
    isAbstract      = '0'  -- not yet used
    isReadOnly      = '0'  -- not yet used
    comment         = ''   -- a comment
>
```

All these attributes are described in section 2.3.2.

2.5.4 Attribute

The `<attribute>` element is a child of the `<entity>` element. It may have the following attributes:

```
<attribute
    name            = ''   -- class attribute name
    type = ( string | int | float | DateTime ) : string  -- python type
    isClassProperty = '1'  -- '0' or '1'; attribute is also a class property
    isRequired      = '1'  -- '0' or '1'; cannot be null
    columnName      = ''   -- database table column name
    externalType    = ''   -- database type
    width           = ''   -- qualifier for some values of externalType
    precision       = ''   -- qualifier for some values of externalType
    scale           = ''   -- qualifier for some values of externalType
    defaultValue    = ''   -- class attribute default value
    displayLabel    = ''   -- text label to use in applications
    comment         = ''   -- a comment
/>
```

All these attributes are described in section 2.3.3.

2.5.5 Relationship

The <relationship> element is a child of the <entity> element. It has the following attributes:

```
<relation
    name                    = ''   -- relation name
    deleteRule = ( 0,DELETE_NULLIFY | 1,DELETE_DENY | 2,DELETE_CASCADE |
                   3,DELETE_NOACTION ) : 0  -- behaviour when object is deleted
    isClassProperty         = '1'  -- '0' or '1'; relation is also a class property
    multiplicityLowerBound = ( int > -1 ) : 0
    multiplicityUpperBound = ( int > -1 | -1,* ) : 1
                                   -- -1 or * indicate unconstrained upper bound
    destinationEntity       = ''   -- name of destination entity
    joinSemantic = ( 0,Inner | 1,FullOuter | 2,LeftOuter | 3,Right Outer ) : 0
    displayLabel             = ''  -- text label to use in applications
    comment                  = ''  -- a comment
>
</relation>
```

All these attributes are described in section 2.3.4.

Here are the possible values for these attributes, and their meaning:

`isClassProperty`:
- code'0': `DELETE NULLIFY`
- `'1'`: `DELETE DENY`
- `'2'`: `DELETE CASCADE`
- `'3'`: `DELETE NOACTION`

Note: in future it will be possible to enter the real names (such as `DELETE_CASCADE`), not only their numerical value (defined in module `ClassDescription`.

`isClassProperty`: Default value: `'1'` (*Yes*). `'0'` stands for *No*.

`multiplicityLowerBound`, `multiplicityUpperBound`: You can enter `'*'` or `'-1'` for an unconstrained upper bound.

Default values: `'0'` for lower bound, `'1'` for upper bound.

`joinSemantic`: Possible values and their respective meaning are:

- `'0'`: (default) Inner join
- `'1'`: Full outer join
- `'2'`: Left outer join
- `'3'`: Right outer join

(as defined in module `Relationship`)

The full definition of a relationship implies that it defines at least one *join*.

See also:

- section 2.6.2, which describes how relationships should be designed, and why.
- section 2.3.3, which discusses namespace issues for relationships and attributes.

Joins

A join has no equivalent in the object model. In fact, it is one the rare elements (such as the properties defined by tags `<primaryKey>` and `<attributesUsedForLocking>`) that only describe the underlying database schema. A full discussion on how relationships should be modeled can be found in the next section. For the moment, we will only describe the xml element `<join>`.

We already know the every relationship should have at least one join. The exact definition of a join goes like this:

```
<relation ...>
  <join sourceAttribute='' destinationAttribute=''/>
</relation>
```

The `<join>` element requires the following two attributes:

`sourceAttribute`: the name of an attribute belonging to the enclosing relationship's entity.

`destinationAttribute`: the name of an attribute belonging to the destination entity pointed by the relationship, i.e. the entity given by `"../relation/@destinationEntity"`.

It is formatted this way:

```
<relation
    name                  = ''   -- relation name
    ...
    destinationEntity     = ''   -- name of destination entity
>
  <!-- unordered content: (join) -->
  <join
      sourceAttribute = ''      -- name of source attribute, in enclosing entity
      destinationAttribute = '' -- name of target attribute, in ../@destinationEntity
  />
</relation>
```

2.5.6 Full format of an xml-model

```
<?xml version='1.0' encoding='iso-8859-1'?>
<model
    name        = ''             -- unique per model set
    packageName = ''             -- generate classes into, and import from, this package
    adaptorName = ( MySQL | Oracle | Postgresql | SQLite ) : Postgresql    -- a supported database adaptor
    comment = ''                 -- a comment
    connectionDictionary = "{}"  -- string representation of a python dictionary
>
<!-- unordered content: (entity*, relation*) -->
    <entity
        name            = ''   -- relates class to db table
        className       = ''   -- python class name for this entity
        moduleName      = ''   -- class is generated into, and accessed from, this module
        externalName    = ''   -- db table name
        parentEntity    = ''   -- name of entity to "inherit" from
        typeName        = ''   -- not yet used
        isAbstract      = '0'  -- not yet used
        isReadOnly      = '0'  -- not yet used
        comment         = ''   -- a comment
    >
    <!-- unordered content: (primaryKey, attributesUsedForLocking*, attribute*, relation*) -->
        <primaryKey
            attributeName = ''     -- name of an attribute in this entity
        />
        <attributesUsedForLocking
            attributeName = ''     -- name of an attribute in this entity
        />
        <attribute
            name            = ''   -- class attribute name
            type = ( string | int | float | DateTime ) : string  -- python type
            isClassProperty = '1'  -- attribute is also a class property
            isRequired      = '1'  -- cannot be null
            columnName      = ''   -- database table column name
            externalType    = ''   -- database type
            width           = ''   -- qualifier for some values of externalType
            precision       = ''   -- qualifier for some values of externalType
            scale           = ''   -- qualifier for some values of externalType
            defaultValue    = ''   -- class attribute default value
            displayLabel    = ''   -- text label to use in applications
            comment         = ''   -- a comment
        />
        <relation
            name                    = ''   -- relation name
            deleteRule = ( 0,DELETE_NULLIFY | 1,DELETE_DENY | 2,DELETE_CASCADE |
                           3,DELETE_NOACTION ) : 0  -- behaviour when object is deleted
            isClassProperty         = '1'  -- relation is also a class property
            multiplicityLowerBound = ( int > -1 ) : 0
            multiplicityUpperBound = ( int > -1 | -1,* ) : 1
                                        -- -1 or * indicate unconstrained upper bound
            destinationEntity       = ''   -- name of destination entity
            joinSemantic = ( 0,Inner | 1,FullOuter | 2,LeftOuter | 3,Right Outer ) : 0
            displayLabel             = ''  -- text label to use in applications
            comment                  = ''  -- a comment
        >
        <!-- unordered content: (join) -->
            <join
                sourceAttribute = ''      -- name of source attribute, in enclosing entity
                destinationAttribute = '' -- name of target attribute, in ../@destinationEntity
            />
        </relation>
    </entity>
</model>
```

2.6 General guidelines and gotchas

In this section we cover some general design questions.

2.6.1 Simple models (no inheritance)

If your UML/object model has no inheritance hierarchy, the modelization of the mapping between the object world and the relational world is quite straightforward.

Mapping classes to entities You first need to create one entity per class. You then populate each entity with the attributes and relationships –see 2.6.2– that are in your UML model.

Each of these entity needs to have a *primary key*. A primary key is typically an attribute which is **not** a class property, that is required, whose python type is `integer` and SQL type (the so-called *external type*) `INTEGER`. A primary key is usually named `id` but this is ultimately up to you. Primary keys should not be made class properties, and as a consequence they won't be accessible in the python object ; this is intended and should not be changed (see also: B.1, "FAQ").

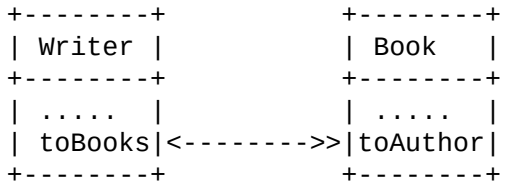
For regular attributes and their properties, see 2.3.3. Note that there is no way to map a class attribute: only instance attributes can be used here.

For relationships, you will need to have your primary keys and some *foreign keys* prepared: section 2.6.2 gives the details of their definition.

2.6.2 Designing relationships

A relationship and its inverse relationship (when defined) define an association. If a relationship has no inverse, the corresponding association is said to be uni-directional, otherwise it is bi-directional. Associations can be of the three kinds, with the following constraints:

One-to-many this is the most common case. To define a one-to-many relationship between the entities `Writer` and `Book`, like this:



(extracted from the full model defined at section 2.1.1) you need to perform the following steps:

1. Both entities should (already) have a primary key defined, e.g. `id` (see 2.6.1). The `id` will use the python type `int` and the SQL type `INTEGER`.
2. Define a foreign key in `Book`, i.e. an attribute, say `FK_Writer_id`, which is **not** a class property and is mapped to the python type 'int' and the SQL type `INTEGER` as well.
3. Define a relationship *toBooks* in `Writer`, joining the primary key `id` to the destination entity `Book` 's foreign key `FK_Writer_id`, with a multiplicity's upper bound > 1, say, `*`.
4. Define a relationship *toAuthor* in `Book`, joining the foreign key `FK_Writer_id` to the `Writer` 's primary key `Writer.id`, with a multiplicity's upper bound equal to 1.

One-to-one One-to-one relationships should be modeled as one-to-many relationships, to which you add custom validation logic to enforce that the toMany relationship does not have more than one object in relation (in the future we will support one-to-one relationship, by supporting propagation of primary keys)

Many-to-many While automatic handling of many-to-many relationships is not supported yet, many-to-many relationships can be modeled and used with minimal efforts. Please refer to the dedicated section 2.6.3, below.

You may want to specify an other inverse relationship than the one calculated by the framework. (Note that when working with the ZModelizationTool, it shows you the inverse relationship it finds for a given relationship when in the Entity's global view. If the inverse relationship is not found or is wrong, first check that the relationships are correctly defined.)

The way an inverse relationship is calculated is simple: the framework looks at the destination entity and searches for a relationship whose joins have source and destination attributes corresponding one-to-one to the destination and source attributes in the original relationship's joins. And if this is not clear, look at the example above (step-by-step procedure to build a one-to-many association): *toBooks* is the inverse relationship for *toAuthor* and conversely, as expected.

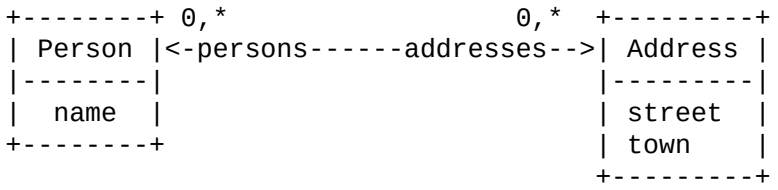
If, however, the framework is not able to find the correct inverse relationship, you can force it to use you own point of view; for a complete description of how this can be done, refer to `CustomObject.inverseForRelationshipKey()` documentation string.

2.6.3 Modeling many-to-many relationships

Many-to-many relationships are not *automatically handled* by the framework yet, however they can be modeled and used with minimal efforts.

We'll first examine how many-to-many relationships are mapped in relational database schema. Then we will give a short example demonstrating how to do this, including the code that you'll have to insert in your classes to handle them.

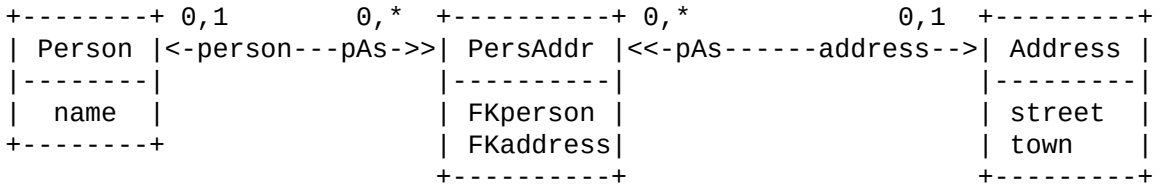
We will use the following model as an example:



General principle: the correlation table

Modeling many-to-many relationships in a relational database schema usually involves designing a *correlation table*, a table which holds the informations about the objects in the two tables, `Person` and `Address`. Each line in the correlation table is basically a tuple of two elements, one from `Address` and one from `Person`, so that we can tell that if `person_1` and `address_2` appear in a single row of the correlation table, we know that those two objects are related to each other.

Now let's integrate the correlation table into our model:



where:

- `PersAddr` is the abbreviation for PersonAddress (the correlation table is usually named after the names of the two tables it correlates)
- relationships named `personAddresses` are abbreviated with `pAs`

To fully understand what's going on here, let's look at the tables themselves. Suppose tables `Address` and `Person` have 3 rows each:

```
SELECT * FROM Person;
 id | name
--- +------
  1 | p1
  2 | p2
  3 | p3

SELECT * FROM Address;
 id |  street   | town
----+-----------+------
  1 | street a1 |
  2 | street a2 |
  3 | street a3 |
```

Then, for the following situation:

- `p1`'s addresses are `[a1, a2]`
- `p2`'s addresses are `[a1]`
- `p3`'s addresses are `[a1, a3]`

The correlation table looks like:

```
SELECT * FROM Person_Address;
 id | fk_person | fk_address
----+-----------+------------
  1 |         1 |          1    # p1 <--> a1
  2 |         1 |          2    # p1 <--> a2
  3 |         2 |          1    # p2 <--> a1
  4 |         3 |          3    # p3 <--> a3
  5 |         3 |          1    # p3 <--> a1
```

A short example

Now that we now how many-to-many relationships are handled, we know what we should do:

1. define the correlation table as an entity in the model,
2. connect the two tables to the correlation table and back,
3. because we do not want to directly manipulate objects coming from the correlation table, we will write some code so that we can directly access persons from a given address, and addresses from a given person;

We've seen in the previous how the model looks like with the correlation table. Here is the full PyModel:

```
from Modeling.PyModel import *

# defaults
AString.defaults['width'] = 40
Entity.defaults['properties'] = [
  APrimaryKey('id', isClassProperty=0, isRequired=1, doc='PK')
]

##
_connDict = {'database': 'MM1', 'user': 'postgres', 'host': 'localhost',
             'password': ''}
model = Model('MM1',adaptorName='Postgresql',connDict=_connDict)
model.version='0.1'
model.entities = [
  Entity('Person',
         properties=[ AString('name',isRequired=1) ] ),
  Entity('Address',
         properties=[  AString('street', isRequired=1),
                       AString('town') ] ),
  Entity('PersonAddress'),
  ]
#---
model.associations=[
  Association('PersonAddress','Person',
              relations=['person','personAddresses'],
              delete=['nullify','cascade'] ),
  Association('PersonAddress','Address',
             relations=['address','personAddresses'],
             delete=['nullify','deny'] ),
  ]
```

We just defined a new entity, `PersonAddress`, and two associations modeling the relationships between the correlation table and the correlated ones.

Last, we'll need to add some code in classes `Person` and `Address` to directly manipulate the many-to-many relationships. Since relating an object to an other one is just a matter of adding a object/a row to the correlation table, the code is pretty straightforward.

In `Person`, you'll add the following methods:

```
# Relationship: addresses
def getAddresses(self):
  return self.valueForKeyPath('personAddresses.address')

def addToAddresses(self, address):
  if address in self.getAddresses():
    return
  from PersonAddress import PersonAddress
  _pa=PersonAddress()  # Create an object in the correlation table
  self.editingContext().insert(_pa)
  self.addToPersonAddresses(_pa)
  _pa.setPerson(self)
  _pa.setAddress(address)
  address.addToPersonAddresses(_pa)

def removeFromAddresses(self, address):
  _pa=[pa for pa in self.getPersonAddresses() if address == pa.getAddress()]
  if not _pa:
    raise ValueError, 'Cannot find address'
  # Here we simply need to remove the corresponding object in the
  # correlation table
  _pa=_pa[0]
  self.removeFromPersonAddresses(_pa)
  _pa.getAddress().removeFromPersonAddresses(_pa)
  _pa.setAddress(None)
  _pa.setPerson(None)
  self.editingContext().delete(_pa)
```

And you'll add the equivalent methods in `Address`:

```
# Relationship: persons
def getPersons(self):
  return self.valueForKeyPath('personAddresses.person')

def addToPersons(self, person):
  if person in self.getPersons():
    return
  from PersonAddress import PersonAddress
  _pa=PersonAddress()  # Create an object in the correlation table
  self.editingContext().insert(_pa)
  self.addToPersonAddresses(_pa)
  _pa.setPerson(person)
  _pa.setAddress(self)
  person.addToPersonAddresses(_pa)

def removeFromPersons(self, person):
  _pa=[pa for pa in self.getPersonAddresses() if person == pa.getPerson()]
  if not _pa:
    raise ValueError, 'Cannot find person'
  # Here we simply need to remove the corresponding object in the
  # correlation table
  _pa=_pa[0]
  self.removeFromPersonAddresses(_pa)
  _pa.getPerson().removeFromPersonAddresses(_pa)
  _pa.setAddress(None)
  _pa.setPerson(None)
  self.editingContext().delete(_pa)
```

Now we can normally call e.g. `getAddresses` or `addToAddresses` on a `Person` object, passing a `Address` object, without caring about the details anymore.

2.6.4 How to model inheritance

When you have inheritance hierarchies in your object model, they should be correctly modeled. We present below the three possible ways to map an inheritance hierarchy to the relational world, then we'll see how to model these so that they can be taken into account by the framework.

There are three typical ways to model an inheritance hierarchy into an RDBMS schema–we do not take into account special RDBMS functionalities, just raw relational/SQL possibilities. Among these three, only one, *Horizontal Mapping*, is supported by the framework so far.

A more complete discussion, along with figures and examples, can be found at *http://www.objectmatter.com/* (at http://www.objectmatter.com/vbsf/docs/maptool/ormapping.html),

Horizontal Mapping In this configuration, each entity stores all instance data in a single table, whatever the inheritance hierarchy can be. That is to say that, if class `A2` inherits from `A1`, they each have their own table (respectively, table `A2` and table `A1`) which stores the instance data. This configuration does not put any information about the inheritance schema into the relational database, rather it lets the runtime do the job and take the inheritance into account.

Vertical Mapping (not supported yet) In this configuration, the root entity has its own table. A sub-entity then defines a table whose columns corresponds to the set of attributes that are *not* inherited; inherited informations, such as attributes or foreign keys for inherited relationships, are stored in the root entity's table, and are retrieved using a SQL join between the two tables.

Filtered Mapping or "Single table" (not supported yet) In this configuration, a root entity and all its sub-entities store their data in the same table. This table hence defines the whole set of possible attributes. To be able to distinguish rows corresponding to `A1` instances from rows corresponding to `A2` instances, each entity defines a qualifier. For example, instances of `A1` will have a type code of `1` stored in the table's column `'TYPE'`, while instances of class `A2` will map to rows whose type is `2`.

Note: Remember the fact that the model, build in the `ZModelizationTool` or directly in an xml-model and used by the framework's core, is *not a model per se* but rather a *mapping of an object model to a relational model*. As a consequence, you should not be surprised that, whatever mapping you choose to model a specific inheritance hierarchy, each entity and all sub-entities **must** define and describe all entity attributes and relationships–inherited or not.

Horizontal Mapping

(requires review)

When you use Horizontal Mapping, you should, at first, define and fully describe the classes that do not participate in the inheritance hierarchy, then you model the root classes, then their sub-entities, etc.

The reason is that every sub-entity should describe every attributes and relationships they inherit from its parent. The `ZModelizationTool` offers you the opportunity to automatically derive an sub-entity from a already defined entity, so you don't not have to re-declare every inherited properties by hand. If you're designing the model outside the tool, by directly editing the xml file for example, you'll probably want to define the root entity first, then to copy-paste the entire `<entity>` declaration and rename it with the sub-entity's name.

Warning! During the development phase, your model will probably change often, and your root-entities are likely to add and/or suppress attributes after you create their sub-entities. In such cases, you should be particulary careful to propagate these modifications to the sub-entities, or things will go really wrong. You can do this, either in the `ZModelizationTool` itself, or directly in the XML file with your favorite editor (be sure to import/export the XML file from/into the `ZModelizationTool` before/afterwards if you're using the tool and occasionally edit the xml-file by hand).

You will find an example of a model defining an inheritance hierarchy in the package `Modeling.tests.testPackages.StoreEmployees`.

Vertical Mapping

Not supported yet.

Filtered Mapping

Not supported yet.

2.7 Tools

2.7.1 The ZModeler

The ZModelization is a tool to help you design a model.

It lets you create and edit the model through a web interface, that is to say that every single element being part of the xml-model can be modified through easy-to-use web forms. It has some niceties, such as the possibility to create a relationship and its inverse along with the foreign key if needed in a single operation, or the ability to create in a single click a sub-entity having all of its parent's attributes and relationships.

Last, you may validate the model, generate the database schemas and the python code for your model, just by a few mouse clicks.

Installation To install the tool, you first need to install Zope (http://www.zope.org). Then copy the directory 'ZModelizationTool/' into zope's `Products` directory (located in '$ZOPE_HOME/lib/python/' in a brand new zope installation[6]).

Run Zope, access to the zope management interface and create a 'ZModelization Tool' where appropriate (for example, in the root directory). That's it.

2.7.2 Scripts

(requires review)

If you choose not to use the ZModeler and design your xml-models by hand, you may still take advantage of some scripts to help you with the following tasks:

- 'mdl_validate_model.py': validation of an xml-model
- 'mdl_generate_DB_schema.py': generation of the database-schema
- 'mdl_generate_python_code.py': generation of the python code from an xml-model

Running each of the above scripts with `--help` will give further details.

2.8 Some References on E.-R. Modelling

- Peter P. Chen, 1976 – see. http://bit.csc.lsu.edu/~chen/display.html

[6] or in the 'Products/' directory of your *instance home* –see *Make your life easier with INSTANCE_HOME* (at http://www.zope.org/Members/4am/instancehome) for details

- CASE* Method – Entity Relationship Modelling, Richard Barker, Addison-Welsey Publishing Company, ISBN 0-201-41696-4, 1990
- Entity-Relationship Approach: Ten Years of Experience in Information Modeling – Proceedings of the Fifth International Conference on Entity-Relationship Approach, Dijon, France, Nov. 17-19, 1986 Edited by Stefano SPACCAPIETRA, Noth-Holland, ISBN: 0-444-70255-5 Publishers: Elsevier Science Publishers B.V., Amsterdam, Holland.
- Rational white paper addressing a possible extension for UML – *Data Modeling Profile* (at http://www.rational.com/products/whitepapers/101516.jsp)

CHAPTER

THREE

Functionalities for Object Management

All python objects must inherit from the framework class `CustomObject`, to be thus connected to the framework's core. The `CustomObject` class defines all the necessary logic to make it possible to ensure the objects' persistency within the RDBMS, as well as numerous functionalities to help you with your own business logic. For those purposes, it works with a model, from which the classes are generally generated.

This chapter presents some important issues that you should not forget when coding and customizing your classes and logic–particularly important are the methods `willRead` and `willChange`. We will also look at other functionalities you can take advantage of, among which: validation of the referential constraints and custom validation logic.

Note that, you may start working directly with the generated classes for your model–adding, modifying, and getting data (as explained in chapter 4)–and letting the framework do the above-mentioned tasks transparently. The information here is provided to allow you to better understand how the framework performs these tasks, and to thus make it easier to add to, or modify, the framework's default behaviour to better suit your application needs, e.g. to add custom validation logic.

Note: Only a small sampling of the methods and functionalities of the `CustomObject` class, and other supporting classes, are mentioned here. For a complete picture you will need to look at the source doc strings, for `CustomObject` itself, as well as the doc strings for the interfaces it implements, namely `RelationshipManipulation`, `KeyValueCoding` (these two interfaces are exposed in details in chapter 8) and `DatabaseObject`.

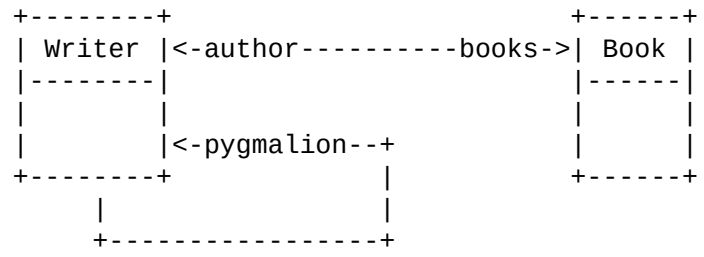
3.1 From model to python code

A (valid!) model contains all the necessary informations to generate a usable python package that can be used immediately to store and retrieve informations in a database.

In this section, we examine the different ways offered in the framework to derive usable python code from a model; the model we'll use is `AuthorBooks`, that you'll find it in:
'Modeling/tests/testPackages/AuthorBooks' (either the pymodel in `pymodel_AuthorBooks.py`, or the xml-model in `model_AuthorBooks.xml`), and which is described in section 2.1.1.

Note: Even if the framework offers different ways of deriving python code from models, you do not *have to* use these tools: they are provided for convenience but you can choose to write your package and code by other means of your own. If you choose to do so, you'll find below in the section **??** the few requirements needed to bind your classes to a model and to the framework.

Quick reminder: in this model, we have two entities, `Writer` and `Book`:



3.1.1 Generating the python code

The minimum

The first option you have is to generate the whole python package `AuthorBooks` and its modules `Writer.py` and `Book.py`.

For that purpose, you'll use the script **mdl_generate_python_code.py** (note that the same functionalities are also offered in the ZModeler).

On the command-line, type:

```
mdl_generate_python_code.py model_AuthorBooks.xml
```

We'll see in the next paragraph that there exists two different way of generating the code. That one is the simplest and uses the default option of the script **mdl_generate_python_code.py**: **-C** or **--compact-generation-scheme**[1].

The script creates the python package `AuthorBooks` in the current directory, in which you'll find the following files:

```
AuthorBooks/
|--   Book.py
|--   Writer.py
|--   __init__.py
|-- * model_AuthorBooks.py
|-- * model_AuthorBooks.xml
'--   setup.py
```

Note: the files marked with a star (`*`) are the only one that are overwritten by the script, if they already exist.

- `__init__.py`, which takes care to load the model within the default `ModelSet`,
- a copy of the model in the corresponding xml file which is used, and a python file containing a copy of the xml model as well; the reason for which the model is copied into a python file is that it makes it easy to distribute and install it along w/ the other modules using the standard distutils.

  Also note: even if you generate the python module from a pymodel, you'll only get those two xml-models in the generated package. This might change in the future, in the meantime, feel free to replace them with your pymodel, the code generated in '__init__.py' loading the model is capable of finding either a pymodel or an xml-model (see Model.searchModel() (at http://modeling.sourceforge.net/API/Modeling-API/public/Modeling.Model-module.html#searchModel)

[1] Please refer to **mdl_generate_python_code.py --help** for a comprehensive view on the script's usage and options

- `Writer` and `Book`: they contain the classes `Book` and `Writer`, ready to be used
- `setup.py`: a `distutils` script that you can use to install the package, to distribute it, etc. (see Distributing Python Modules (at http://www.python.org/doc/current/dist/dist.html) and Installing Python Modules (at http://www.python.org/doc/current/inst/inst.html) for details about the Python Distribution Utilities).

This bunch of files gives a minimalist view of what is needed to bind python code to a model and to the framework –this subject is fully discussed in the dedicated section 3.2. You can use it:

- to quickly test a model, but in this case, you'll probably prefer to dynamically build the package and modules (see section 3.1.2, below),
- or to start coding your classes from that point. Now if you want to use the generated code as a basis for your own development, we strongly advise you to consider using the other generation scheme exposed in the next paragraph.

Separating the generated code from your own work

During development, especially at early stages, the model is likely to change at a high rate, and so does the code which is automatically derived from the model. In such situations (and, in fact, each time the model change), it is quite easy to see that the so-called "compact scheme" used in the paragraph above is not very convenient: the script does not overwrite any existing file, so integrating the new changes consists in, for example, moving the python file elsewhere, regenrate the code, then integrate any code you may have written by hand from the previsouly moved python file to the new one...

That's why the script **mdl_generate_python_code.py** has an other option: **-B** or **--base-generation-scheme**, which is specially designed to keep the generated code separated from the business logic you add, so that you'll never have to worry mixing your own logic with automatically derived portions of code.

As an example, let's see the generated files by this scheme:

```
AuthorBooks/
|-- Book.py
|-- MDL
|   |-- * Book.py
|   |-- * Writer.py
|   |-- * __init__.py
|   |-- * model_AuthorBooks.py
|   `-- * model_AuthorBooks.xml
|-- Writer.py
|-- __init__.py
`-- setup.py
```

As you can see, a new subpackage `MDL` is created, where the files are always overwritten when regenerating the code –and more: the files within that directory are the only ones that can be overwritten by the script.

The modules `Book` and `Writer` in the top-level directory are the place where you'll put your own code; the classes within directly inherit from the ones in the `MDL` subpackage, keeping the automatically generated portion of codes completely separated from your own code.

3.1.2 Building the python package dynamically, at run-time

You also have the option to derive all necessary modules and classes directly from a model, at runtime. The module `Modeling.dynamic` has been added for that purpose, and offers two different options.

Dynamic generating of “standard” python code

The first method simply consists in building the exact same code as the one generated by `mdl_generate_python_code.py`:

```
### Load the model
def load_model():
  from Modeling import ModelSet, Model
  model=Model.searchModel('AuthorBooks', '.', verbose=1)
  ModelSet.defaultModelSet().addModel(model)

# build and use it!
from Modeling import dynamic
dynamic.build(model, define_properties=0)
from AuthorBooks.Book import Book
```

As expected, if you call `build` with `define_properties=1`, the method adds python properties (see `property')` in built-in functions (at http://docs.python.org/lib/built-in-funcs.html) for each attribute or relationship in the entity, so that you do not need anymore to use e.g. `book.getTitle()` or `book.setTitle`, but simply `print book.title` or `book.title="my title"`.

Dynamic generation using metaclass

The second method makes use of the metaclass `dynamic.CustomObjectMeta`:

```
# file: Book.py
# We assume that the model is already loaded
from Modeling import dynamic

class Book:
  __metaclass__=dynamic.CustomObjectMeta
  entityName='Book'
  mdl_define_properties=1

  # your own code here
```

The metaclass `CustomObjectMeta` automatically adds all the necessary methods to a class for integration with the modeling framework. It looks for the following attributes in the class:

- `entityNamed`‘: mandatory, this is the name of the class’ entity. The corresponding model should have been loaded prior to the class declaration, or you’ll get a `dynamic.EntityNotFound` exception.
- `verbose_metaclass`: if set, the metaclass prints on `sys.stderr` some info while building the class.
- `mdl_define_properties`: if set, the metaclass will also add properties for each attribute on relationship in the entity.

Last, then `dynamic` module also offers a method `build_with_metaclass(model, define_properties=0, verbose=0)` which you can use to derive the necessary package and modules from a model just like `dynamic.build()` we saw above, the ony difference being that the classes created at runtime use the metaclass approach.

3.1.3 Static vs. dynamic: what's best?

The question of which option one should use is an opened question! In fact, it highly depends on your own preferences: some people do not want to hear of code generators, others keep metaclasses away from their code. Some argue that it is best to maintain the model only, deriving the necessary modules at runtime, while others prefer to statically generate the code so that it can be put under version control, for example, or because they want to keep an eye on the generated parts.

This being said, here are a few comments:

- if you go for the static generation of python code, better choose the "base" scheme if you use the script `mdl_generate_python_code.py` (or a similar approach if you build the code by your own means).
- whichever method you choose, static generation, dynamic build with or without metaclasses, it is "officially" supported. Some people have argued that having all these possible ways available makes it difficult to make a choice; our position is that since we are here speaking about the code people write, we shouldn't force anyone to follow a given path if they prefer the other one. We think (and hope!) that the offered possibilities cover most code pratices –if you think we forgot something, we'll be happy to hear from you.

  Now if you're really looking for an advice, let's say that our personal preference consists in dynamically building the modules, using metaclass and properties.

3.2 The framework's requirements on python code

3.2.1 Package's, modules' and classes' names

When it fetches objects from the database, the framework needs to be able to instantiate every classes referenced in a model, so it also needs to find them! It does so by simply importing the class with a statement equivalent to:

```
>>> from package_name.module_name import class_name
```

where the package's name, the module's name and the class' name are the one provided in the model (the package's name is in the model's properties while the two others are in the crresponding entity's properties, see sections 2.3.1 and 2.3.2).

Thus, these three properties should always be kept in sync with the corresponding code.

3.2.2 Within classes

A class corresponding to an entity must meet the following requirements:

Inheritance: the class must include `CustomObject` in its inheritance list

Initializer: the method `__init__()` should be designed so that it is possible to call it with no arguments at all: for example, if the method accepts arguments, each one should have default values. The reason for this is that the framework relies on a call to `__init__()` without any arguments, when it needs to create an instance before populating it with data fetched from the database.

Entity's name: the class must define a method `entityName()` taking no argument: this is the way the framework currently binds an object to its entity. This should be changed (see TODO)

`willRead()`: this method defined in `CustomObject` must be called prior to accessing an object's property[2]. It informs the object that it is time to fetch the values stored in the database if it's not done yet. This is because

[2] i.e. either an attribute or a relationship defined in the entity

objects can be "*faults*" (ZODB speaking, they are ghosts), i.e. they have been instantiated but not fully initialized yet (lazily initialization)

`willChange()`: defined in `CustomObject` as well, this method should be called prior to modifying an object's property. This is part of the `Observing` interface, and its purpose is to notify the `EditingContext` that the object is about to change: the `EditingContext` needs to keep track of changes in its graph of objects in order to be able to save the changes (ZODB speaking, this is what the mix-in class `Persistent` does transparently for immutable attributes (see also: TODO)).

Of course, `willChange` automatically invokes `willRead`.

Getters, setters: although the python code derived from a model stores its properties in attributes beginning with an underscore (for example, `_lastName` for attribute `lastName`), the framework itself does not require this. Instead, it accesses and sets the values using the so-called private API of KeyValueCoding; shortly said, this means that in order to read a property 'name' for example, it tries to find either an attribute or a method called '_name', '_name()', '_getName()', 'name', 'getName()'. Refer to 8.2 for a complete overview.

Warning: It is your responsability to call `willRead` and `willChange` when you are about to, respectively, access or change an object's property corresponding to a Model's Attribute or Relationship; if you do not, the `EditingContext`, which is responsible for examining the changes and making them persistent, is likely to do only part of its job, possibly resulting in objects being partially saved, or not at all.

3.3 Automatic validation of referential and business-logic constraints

A major part of the checks made at runtime are validation operations –for example, you want to verify that a zipCode has a valid format, that the age of a person is a positive integer, etc.

The framework defines an interface, `Validation`, which allows you and the framework as well to trigger these validations when needed, in a defined scheme. You automatically get these functionalities when your class inherits from `Modeling.CustomObject` and corresponds to an entity defined in a model.

There is, roughly, two kinds of validation checks. The first one consists in checking individual properties of an object (such as: check that a value for a given key is of the proper type, that its width is ok if it is a 'string', etc.); the second one considers objects as a whole: its checks are like post-conditions, or consistency checking. We will see in the following how both can be adjusted and triggered.

3.3.1 Integrity constraints derived from the underlying model

When you define a model, you also define a number of constraints that has to be checked. For example, lower- and upper- bounds of a relationship's multiplicity, say (min=1, max=3), specify that an object cannot have less than one object (of a certain type!) in relation with itself, and no more than three as well. You can also mark an attribute as 'required' to avoid it being `None`.

The framework naturally verifies these constraints: the checking occurs automatically when an EditingContext is about to save changes (we will see hereafter how this can be manually triggered).

The constraints that are enforced are the following:

- Attributes:
  - type of the stored value,
  - if it is marked as 'required', it shouldn't be `None`
- Relationships:
  - the type of the objects in relation with oneself,
  - the number of objects in relation *vs.* the relationship's multiplicity.

3.3.2 Checking constraints: key by key

To verify a given value for a specific attribute, say on *lastName* for a `Writer` object, you use the method `validateValueForKey`:

```
aWriter.validateValueForKey('Cleese', 'lastName')
```

This check that the value `'Cleese'` is a valid value for *Writer.lastName*. Note that this value is directly given as a parameter and is not stored, nor searched, within the object: the validation can thus be done without actually assigning the value to an attribute (but the "global" checks we'll see hereafter do not work that way).

Return code

When the value is valid, `validateValueForKey` simply returns. But if it encounters an error, it raises the exception `Validation.ValidationException`. This exception holds a dictionary gathering the reasons of failure; this dictionary has the following characteristics:

- its keys are names of attributes for which the validation failed,
- the corresponding values indicates the reason(s) for the failure.

For example, suppose we called `aWriter.validateValueForKey('', 'lastName')` and that attribute *lastName* is marked as 'required', the raised exception's dictionary is then::

```
{'lastName': ['Key is required but value is void',],
}
```

If `validateValueForKey` raises an exception, the dictionary will only contain one key: the one that was supplied as the parameter 'key'. Its corresponding value is a list of all observed errors; here, it corresponds to the constant `Validation.REQUIRED` defined in the Validation interface. You will find there the complete list of possible error codes.

How to define your own validation logic

Now suppose that you want to enforce that the value stored in the *lastName* attribute does not begin with a `'J'`. This sort of constraint is not expressed within the model, so you have to write some code for that, and you want that to be checked transparently, along with other constraints.

The `validateValueForKey` is ready for such a situation: it expects you to write a method named `validateLastName` (note the capitalized letter in `validate`**`L`**`astName`). If it exists, then it gets automatically called. This is how you would write it:

```
from Modeling import Validation

def validateLastName(self, aValue):
  "Checks that the provided value does not begin with a 'J'"
  if aValue.find('J')==0:
    raise Validation.ValidationException
  return
```

Let's call `aWriter.validateValueForKey("Jleese", "lastName")` one more time, catch the exception, and checks its error dictionary:

```
{ 'lastName': ['Custom validation of key failed'],
}
```

Our own validation method has been taken into account, as expected, and the value `Validation.CUSTOM_KEY_VALIDATION`, part of the errors for key *lastName*, signals it.

3.3.3 `Validation.ValidationException`: the list of error-codes

The values that can be found in the list of reasons of failures for a given key are the following (extracted from the `Validation` interface):

```
REQUIRED="Key is required but value is void"
TYPE_MISMATCH="Wrong type"
CUSTOM_KEY_VALIDATION="Custom validation of key failed"
LOWER_BOUND="Lower bound of key's multiplicity constraint not fulfilled"
UPPER_BOUND="Upper bound of key's multiplicity constraint not fulfilled"
DELETE_DENY_KEY="Key has rule 'DELETE_DENY' but object still holds object(s)"
CUSTOM_OBJECT_VALIDATION="Custom validation of object as a whole failed"
OBJECT_WIDE="Validation of object as a whole failed"
OBJECT_WIDE_KEY='OBJECT_WIDE_VALIDATION'
```

The last three items occurs when an object is validated as a whole, not when its individual properties are checked; all other values can be returned by `validateValueForKey`.

3.3.4 Validating an object "as a whole"

We know how specific properties can be checked, but we still need a way to validate the consistence of the whole object, so that e.g. invariants of an object are ensured before its data is stored in the database. The framework defines the following methods for that purpose:

- `validateForInsert()`
- `validateForUpdate()`
- `validateForSave()`
- `validateForDelete()`

The two first methods simply calls the third one. The fourth one verifies a particular set of constraints we'll see in a moment.

`validateForSave()`

This method iterates on every key (attribute **and** relation) and calls `validateValueForKey` for each of them, using the stored value as the parameter 'value'.

Note: if you defined your own validation logic for some keys, they are called as well, as expected and seen above.

You can also define your own global validation method, like:

```
from Modeling import Validation
def validateForSave(self):
  "Validate "
  error=Validation.ValidationException()
  try:
    CustomObject.validateForSave(self)
  except Validation.ValidationException, exc:
    error.aggregateException(exc)
  # Your custom bizness logic goes here
  if self.getFirstName()=='John': # No John, except the One
    if self.getLastName()!='Cleese':
      error.aggregateError(Validation.CUSTOM_OBJECT_VALIDATION,
                           Validation.OBJECT_WIDE_KEY)
  error.finalize() # raises, if appropriate
```

Now suppose that our object `aWriter` stores `Jleese` for *lastName* and `John` for *firstName*. The dictionary of errors stored in the raised exception, after `validateForSave` is called, will be:

```
{ 'OBJECT_WIDE_VALIDATION':
            ['Validation of object as a whole failed',
             'Custom validation of object as a whole failed'],
  'lastName':
            ['Custom validation of key failed'],
}
```

The value `Validation.OBJECT_WIDE_KEY` is used to report global validation errors. You will find it **as soon as an error has been detected while validating**. Here you find an other value for that key, `Validation.CUSTOM_OBJECT_VALIDATION`, indicating that our own code did not validate the values as well. Note that the validation for key *lastName* has failed as well and is also reported.

Warning: IMPORTANT NOTE – contrary to what happens with methods `validate<AttributeName>`, the method `validateForSave()` we defined here overrides the definition inherited from `CustomObject`. Hence, it is very important not to forget calling the superclass' implementation, as described in the code above.

`validateForDelete()`

TBD.

3.3.5 Misc.

A `Validation.ValidationException` object defines `__str__`, so for the previous example, `str(error)` says::

```
Validation for key OBJECT_WIDE_VALIDATION failed:
- Validation of object as a whole failed
- Custom validation of object as a whole failed
Validation for key name failed:
- Custom validation of key failed
```

CHAPTER

FOUR

Working with your objects: insert, changes, deletion

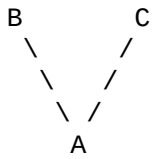
Now that we have seen how an object interacts with the core, we still have to learn about how they can be inserted, updated or deleted and how these changes can be made persistent into a RDBMS.

There is one component which fulfills these goals: the `EditingContext`. `EditingContext` can be thought as a graph of objects. Its major feature is to hold objects at runtime for which it is its responsability to track changes, either in the object (changes for some of an object's attributes) or between objects (when the objects in relationships change).

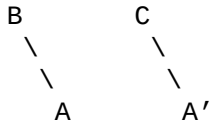
Everything you do within an `EditingContext` is likely to be made persistent (given that the objects derive from `CustomObject` and that they have a corresponding entity defined in some model).

4.1 Ensuring unicity of an object

An other feature assumed by `EditingContext`s is that they make sure that a given object has a single instance within them. So, if you fetch a particular object, then later access the same object e.g. by traversing some relationships, you are sure that the two are the same object (exactly the same, not only the same values). Put differently, say you have `B` in relation to `A` and `C` in relation to `A`, (and given that `A`, `B` and `C` are in the same `EditingContext` as they should be), you will always have this situation:



and you will NEVER get such a graph:



with objects `A` and `A'` ultimately referring to the very same row in the database.

To create an `EditingContext`, simply use the statements:

```
>>> from Modeling.EditingContext import EditingContext
>>> ec=EditingContext()
```

Note: You can refer to ‘modeling/tests/test_EditingContext_Global.py’ for examples of use – it covers all topics described here.

4.2 Inserting an object

As soon as an object is created, you should inform the `EditingContext` of its existence, so that, when instructed, it will be able to save it into the correct database. In fact, a brand new object should be inserted in an `EditingContext` as soon as you wish to establish some relationships with other objects.

To insert an object in an `EditingContext`, simply call ’insertObject()’:

```
>>> newBook=Book()
>>> ec.insertObject(newBook)
```

Alternatively, you can use the method `insert` ; both `insert` and `insertObject` are completely equivalent, and depending on your own feeling you may prefer one or the other.

```
>>> newBook=Book()
>>> ec.insert(newBook)
```

4.3 Updating objects

As explained in section **??**, the getters and setters take care of informing the `EditingContext` of any changes made to an object. Thus, except for the `willRead` and `willChange` methods in the getters and setters, you do not need to take any particular other action.

We will see in section 4.5 how to retrieve objects from the database.

4.4 Deleting an object

(We suppose here we already know how to get objects –this is covered by the next section)

When you want an object to be deleted, you inform it with the `deleteObject` message:

```
>>> ec.deleteObject(aBook)
```

Alternatively, you can use the method `delete` ; both `delete` and `deleteObject` are completely equivalent, and depending on your own feeling you may prefer one or the other.

You can also discard the insertion of an object you’ve just added:

```
>>> newBook = Book()
>>> ec.insert(newBook)
[... then at some point you can change your mind]
>>> ec.delete(newBook)
```

Note that, before a deleted object is about to be made persistent (i.e. when its corresponding row in the database is about to be deleted), some logic is triggered. For example, if this object still has some relationships but these relationships are marked as `CASCADE_DELETE`, the objects in relations will be deleted as well (given that their own validation logic allows them to be deleted, of course). Or, if it is marked as `DELETE_DENY`, the deletion will be denied and for that object to be deleted, you will need to remove any object in relation with it.

You can refer to the next section, 4.6, and to relationships' properties described in section 2.6.2 for further details.

4.5 Fetching objects

We will now see how you can fetch specific objects from the database. Note that once you fetched one or more objects, you do not need to explicitly fetch the objects in relations, this is done automatically. For example, once you have fetched a `Writer` object (see **??**), its *pygmalion* or its *books* will be transparently and automatically fetched when you ask for them (`aWriter.getBooks()`, so you do not have to worry about this. Moreover, these objects in relation to 'aWriter' are only fetched against the database where they are needed, so the memory footprint of your application remains reasonable (and the whole database is not fetched into memory when you access a single object!).

Naturally, before traversing relationships just like you normally do in python code, you need to fetch at least one object. We will see now how this is done.

In this section, we use the model described here: **??**, which is one of the model used in the unittests of the framework.

4.5.1 Principles

To fetch an object, you need:

1. to know the entity (or the root-entity of an inheritance hierarchy) corresponding to the object(s) you want to fetch
2. to qualify the object(s) you need to fetch, i.e. to provide some informations about the properties it/they have; this is optional: you may want to fetch all objects of a certain type.

As expected, the `EditingContext` is again the object to which we will ask for the service. The corresponding method is `fetch()`.

4.5.2 Simple fetch

To fetch all the `Writer` objects, you simply write:

```
objects=ec.fetch('Writer')
```

That's it.

Now what if you want the `Writer` objects whose last name is `'Hugo'`? That's not really more complicated:

```
objects=ec.fetch('Writer', qualifier='lastName=="Hugo"')
```

This is almost all you need to fetch any object from the database: just identify the type of the objects you're fetching, and the qualifier describing them. Everything is then done automatically, i.e. the correct SQL statement(s) are built and executed and the objects are then built from the retrieved rows and given back to you in a sequence.

In the next sections, we will see more complete examples, but it basically always this line that you'll use for fetching objects.

4.5.3 Pattern matching

Suppose you want to get all the writers whose names begins with 'Hu', you'll write:

```
objects=ec.fetch('Writer', qualifier='lastName like "Hu*"')
```

Available wildcards are `'*'` and `'?'`. The former matches any number of characters (including 0 –zero– character), the latter exactly one occurrence of a character.

You can also use the operator `'caseInsensitiveLike'`:

```
objects=ec.fetch('Writer', qualifier='lastName caseInsensitiveLike "hu?o"')
```

will match: `'Hugo'`, `'Hulo'`, `'HUGO'`, `'hUXO'`, ...

Alternatively, you'd probably prefer to use a shorted alias for `caseInsensitiveLike`: `ilike`

```
objects=ec.fetch('Writer', qualifier='lastName ilike "hu?o"')
```

Last, if you want to add raw `'*'` and `'?'` characters in a like pattern, escape them. For example, this fetchs all books whose title ends with `'here?'`:

```
objects=ec.fetch('Book', qualifier='title like "*here\?"')
```

4.5.4 Equality, comparisons, `in` and `not in`

When building your qualifiers, you can use the following operators:

- `'=='`: equality
- `'<'`: less than
- `'<='`: less than, or equal
- `'>'`: greater than
- `'>='`: greater than, or equal
- `'!='`: different than

- `'in'`: check that the value is in a list (rvalue is a list, not a tuple)
- `'not in'`: check that the value is in a list (rvalue is a list, not a tuple)

So, with this qualifier used to fetch `Writer` objects,

```
objects=ec.fetch('Writer', qualifier='age >= 80')
```

you'll get all the authors who are 80 years old or more.

Note: `in` and `not in` operators require that the right value is expressed as a list, and not as a tuple (i.e. surrounded square brackets '`[`' and '`]`'). For example:

```
objects=ec.fetch('Writer', qualifier='age in [82, 24]')
```

4.5.5 Negating, Con- or disjoining qualifiers

Now that we've seen all possible operators, we'll see that it is possible to conjoin, disjoin, or negate them. For example:

```
objects=ec.fetch('Writer', qualifier='lastName like "H*" AND age >= 80')
```

will give you the author whose lastName begins with the capitalized letter `'H'` and who are older than 80.

Likewise,

```
objects=ec.fetch('Writer', qualifier='age<50 OR lastName like "????"')
```

will give you the `Writer` objects who are less than 50 years old, or whose last name has exactly 4 letters.

Last, operator `'NOT'` negates the expression, so

```
objects=ec.fetch('Writer', qualifier='NOT(age<50 OR lastName like "????")')
```

will fetch the `Writer` objects who are older than 50 *and* whose lase name is not exactly 4 letter long.

Note: Operators `'AND'`, `'OR'`, `'NOT'`, `'IN'` and `'NOT IN'` can be written upper-case or lower-case, while operators `'like'`, `ilike` and `'caseInsensitiveLike'` should be written as-is (exact typo.)

> **Warning:** The precedence of the operators is not very clear –it depends on a parser (spark: cf. spark home page (at http://pages.cpsc.ucalgary.ca/%7Eaycock/spark/)) which I do not fully understand, I must admit. Moreover, it might be subject to changes.
>
> For example: `NOT age<50 OR age>200` is in fact equivalent to `NOT(age<50 OR age>200)`.
>
> Hence, in general, you are strongly encouraged to enclose the expressions in brackets so that the precedence of operators does not intervene at all.

4.5.6 Dotted notation

Last, it is possible to qualify the fetched objects, not only on their own attributes, but also on their related objects' properties as well. The notation is the classical dotted notation, so, in order to fetch the `Book` objects written by authors having a pygmalion whose last name begins with the letter `'R'` (whew!), you simply write:

```
objects=ec.fetch('Book', qualifier='author.pygmalion.lastName ilike "r*"')
```

which is really much simpler to write python than to explain in english!

For the curious, this will trigger a SQL method like:

```
SELECT t0.id, t0.title, t0.FK_WRITER_ID, t0.PRICE
FROM BOOK t0
  INNER JOIN  (WRITER t1
               INNER JOIN WRITER t2
               ON t1.FK_WRITER_ID=t2.ID )
  ON t0.FK_WRITER_ID=t1.ID
WHERE UPPER(t2.LAST_NAME) LIKE UPPER('r%')
```

4.5.7 How much objects will a query fetch?

If you manipulate a big database and wants to bind the result of a query which is built e.g. using your application GUI, you certainly do not want to allow the users to fetch a whole table into memory just because they type `'*'` instead of `'B*'` in the query string.

In that case, you want to have the number of objects that would be retrieved using a given `FetchSpecification`, while *not actually fetching* the objects.

The `EditingContext` provides a dedicated method for that purpose:

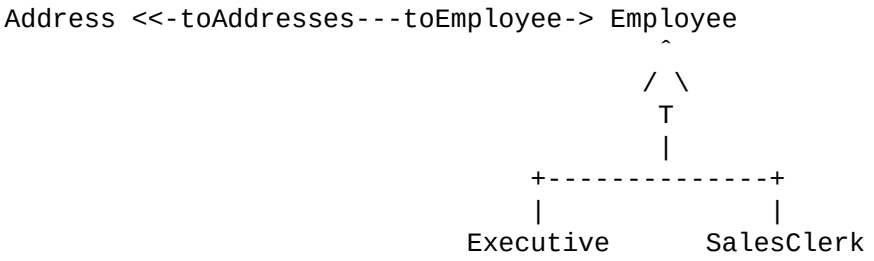
```
nb_of_objects=ec.fetchCount('Book', qualifier='...')
```

returns the number of objects that would be returned if the method `fetch()` is triggered with the very same argument.

4.5.8 Fetching and inheritance

(TBD: This section has to be rewritten – for the moment being it consists mostly of a copy-paste of messages exchanged on the mailing-list)

Suppose you have the following model:



(See the full model description in section 2.1.2)

- An address object can be transparently linked to either an Employee, an Executive or a SalesClerk instance. There is nothing particular to do, this is automatically handled.
- `address.toEmployee()` will automatically retrieve the right object (being an instance of one of those three classes). Here again, no particular action is needed, the framework handles it for you.
- Now, when fetching, you can specify whether you want to fetch a single class or its inheritance tree as well. Compare this, based on the test database and model `StoreEmployees`:

```
>>> from StoreEmployees import Address
>>> from Modeling.EditingContext import EditingContext as EC
>>> ec = EC()
>>> qualifier='toAddresses.zipCode like "4*"'
>>> ec.fetch('Employee', qualifier)
[]
>>> # Now fetch against the inheritance tree below 'Employee'
... all_objects=ec.fetch('Employee', qualifier, isDeep=1)
>>> [(o.entityName(), o.getFirstName(),o.getLastName())
...  for o in all_objects]
[('SalesClerk', 'John Jr.', 'Cleese'),
 ('SalesClerk', 'Jeanne', 'Cleese'),
 ('Executive', 'John', 'Cleese')]
```

As you can see, specifying **`isDeep=1`** when fetching allows you to fetch against the whole inheritance tree for a given entity (here, `Employee`).

4.5.9 The influence of an EditingContext on fetchs

Each `EditingContext` holds a different graph of objects, isolated from the other, where modifications, insertions and deletions can be made independently until they are saved in the database.

The state of a given `EditingContext` naturally have an impact on the objects you fetched. When nothing has been changed, you obviously get the objects as they are stored in the database. However, whenever you insert, modify or delete object, the result set of a fetch changes:

```
>>> from AuthorBooks.Book import Book
>>> ec=EditingContext()
>>> books=ec.fetch('Book')
>>> pprint.pprint([b.getTitle() for b in books])
['Gargantua',
 'Bouge ton pied que je voie la mer',
 'Le coup du pere Francois',
 "T'assieds pas sur le compte-gouttes"]
>>> new_book=Book()
>>> new_book.setTitle('The Great Book')
>>> ec.insert(new_book)
>>> books=ec.fetch('Book')
>>> pprint.pprint([b.getTitle() for b in books])
['Gargantua',
 'Bouge ton pied que je voie la mer',
 'Le coup du pere Francois',
 "T'assieds pas sur le compte-gouttes",
 'The Great Book']
```

As exposed above, newly inserted objects automatically appear in the result set when applicable. Of course, you can still benefit from standard fetch techniques, such as qualifiers:

```
>>> books=ec.fetch('Book', 'title like "*G*"')
>>> [b.getTitle() for b in books]
['Gargantua', 'The Great Book']
```

You get the expected result, even if the inserted book is *not* saved in the database yet.

If you modify your objects, these modifications will always be visible in your result set; continuing on the same example:

```
>>> gargantua=ec.fetch('Book', 'title == "Gargantua"')[0]
>>> gargantua.setTitle('Gargantua et Pantagruel')
>>> books=ec.fetch('Book', 'title like "*G*"')
>>> [b.getTitle() for b in books]
['Gargantua et Pantagruel', 'The Great Book']
```

Your changes to objects are not persistent yet in the database, still, they appear as expected in the result set.

Last, the same principles apply to deleted object:

```
>>> ec.delete(gargantua)
>>> pprint.pprint([b.getTitle() for b in ec.fetch('Book')])
['Bouge ton pied que je voie la mer',
 'Le coup du pere Francois',
 "T'assieds pas sur le compte-gouttes",
 'The Great Book']
>>> [b.getTitle() for b in ec.fetch('Book', 'title like "*G*"')]
['The Great Book']
```

Objects marked as deleted do not appear in the result set, even if the object still exists in the database (it will only be deleted when the `EditingContext` receives the `saveChanges()` message, see 4.6).

4.5.10 Fetching raw rows

Sometimes and for some reasons, you do not want to get a whole set of fully initialized objects.

For example, you need to get the data for a lot of objects to do some processing on some of its attributes, but you don't need the objects *per se*.

Or, building a GUI, you want to present a large list of items from which the user can e.g. choose one for, say, inspection or modification; here again, you do not need all the *objects* to build the list, and most of times you do not want it either since this would imply a too large memory footprint (along with a fetch taking too much time): all what you want is the raw data themselves, and the ability to turn them into real objects just when you need it (e.g. for modification).

Note: Of course, when fetching raw rows, you cannot benefit from most of the framework's capabilities since it manipulates objects, not dictionaries; you shoudl also note that the fetched data are not cached by the framework. However, in some situations like the ones we saw above, it's just what you want; moreover, it speeds up the fetch process, and reduces the memory footprint to just what is needed.

In this section we'll see how to do these both things: fetching raw rows and turning a raw into a real object.

Getting raw data

The framework offers a specific API for this:

```
>>> from Modeling.EditingContext import EditingContext
>>> import pprint, StoreEmployees
>>> ec = EditingContext()
>>> raw_employees = ec.fetch('Employee', isDeep=1, rawRows=1)
>>> pprint.pprint(raw_employees)
[{'firstName': 'Jeanne',
  'fkStoreId': 1,
  'id': 2,
  'lastName': 'Cleese',
  'storeArea': 'DE'},
 {'firstName': 'John Jr.',
  'fkStoreId': 1,
  'id': 1,
  'lastName': 'Cleese',
  'storeArea': 'AB'},
 {'firstName': 'John',
  'fkStoreId': 1,
  'id': 3,
  'lastName': 'Cleese',
  'officeLocation': '4XD7'}]
```

As you can see, with parameter **`rawRows`** you get the raw dictionary directly from the database. No object is initialized by such a fetch. However, you can use every functionality we already saw for fetching (inheritance with parameter `isDeep`, qualifiers), as is.

The very same rule applies to raw fetch as to "normal" fetch, in particular, everything we saw in the section 4.5.9 is still valid. If your `EditingContext` contains some newly inserted objects, you'll them appear; and deleted objects won't appear. For example:

```
>>> from StoreEmployees.SalesClerk import SalesClerk
>>> terry=SalesClerk()
>>> terry.setFirstName('Terry'); terry.setLastName('Gilliam')
>>> ec.insert(terry)
>>> salesClerks=ec.fetch('SalesClerk', rawRows=1)
>>> pprint.pprint(salesClerks)
[{'firstName': 'Jeanne',
  'fkStoreId': 1,
  'id': 2,
  'lastName': 'Cleese',
  'storeArea': 'DE'},
 {'firstName': 'John Jr.',
  'fkStoreId': 1,
  'id': 1,
  'lastName': 'Cleese',
  'storeArea': 'AB'},
 {'firstName': 'Terry',
  'fkStoreId': None,
  'id': <Modeling.GlobalID.TemporaryGlobalID instance at 0x84fcc64>,
  'lastName': 'Gilliam',
  'storeArea': None}]
>>> salesClerk.globalID()
<Modeling.GlobalID.TemporaryGlobalID instance at 0x84fcc64>
```

You probably already noticed the particular value associated to Terry's '`id`' field. Since this object is not saved in the database yet, its current `id` (the primary key for `SalesClerk` objects) is not determined yet either. Instead, the framework returns its identifier, which is a `TemporaryGlobalID`.

The same phenomenon appear on foreign keys when an object is related to a newly inserted object; continuing the previous example:

```
>>> pprint.pprint(ec.fetch('Executive', rawRows=1)) # No modifications yet
[{'firstName': 'John',
  'fkStoreId': 1,
  'id': 3,
  'lastName': 'Cleese',
  'officeLocation': '4XD7'}]
>>> from StoreEmployees.Store import Store
>>> parrot_store=Store()
>>> parrot_store.setCorporateName('We sell parrots')
>>> ec.insert(parrot_store)
>>> john=ec.fetch('Executive', 'firstName=="John"')[0]
>>> john.getToStore().removeFromEmployees(john)
>>> john.setToStore(parrot_store) ; parrot_store.addToEmployees(john)
>>> pprint.pprint(ec.fetch('Executive', rawRows=1))
[{'firstName': 'John',
  'fkStoreId': <Modeling.GlobalID.TemporaryGlobalID instance at 0x8364a0c>,
  'id': 3,
  'lastName': 'Cleese',
  'officeLocation': '4XD7'}]
>>> parrot_store.globalID()
<Modeling.GlobalID.TemporaryGlobalID instance at 0x8364a0c>
```

We see clearly here that this time, `fkStoreId` gets a `TemporaryGlobalID` corresponding to `parrot_store`'s global id. Moreover, we also remark that even if the modifications are not saved into the database yet, raw fetching returns the objects as they currently are in the `EditingContext`, just as with normal fetching.

Turning rows into real objects

Say you've presented to your user a large list of objects to choose from. Now the user selects one for modification or detailed inspection, probably including the objects in relations. There you need to get the magic on real objects back.

This is easily done with `EditingContext`'s `faultForRawRow`; continuing the previous example:

```
>>> raw_john=ec.fetch('Executive', 'firstName=="John"', rawRows=1)[0]
>>> pprint.pprint(raw_john)
{'firstName': 'John',
 'fkStoreId': <Modeling.GlobalID.TemporaryGlobalID instance at 0x8364a0c>,
 'id': 3,
 'lastName': 'Cleese',
 'officeLocation': '4XD7'}
>>> john=ec.faultForRawRow(raw_john, 'Employee')
>>> john
<Executive.Executive instance at 0x8531db4>
>>> john.getFirstName(),john.getLastName(),john.getToStore().getCorporateName()
('John', 'Cleese', 'We sell parrots')
```

Here we converted a raw dictionary to a real object, which is automatically registered within the framework with all its normal capabilities, just as if you directly fetch()'ed it from the database.

As you can see, `faultForRawRow()` takes a dictionary and the name of the entity the object belongs to, and returns the real object. Note that the entity's name shouldn't be exactly the right one: we asked for an `Employee`, we got an object of class/entity `Executive`, sub-entity of `Employee`. The only constraint on `entityName` is that it needs to belong to the same inheritance tree than the object's. This makes life easier when turning into objects raw rows which were fetched against a whole inheritance tree.

For example, you may fetch raw rows for the whole hierarchy beyond entity 'Employee': the rows you'll get will belong to either entities 'Employee', 'SalesClerk' or 'Executive', but this information is not enclosed within the dictionaries themselves[1]. When turning the raws back to objects, just specify the root entity, and the object of the right class will be automatically returned.

4.6 Saving Changes

When you are ready to save changes you made in an `EditingContext`, you simply send it the message `saveChanges()`:

```
ec.saveChanges()
```

which then performs all appropriate actions needed to make your changes persistent into the database.

Before it saves changes, the `EditingContext` checks every object against constraints derived from the underlying model, as well as your own validation logic if you have defined some; of course, deleted objects are examined as well

[1] except that, if the sub-entities have more attributes than their parent, you'll probably be able to guess to which entity a dictionary belongs to: for instance, only '`Executive`' objects have an attribute '`officeLocation`'.

and checked against possible specific relationships' constraints (such as: an object, which has a relationship whose deleteRule is DENY, should have no more objects registered to that relationship) –see 3.3 for details.

Then and again before saving changes, it propagates the deletions, if needed (cf. relationship's deleteRule(): CASCADE and NULLIFY). For precisions on this, and/or if you want to trigger this propagation at specific moments (e.g., when a HTTP request/response loop finishes), see documentation for `EditingContext.processRecentChanges()`.

4.7 Discarding changes: the destruction process of an `EditingContext`

Since an undo/redo mechanism is not supported yet, the only way to forget changes is to delete the `EditingContext` (at least, never send the `saveChanges` message to it).

4.7.1 Finalizing an `EditingContext`: breaking reference cycles

When an `EditingContext` is about to be deleted, its `dispose()` method gets called. This is followed with a two-step procedure, acting on each object registered within the `EditingContext`:

1. **Un-registered:** Each object held by the EC is unregistered and detached, and, if no other reference to that object exists, it will thus be ready to be garbage collected – however, other objects in the `EditingContext` are likely to hold a reference to it especially if your model defines relationships between objects (see below). From this point on, `obj.editingContext()` returns `None`.
2. **Invalidated:** Then, depending on the value returned by `invalidatesObjectsWhenFinalized()`:
    - If true, then the object receives the message `clearProperties`, defined in `DatabaseObject`, which empties its `__dict__` – this is called *invalidating the object*.
    - otherwise, no further action is taken.

The default behaviour for an `EditingContext` is to invalidate its objects. The reason for this is that the objects you fetch or create within an `EditingContext` are most of the time in relation with each other; these relations tend to create reference cycles between them, hence making it harder for the garbage collection to notice they are not referenced anymore. Invalidating each object actually breaks the reference cycles which helps the garbage collecter to detect that they should be destroyed.

4.7.2 Controlling the finalization stage

There are circumstances where the default behaviour is not what you want. For example:

- if your model does not define any relationships, you know you can safely skip invalidating your objects.
- If you are using an `EditingContext` in a batch that loads and processes lots and lots of objects, say 10,000 objects, after which the process simply dies, you really don't want to wait for python to invalidate every single objects, simply because this can take hours (I saw a python process stuck on this for an hour before I lost patience and killed it). You'd probably prefer to let the operating system release the memory allocated to the dying process.

Here is how you can control the finalization stage:

`EditingContext.invalidatesObjectsWhenFinalized_default` is a class attribute that controls the global behaviour for all `EditingContext` instances *except those which were specifically configured* (see below). If set to true, all existing and future EditingContexts will invalidate their objects when they are destroyed, otherwise they won't.

`EditingContext.setInvalidatesObjectsWhenFinalized()` controls whether a specific `EditingContext` instance invalidates its objects when it's been disposed. Note that this setting supersedes the previous one.

CHAPTER

FIVE

Nested `EditingContexts`

Sometimes you need more than a simple `EditingContext`. Suppose you have an application where a complex operation needs several steps to be performed by the user, each of these steps implying the insertion, deletion or updates of many objects. This complex operation can be a sub-process of an other long-standing process.

Normally you want to give to the user the possibility to cancel the modifications at any point in the sub-process. Most of the time, you also want all the changes made during these steps to be applied as a whole, or not at all–this is Atomicity. How can this be done?

One possibility could be to use an other `EditingContext`. It might be suitable for very simple cases, however you probably do not want this, because:

- the `EditingContext` already holds the objects we wish to work with, making re-fetching them into an other `EditingContext` seem unreasonable.
- Worse, if you have already updated, inserted or deleted some objects in your `EditingContext` which have not been saved yet, you will not see these changes in the new `EditingContext`–and you probably do not want to save them at this point.
- Plus, irrespective of the above two points, saving the changes in the second `EditingContext` would unconditionally save the changes (thus overwriting any changes in the first `EditingContext`) that were made but not saved before the complex operation began. This, again, is probably not what you would want.

An other solution could be to track all the changes made during the successive steps. However, this is a difficult task. Each step will probably trigger one or more methods transforming the objects; these methods probably trigger other methods, etc. It will be difficult to track the changes, and it will be even more difficult to revert them.

In such a situation, you will not be able to separately validate the changes from the complex operation: validation would have to apply to all changes, those made by the subprocess along with those previously made by the main process. Hence you have no way of separately checking the Consistency of the changes made in the sub-process itself.

Last, if your application maintains some other view based on the main `EditingContext`, you probably also do not want that the changes made in the sub-process are visible in the other views before the user validates (or discards) them. But insuring Isolation will be a real pain at best, impossible at worst.

5.1 Bringing transactions to the object world

A nested `EditingContext` is the solution for the problem. It provides a clean and convenient way of making changes on an other `EditingContext`, while exposing three of the four ACID properties at the object level:

- Atomicity: Changes made in a nested `EditingContext` will be all saved in the parent `EditingContext`, or none will be applied.

- Consistency: When saving the changes made in a nested `EditingContext` to its parent `EditingContext`, these changes will be checked just as when you save changes on an `EditingContext` to the database: there is no way to save an inconsistent state.
- Isolation: All changes made to a nested `EditingContext` are kept private within the nested `EditingContext`until they are saved to the parent `EditingContext`. Until this point, even the parent will not be aware of the changes.

In other words, you can think of a nested EditingContext as a transaction made at the object level.

Moreover, any object you will get in a nested `EditingContext`will reflect the changes made on it in its parent, even the uncommitted/unsaved ones, e.g. inserted objects will show up in the result set of a fetch, but deleted objects won't.

5.2 Declaring and using a nested `EditingContext`

Creating a nested `EditingContext` You create a nested `EditingContext` by simply supplying the parent `EditingContext` to the initializer:

```
>>> from Modeling.EditingContext import EditingContext
>>> parent_ec=EditingContext()
>>> child_ec=EditingContext(parent_ec)
```

Fetching objects You fetch objects exactly the same way you do with a standard `EditingContext`, as described in section 4.5, "Fetching objects".

However, the result set you'll get will probably be different than the one you would get by asking a standard `EditingContext`, because:

- every new object inserted in the parent `EditingContext` matching the fetch specification will show up in the result,
- you won't see any of the objects that are marked as deleted in the parent `EditingContext`,
- the states of the objects in the result set will reflect any changes made to them in the parent object store, even the uncommitted ones.

Current operations All currently supported operations on an `EditingContext`, such as inserting or deleting an object, making changes and validating them, are available in a child `EditingContext` without any modifications to your code.

Discarding or saving changes You save the changes made in a child `EditingContext` to its parent just by calling `saveChanges()` on it. Remember: a nested `EditingContext` saves its changes to its parent, NOT to the database. If you want to make them persistent, you will need the two following steps:

```
>>> child_ec.saveChanges()  # save changes to the parent_ec
>>> parent_ec.saveChanges() # save the changes to the database
```

Note: That's the reason why the fourth of ACID properties, *Durability* is not supported: since the changes are saved in the parent `EditingContext` and not to the database itself, they will be obviously lost if the application terminates abruptly.

Discarding the changes made in a child `EditingContext` is as simple as forgetting the child (e.g. dy deleting it). Remember that a parent `EditingContext` will never see the changes made in its children until `saveChanges()` is explicitly called on them.

Grand-children of an `EditingContext` Is it possible to declare a child of an already nested `EditingContext`? Yes! *Every* `EditingContext` can have children, irrespective of whether it is already nested; the depth of a hierarchy defined by parent/children `EditingContext` is indeed unlimited.

5.3 Miscellaneous developer's hints

How to dynamically identify a nested `EditingContext`? Simply invoke `parentObjectStore()` on it ; if the result is another `EditingContext`, it is nested. Otherwise it is a regular `EditingContext`.

```
>>> from Modeling.EditingContext import EditingContext
>>> parent_ec=EditingContext()
>>> ec=EditingContext(parent_ec)
>>> ec.parentObjectStore().__class__==EditingContext
1
>>> parent_ec.parentObjectStore().__class__==EditingContext
0
```

How to determine whether a given `EditingContext` is a (grand-)child of an other one? Use the method `isaChildOf`, which examines the parent/children hierarchy and answers appropriately:

```
>>> from Modeling.EditingContext import EditingContext
>>> grand_parent_ec=EditingContext()
>>> parent_ec=EditingContext(grand_parent_ec)
>>> ec=EditingContext(parent_ec)
>>> ec.isaChildOf(parent_ec)
1
>>> ec.isaChildOf(grand_parent_ec)
1
```

Is it possible to get the children of a given `EditingContext`? No.

5.4 Limitations with multiple child `EditingContexts`

You may wonder how child `EditingContext`s behave when changes in any one of them is saved to its parent. The answer, for now, is quite simple: the other children won't notice anything. This leads to some problems, depending on the changes affected in the child `EditingContext`:

1. if the children only update the data without inserting or deleting any object, you can declare several children for a given `EditingContext`. However, any changes committed by a child for an object `obj` will be **overriden** when an other child subsequently saves its own changes if, and only if, the same object `obj` has also been updated in that other child.
2. however, the possibility that multiple children would also concurrently insert or delete objects is **not supported**. We strongly advise against having more than one child for a given `EditingContext` if you insert or delete objects in the child, as this would lead to unpredictible behaviour.

CHAPTER

SIX

Integration in an application

When writing an application using the framework, you normally do not care about any of its components – except `EditingContext`s.

An `EditingContext` is the place where your objects live, i.e. where they are inserted, fetched, deleted, or updated. This important container holds your graph of objects as it gets changed, until the point where these changes are saved as a whole.

However, depending on the kind of applications you are developing, you will take different approaches. We'll see how this can be done, either in a pure-python application or within application servers like Zope or others. We suggest that you read the whole chapter, whatever your specific needs are: it explains how the framework reacts and how it can be used in standard situations.

6.1 Pure python applications

For a standard python application, you probably will not need more than one `EditingContext`, containing all your objects. Typically, you will delegate the building and servicing for that application-wide `EditingContext` to some central manager all your objects/widgets/whatever we have access to.

Additionally, you'll maybe need from time to time to create a child `EditingContext`, for example if you need to make some changes and processing in a pop-up window and want to propose a 'Cancel' button at any point of the process. See chapter 5 for details.

Python "batches": if you're designing python scripts that fetch and manipulate a lot of objects, please also read section 4.7 that gives valuable details about finalization of an `EditingContext`.

6.2 Instructions of use in a multi-threaded environment

The framework itself is designed to work in a multi-threaded environment, and it makes sure that critical sections accessing shared variables are correctly handled (such as when the module `Database` provides or updates the globally cached database snapshots).

However, the `EditingContext` itself is not MT-safe by default. If your application requires that an `EditingContext` is concurrently accessed by different threads, you have to make sure that you lock it before use (e.g., before saving changes); methods `lock()` and `unlock()` are provided for that purpose. Typical usage follows:

```
try:
  ec.lock()
  ec.saveChanges()
finally:
  ec.unlock()
```

Warning: Locking an `EditingContext` does NOT lock the objects it contains. If you want e.g. to be able to concurrently access & update the objects, you can take different approaches. For example, you might decide to use the `EditingContext` lock as a global locking mechanism, or you'll design your own locking scheme for your objects.

Last, two nested `EditingContext` which have the same parent and are managed by two different threads can concurrently save their changes to their parent without explictly locking it –this is managed automatically. This is worth noting, even if this is logical since the framework is supposed to ensure that any operations made on an `EditingContext` is safe in a multi-threaded environment (given that the `EditingContext` is not shared between threads, obviously).

6.3 Integration within application servers: using the sessioning mechanism

For an application run by an application server, different approaches are possible. All of them requires at some point that you have a sessioning mechanism at hand. It should not be a problem since most, if not all, application servers offer such a mechanism.

1. You can choose to have the same configuration as in a standalone python app., i.e. an application-wide `EditingContext`.
2. Or you can also decide to bind a different `EditingContext` to each session.

Each option has its drawbacks: some are inherently bound to what they are, some are due to specific features not being implemented yet. Let's look at the details.

6.3.1 Sharing an `EditingContext` between sessions

The first solution consists in having a single shared `EditingContext` in your application.

It obviously requires that you take all appropriate measures to ensure that your application is MT-safe, as described in the previous section.

What about making changes? Obviously you do not want your users to see the changes made by others until they are committed to the database. Thus, you'll occasionally create a specific `EditingContext` on which the changes will be made – we recommend that you use a child `EditingContext` in this case, see chapter 5. For the same reason, the `EditingContext` used for registering and saving the modifications should not be shared by different users.

Then, you must keep in mind that the shared `EditingContext` will receive all the objects that can be possibly loaded by the sessions coming up. The main problem with such an approach is that you will probably end up with all your objects being loaded in the shared `EditingContext` after some hours or days (depending on the number of hits your application receives, the number of objects that a request can load, etc.). If your database is quite big and/or if it quickly grows, your application is likely to end with exhausting the available memory.

What happens here is you do not have any means to distinguish between the objects loaded by session $S1$ and objects loaded by session $S2$; as a consequence you cannot clean up the shared `EditingContext` when a session is expired

and destroyed. The only thing you can do for clean-up is to detect that no more sessions are available: at this point, you would probably drop the existing `EditingContext` and create a brand new one.

For these reasons, we do not recommend this solution – except maybe in special cases where the database is a small one, but even then, why would you use a database if you can't count on your application to scale when the db is growing?

Note: At some point in the future, with the implementation of the following feature described below, this negative recommendation may change: We may support a special attribute for you to specify the maximum number of (unchanged) objects you want an `EditingContext` to contain. Then, when the maximum number of objects is reached, it would be possible to automatically clean the `EditingContext` (the oldest object would be re-faulted/invalidated, or "ghostified" in ZODB jargon). There is no ETA for this feature however, it is just a TODO item. If you think you need the feature, please go ahead and let us know!

6.3.2 Sessioning

The second approach involves a per-session creation mechanism for `EditingContext`s.

The framework provides the module `utilities.EditingContextSessioning` for such situations. This module acts as central repository to which a sessioning mechanism can be bound for creating, accessing and destroying an `EditingContext` as sessions come up and expire. The module documentation gives full details about its methods and how they can be used.

Known problem: By definition, such a configuration isolates the changes different users make *until* they are committed. The problem here is that even *after* a user has committed changes, the other users that are already connected and whose session already got an `EditingContext` **will not see** the changes made to objects previously fetched – new users connecting afterwards will see the changes, though, as will current users who did not fetch the updated objects before they were committed.

> This is due to a feature missing in the framework, where changes to an EditingContext are not broadcasted to others. Resolution of this issue is planned for after release 0.9.

There is currently no satisfying solution for this problem.

6.4 Zope

The framework is shipped with a particular component, `ZEditingContextSessioning`, that makes it possible to have an `EditingContext` lazily created on a per-session basis.

When installed in the `Products/` folder of a Zope instance, it automatically modifies the class `TransientObject`[1] and adds a new method to it: `defaultEditingContext`. It also binds itself to the sessioning machinery so that the `EditingContext` attached to a session is automatically finalized when the session is expired.

Accessing the `EditingContext` bound to a particular session is as simple as calling `defaultEditingContext()` on the `SESSION` object.

Note: Zope does not immediately destroy expired sessions ; they are marked as expired and are only destroyed when a given thread is elected for doing the "housekeeping" (see 'lib/python/Products/Transience/Transience.py', method_getCurrentBucket()). This means that the `defaultEditingContext` assigned to a session will not be finalized when the session expires, but a certain amount of time afterwards.

[1] The `SESSION` object is an instance of `TransientObject`

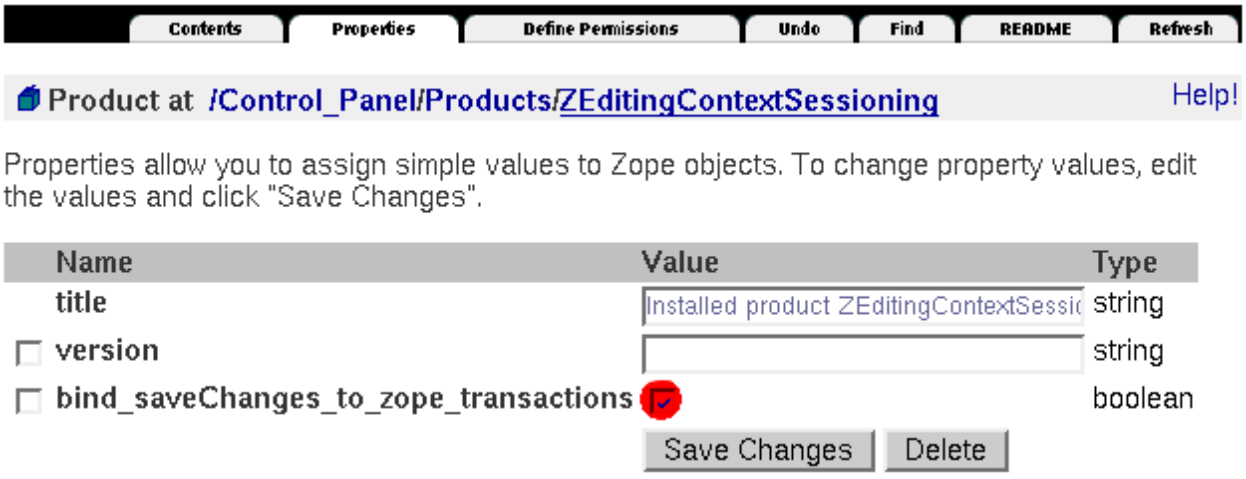


Figure 6.1: Binding Zope txn to SESSION's defaultEC() txn

6.4.1 Binding the default EC transactions to Zope transactions

A special property has been added to the `ZEditingContextSessioning` product which allows the transactions of a session's default `EditingContext` to be bound to Zope's transactions. This means that each time a Zope's transaction/request ends, `saveChanges()` is automatically called on the session's `defaultEditingContext()`.

To enable this feature, go to `Control_Panel >Products >ZEditingContextSessioning >Properties` and check the `bind_saveChanges_to_zope_transactions` box –see fig.6.1.

6.5 Others

As far as I know the framework has not been integrated in other frameworks. However it is shipped with a component, `utilities.EditingContextSessioning`, which should make it easy to bind it to any existing sessioning machinery. The documentation in this module gives all necessary details and instructions of use.

We will be happy to hear from you if you integrate it to other development platforms!

Part II

Advanced techniques

CHAPTER

SEVEN

Accessing a model and its properties

Every model required at runtime is loaded into a single `ModelSet`, the so-called “default model set”. It is accessed this way:

```
>>> from Modeling.ModelSet import defaultModelSet
>>> ms=defaultModelSet()
```

Suppose we already imported the two test packages `AuthorBooks` and `StoreEmployees`, then we have:

```
>>> ms.modelsNames()
['AuthorBooks', 'StoreEmployees']
```

The two corresponding models have been correctly imported, as expected.

Accessing a model in particular, or one of its entities is straightforward:

```
>>> model_AuthorBooks=ms.modelNamed('AuthorBooks')
>>> model_AuthorBooks.entitiesNames()
['Writer', 'Book']
>>> model_AuthorBooks.entities()
(<Entity instance at 8396b50>, <Entity instance at 8337cd0>)
>>> entity_Book=model_AuthorBooks.entityNamed('Book')
```

Note: you do not need to access a model to get one of its entities; since entities share a common namespace inside a `ModelSet`, this can be asked to the model set as well:

```
>>> ms.entitiesNames()
('Writer', 'Book', 'Store', 'Employee', 'Address', 'SalesClerk',
 'Executive', 'Mark')
>>> ms.entityNamed('Book')
<Entity instance at 8337cd0>
```

Last, given an entity, you can access its attributes and relationships quite the same way, for example:

```
>>> # All attributes' names
... entity_Book.attributesNames()
('title', 'id', 'price', 'FK_Writer_Id')
>>> # All relationships' names
... entity_Book.relationshipsNames()
('author',)
>>> # Class properties only
... entity_Book.classProperties()
(<Attribute instance at 839bc38>, <Attribute instance at 83c5720>,
 <Attribute instance at 83eaca8>, <SimpleRelationship instance at 840b280>)
>>> # and their names
... entity_Book.classPropertiesNames()
['title', 'id', 'price', 'author']
>>> # dealing with an attribute
... book_title=entity_Book.attributeNamed('title')
>>> book_title.type(), book_title.externalType(), book_title.width()
('string', 'VARCHAR', 40)
```

Each of the classes `Model`, `Entity`, `Attribute` and `Relationship` have more functionalities then that. We invite you to look at their respective API for further details.

CHAPTER

EIGHT

Generic manipulation of objects

In this chapter, we'll see how each object's properties can be dynamically handled.

8.1 Manipulating objects and their relationships

With the methods defined by the interface `RelationshipManipulation` (implemented by `CustomObject`), you can assign an object `Book` to another object `Author`, and conversely, without even knowing if the relationship has some inverse relationship, or even if the relationship is toOne or toMany: all that you need to know is the 'key' (i.e. the name of the relationship) to which you want to add an given object. Example:

```
aBook.addObjectToBothSidesOfRelationshipWithKey(anAuthor, 'author')
```

Here, the framework analyses the underlying model, does what is necessary so that your objects are in sync. Moreover, suppose `aBook` already has *author2* assigned to relationship *author*, then the inverse relationship joining *author2* to *aBook* is nullified, so that the graph of object remains consistent.

Compare this with that you normally do by hand, the explicit way:

```
_author=aBook.getAuthor()
if _author:
  _author.removeFromBooks(aBook)
aBook.setAuthor(anAuthor)
anAuthor.addToBooks(aBook)
```

`addObjectToBothSidesOfRelationshipWithKey` does exactly the same thing, taking care of all the details for you (even it's a bit slower than the explicit statements because it has to examine the model).

This can come in handy for rapid prototyping, or to make things smoother when you are beginning the dev. and that the model can rapidly change, or for designing generic algorithms where the manipulated objects and relationships are not known at runtime.

Last note: the inverse method is the following one:

```
removeObjectFromBothSidesOfRelationshipWithKey
```

and works on the same principles.

8.2 Accessing the objects' properties

(quick notes, needs to be further documented)

The `KeyValueCoding` interface is also implemented by `CustomObject`; it defines methods for accessing and setting attributes' values to keys. In fact, the `RelationshipManipulation` interface is, somehow, the equivalent of the `KeyValueCoding` for relationships.

The `KeyValueCoding` interface defines methods `valueForKey` and `takeValueForKey`. These methods allow you to access a property (i.e. the value stored for an attribute or a relationship) in an object without knowing exactly how it is stored. This is quite similar to, for example, the "Unified Dotted Notation" (see: Name Mapper Syntax (at http://www.cheetahtemplate.org/docs/users_guide_html_multipage/language.namemapper.html) and Underscored Attributes (at http://www.cheetahtemplate.org/docs/users_guide_html_multipage/language.namemapper.underscore.html) in CheetahTemplate's documentation).

8.2.1 How does it work

The `valueForKey` searches the property `'key'` in the following way: it first searches some getters (methods), and falls back to searching for instance's attributes if none where found; if you need to remember something, it is that methods are always preferred. The searches are made in the following order:

1. method `get`*`Key`*`()`
2. method *`key`*`()`
3. method `_get`*`Key`*`()`
4. method `_`*`key`*`()`
5. variable key
6. variable _key

The method `takeValueForKey` also searches setters, than attributes, in the following order:

1. method `set`*`Key`*`()`
2. method `_set`*`Key`*`()`
3. variable key
4. variable _key

8.2.2 The whole API

- `valueForKey()` gets a value for a given attribute's name,
- `takeValueForKey()` sets a value for a given attribute's name,
- `valueForKeyPath()` is working the same way than `valueForKey`, except that the key can be here expressed by dotted notation to automatically access an other related object's property (such as: `"author.lastName"` for a `Book` object –see examples, below)
- `takeValueForKeyPath()` is like `takeValueForKey` with dotted notations

Additionally, two more methods are available:

- `valuesForKeys()`: returns the list of values taken by the supplied attributes
- `takeValuesFromDictionary()`: sets each key/attribute with its corresponding value

8.2.3 Examples

Some quick examples:

- `aBook.valueForKey("title")` is equivalent to `aBook.getTitle()`
- `aBook.takeValueForKey("Les miserables", "title")` is equivalent to `aBook.setTitle("Les miserables")`
- `aBook.valueForKeyPath("author.lastName")` is equivalent to `aBook.getAuthor().getLastName()`
- `aBook.takeValueForKeyPath("Hugo", "author.lastName")` is equivalent to `aBook.getAuthor().setLastName("Hugo")`
- `aBook.valuesForKeys(['title', 'price'])` is equivalent to `[aBook.getTitle(), aBook.getPrice()]`

Of course, the objects' properties may always still be accessed using the getters and setters of your python objects,

8.3 Mixing KeyValueCoding and model's properties

We'll see that dynamic manipulation of an object's properties and relationships can be completely generic, using the techniques we saw in the previous chapters.

Since we know how model's properties can be retrieved, and how an object can be generically asked for a given property, we can combine these two techniques to generically manipulate any object, for example to print informations on a list of different objects:

```
>>> ms=defaultModelSet()
>>> objs=ec.fetch('Writer', qualifier='lastName=="Cleese"')
>>> objs.extend(ec.fetch('Book'))
>>> #
... # Now we will manipulate 'objs' without explicitly referring
... # to its methods
...
>>> for o in objs:
...   print 'Object ',o
...   for cp in ms.entityNamed(o.entityName()).classProperties_attributes():
...     print '    %s: %s'%(cp.name(),o.valueForKey(cp.name()))
Object  (<Writer.Writer instance at 0x8485a04>) John Cleese
    age: 24
    lastName: Cleese
    firstName: John
    birthday: 1939-10-27 08:31:15.00
Object  <Book.Book instance at 0x8491f94>
    title: Gargantua
    id: 1
    price: None
Object  <Book.Book instance at 0x848a1fc>
    title: Bouge ton pied que je voie la mer
    id: 2
    price: None
Object  <Book.Book instance at 0x848f50c>
    title: Le coup du pere Francois
    id: 3
    price: None
Object  <Book.Book instance at 0x84a38b4>
    title: T'assieds pas sur le compte-gouttes
    id: 4
    price: None
```

We can achieve the same thing with `valuesForKeys()`:

```
>>> for o in objs:
...   print 'Object ',o
...   cp_attrs=ms.entityNamed(o.entityName()).classProperties_attributes()
...   cp_attrs_names=[cp.name() for cp in cp_attrs]
...   print '    ',o.valuesForKeys(cp_attrs_names)
Object  (<Writer.Writer instance at 0x8485a04>) John Cleese
     [24, 'Cleese', 'John', <DateTime object for '1939-10-27 08:31:15.00' at 81984f0
Object  <Book.Book instance at 0x8491f94>
     ['Gargantua', 1, None]
Object  <Book.Book instance at 0x848a1fc>
     ['Bouge ton pied que je voie la mer', 2, None]
Object  <Book.Book instance at 0x848f50c>
     ['Le coup du pere Francois', 3, None]
Object  <Book.Book instance at 0x84a38b4>
     ["T'assieds pas sur le compte-gouttes", 4, None]
```

CHAPTER

NINE

Handling custom types for attributes

The framework handles automatically a subset of built-in python types: `int`, `string`, `float` and date types (e.g. `mx.DateTime`). We'll see here how you can make the framework automatically assign to attributes' values real objects.

9.1 Example: using FixedPoint for a price attribute

Sometimes you need more than this. Let's take the test package `AuthorBooks`, and suppose we want to use `FixedPoint`[1].

- change the model so that price is a `string`/VARCHAR (was: a `float`/`NUMERIC(10,2)`): we will store the `FixedPoint` object as a string, not as a float, because of the inherent imprecision which goes any (binary) representation of float[2]
- add `_setPrice()` and `_getPrice()` to `AuthorBooks.Book`:

```
PRECISION=2
def _setPrice(self, value):
  if value is None:
    self._price=None
  else:
    self._price  = FixedPoint(value, PRECISION)

def _getPrice(self):
  if self._price:
    return None
  else:
    return str(self._price)
```

Now let's test this: (remember to change the DB schema so that table BOOK's attribute `price` is a VARCHAR)

[1] `FixedPoint` package can be found on sourceforge (at http://fixedpoint.sourceforge.net/html/lib/module-FixedPoint.html)

[2] try to type '`0.7`' in a python interpreter:

```
>>> 0.7
0.69999999999999996
```

```
>>> from fixedpoint import FixedPoint
>>> from AuthorBooks.Book import Book
>>> from Modeling.EditingContext import EditingContext
>>> ec=EditingContext()
>>> book=Book()
>>> book.setTitle('Test FixedPoint')
>>> book.setPrice(FixedPoint("3.341"))
>>> book.getTitle(), book.getPrice()
('Test FixedPoint', FixedPoint('3.34', 2)) # precision=2
>>> ec.insert(book)
>>> ec.saveChanges()
>>> book.getTitle(), book.getPrice()
('Test FixedPoint', FixedPoint('3.34', 2))
```

Here you can check in you db that it was stored as a `varchar`, as expected. Start a new python and test the fetch:

```
>>> from fixedpoint import FixedPoint
>>> from Modeling.EditingContext import EditingContext
>>> ec=EditingContext()
>>> books=ec.fetch('Book')
>>> books[0].getTitle(), books[0].getPrice()
('Test FixedPoint', FixedPoint('3.34', 2))
```

As you can see, `FixedPoint` is now correctly and automatically handled by the framework.

This technique can be used for any custom type you want to use. The next section gives some details on how this works.

9.2 Behind the scenes

We have seen how to map any attribute's value to an instance of given class. Here again, this is the `KeyValueCoding` in action, as described in section 8.2.

The framework *always* accesses the attributes' values with the so-called "private" methods (`storedValueForKey()`, `takeStoredValueForKey()`). We already know that they will try to use private setters/getters –such as `_setPrice()` and `_getPrice()`– before the public ones –being `getPrice()` and `setPrice()`).

So, what happens here is:

1. when the framework is about to save the data, it collects the attributes' value using `storedValueForKey`. This one finds `_getprice()`, which gently returns the corresponding string,

   Note: the same happens for validation before saving: type checking also calls `_getPrice()` and gets a string, so everything's ok.

2. when the framework fetches the data, it uses `takeStoredValueForKey()` to initialize attributes' values; for the attribute `price`, this method finds and calls `_setPrice()` which turns the string back to `FixedPoint`.

Part III

Appendices

APPENDIX

A

Environment Variables

A.1 Core

Warning: Environment variable `MDL_PERMANENT_DB_CONNECTION` has been deprecated and has no effect anymore. It has been replaced by `MDL_TRANSIENT_DB_CONNECTION` which has the opposite semantics.

Name	Description	Possible values

`MDL_DB_CONNECTIONS_CFG`	It happens that a model can be written in different places: in an xml file, in a python file, and even in a pickle when you use `mdl_compile_model.py`. The consequence is that the connection dictionary itself for each model, containing the user and its password, is written two or three times in the filesystems. For security and administrative reasons, one would prefer that this sensitive information should be written in a single place.
	
	This environment variable allows you to externalize the connection dictionaries of all your models in a single file; you'll typically remove the user and password from your models, and add these informations to a single file, say '/full/path/to/db_conn_dicts.cfg', in a dedicated section for each model, like this:
	
	`[<ModelName>]`
	`user: <username>`
	`password: <passwd>`
	
	Then, simply assign to the env. variable `MDL_DB_CONNECTIONS_CFG` the full path to your 'db_conn_dicts.cfg', and your models' connection dictionaries will automatically be updated when they are loaded.
	Last, every parameter normally assigned to a connection dictionary can be defined in this file, such as: `host`, `port`, etc. An extra field can be specified, `adaptor`, which overrides the adaptor's name in your model (example value: `Postgresql`, `MySQL`).
	See also python documentation for `ConfigParser`.
		full path to an ini-file

`MDL_ENABLE_DATABASE_LOGGING`	DatabaseAdaptors for Postgresql and MySQL use a common logging mechanism, whose methods are located in module `Modeling.logging`. This logging is not activated by default, except for error and fatal errors. To enable it, you set this variable to any non-empty string ; log outputs go to the `sys.stderr`. To disable it, just unset this variable, or set it to the empty string.	e.g. '1', 'YES' (plus the empty string for de-activation)

Name	Description	Possible values

MDL_ENABLE_SIMPLE_METHOD_CACHE	When this variable is set to any non-empty string, the framework automatically caches the simple methods of models and class description (by simple methods, we mean methods taking `self` as their only argument). Since both of these objects are heavily used at runtime (because model introspection is needed at various stages), enabling this cache speeds up the simple operations (for example, the inverse relationship for a given relation is computed once, then it is cached).
	There are situations, however, where you do not want to enable this. For example, if after loading your model you need to change anything in it (such as modifying the connection string, changing an attribute's external type, adding a relationship, etc.), this shouldn't be enabled; if it is, the changes won't have any effect on the model themselves: e.g. you can set the connection dictionary but `connectionDictionary()` will keep returning its initial value. In such situations, you can either:
	• not enable this cache for the whole application,
	• or enable it after all changes have been made, by executing the following code:
	`import os`
	`from Modeling.ModelSet import defaultModelSet`
	`os.environ['MDL_ENABLE_SIMPLE_METHOD_CACHE']='Y'`
	`for model in defaultModelSet().models():`
	`  model.cacheSimpleMethods()`
	Attention: once the caching mechanism has been enabled, it cannot be disabled (so no more changes can be made to your models)
	Note: The environment variable `MDL_DB_CONNECTIONS_CFG` is taken into account to update a model's properties before it is cached, no particular action is needed in this case.
		e.g. '1', 'YES' (plus the empty string for de-activation)

MDL_TRANSIENT_DB_CONNECTION	By default, a database connection is left opened after each request, so that it can be re-used for subsequent requests. When this environment variable is set to any true value, any opened database connection is closed after it's been used –for example, the database connections needed to perform the actions defined in `ec.objectsWithFetchSpecification()`, `ec.objectsCountWithFetchSpecification` or `ec.saveChanges()` are closed as soon before these functions return.	e.g. '1', 'YES' (plus the empty string for de-activation)

A.2 Postgresql specific

Name	Description	Possible values
MDL_PREFERRED_PYTHON_POSTGRESQL_ADAPTOR	By default, the Postgresql Adaptor Layer uses the first module available among the following ones (in that order): psycopg, pgdb, pypgsql. If you have more than one of these modules installed, and you prefer that a specific one is used, you set this variable to the desired name.	'psycopg', 'pgdb', 'pypgsql'
MDL_POSTGRESQL_SERVER_VERSION	Postgresql servers v7.2 and 7.3 behave differently w.r.t. dropping tables and foreign key constraints. With v7.2 the foreign key constraints are not easily dropped, in fact the Postgresql Adaptor Layer does not generate any code for this ; this is however not a problem since v7.2 implicitly drops the related constraints when dropping a table. On the other hand, version 7.3 does not allow a simple DROP TABLE mytable when constraints are bound to table, but it makes it possible to identify and drop the PK/FK constraints. Hence the Postgresql Adaptor Layer can and will generate drop statements for integrity constraints before dropping a table –the other solution would be: DROP TABLE mytable **CASCADE** but this is explicitly and intentionally not implemented. Please note: this only affects db-schema (re)generation and model validation (because postgresql v7.3 does not support datatype DATETIME anymore) ; this does not affect the runtime core for the moment being.	'7.2' (default), '7.3'

A.3 Mysql specific

Name	Description	Possible values
MDL_MYSQL_SERVER_VERSION	The framework uses MySQL version number to determine whether SQL92 JOIN statements can be used; if not, the corresponding clause is added to the WHERE clause. This is because MySQL servers up to 4.0 do not handle JOIN statements in a standard way, making it impossible to generate a valid SQL query when more than 2 joins are involved. If the environment variable is left unset, it defaults to the value returned by MySQLdb.get_client_info().	'3.23', '4.0', 4.0.11a-gamma (see at end of description for the default value)

APPENDIX

B

Frequently Asked Questions

B.1 Designing the model

What if I want to move my package/modules to another package/location/whatever after it has been generated? If, after generation of code, you need to move the generated package to some other package, or if you need to move a class to some other module, just adjust you model to reflect your change. The following adjustments should be made:

- for the model, see field `'packageName'`
- for entities, see fields `'moduleName'` and `'className'`

The only limitation is that all classes corresponding to entities in a model should be located within the same package (the one identified by the field `'packageName'` in model's properties).

What if I want to see the primary key values in my objects? Making it a class property Really, you should not –simply because we do not support anything but *automatic* generation of *simple* primary key (one attribute only, no compound primary keys), you shouldn't need to expose the PK values as class' attributes. But, ok, if you really want to do that, that is to say, if you declare them as class properties:

- you must also define a default value of `0` (integer zero)
- please keep in mind that they should be considered READ-ONLY ; if you modify them at run-time, the framework will not even notice that–it just does not expect it–and things will be out-of-control, for sure. You've been warned!

Using `CustomObject.snapshot_raw()` This method is the best way to get the values that will be stored for an object at a given state in the database when changes are saved. It returns a dictionary from which the value of the primary key can be retrieved. Be sure to carefully read its documentation before using it.

This method does not require that the primary key is made a class property.

Accessing it at run-time There's a third alternative: the primary key value can always be retrieved from your objects, without requiring it to be a class property and apart from `snapshot_raw()`. You can access it with `globalID`; every single object managed by the framework is uniquely identified by its global id. Among other things, this one has a dictionary holding the PK values:

```
>>> object=ec.fetch('Writer', qualifier='age<50')[0]
>>> object.globalID().keyValues()
{'id': 1}
```

So if you need to access the PK for objects of a given class, you can add a method like this one:

```
  def pk(self):
    "Returns the pk's value"
    gid=self.globalID()
    if not gid:
      return None
    if gid.isTemporary():
      # temporary gid: object has been inserted but not saved yet.
      # Change this to return any value you find more appropriate
return gid
    return gid.keyValues()['id']
```

Note: `CustomObject.globalID()` can return `None` when an object has just been inserted into an `EditingContext` but has not been saved yet.

What if I want to see the foreign key values in my objects? As a general rule, foreign keys should not be made class properties; this is an even stricter rule than the one for primary keys, because the framework *does not update foreign keys values* in objects that define them as class properties (but of course, they will saved as expected in the database).

The reason is that a foreign key is usually used to store the piece of information needed to store in the database a to-one relationship at the object level. Say you have an entity, `Writer` related with the entity `Book` in a one-to-many relationship; the table for entity `Book` stores the primary key value for the corresponding `Writer` in a foreign key `FK_Writer_id`. Suppose now that a book is moved from one author to another: at the object-level, the book is removed to one author's set of books and added to the others, while the book itself gets a new author:

```
>>> writer1.removeFromBooks(book)
>>> book.setAuthor(writer2)
>>> writer2.addToBooks(book)
```

If the foreign key `Book.FK_Writer_id` is a class property, it is now out of sync with the book's author, because it stores the former author's id while it has been assigned to an other one.

However, there are several alternate solutions for accessing a foreign key value, if you really insist on doing this:

Using CustomObject.snapshot_raw() This particular method was especially design for that purpose, as it returns the raw data that do/will represent the corresponding database row. As far as foreign keys are concerned, it tries everything possible to return a foreign key value reflecting the current state of the graph of object. There are, however, situations where the returned value can get out-of-sync; please refer to the documentation of `CustomObject.snapshot_raw()` for details.

Explicitely extracting the related objects's primary key You can also do it manually, by traversing the relationship to get its primary key, for example by adding such a method to your `Book`:

```
 def getFKWriterId(self):
   if self.getWriter() is None:
     return None
   else:
     w_gid=self.getWriter().globalID()
     if w_gid.isTemporary():
       # temporary gid: object has been inserted but not saved yet.
       # Change this to return any value you find more appropriate
       return w_gid
     else:
       return w_gid.keyValues()
```

This code is, in fact, the relevant part of the code of `CustomObject.snapshot_raw()`. It's exposed here so that you can see the different cases that can happen. Of course, it assumes that a to-one relationship for this foreign key is defined in the object's entity –if the FK is only involved in an other entity's to-many relationship with no inverse, there's little to do, except relying on `CustomObject.snapshot_raw()` (see above) that can also retrieve the correct FK value after the first fetch, and after the EditingContext saved its changes. Please refer to its documentation for full details.