Chapter 7: Reflexive and Optional One-to-many Relationships

The objective of this chapter is to explain reflexive and optional one-to-many relationships. A reflexive relationship is defined on a single concept. A reflexive one-to-many relationship is a parent-child relationship between entities of the same concept. For example, a category may have from 0 to N sub-categories, and a category may have a super-category. A root category is a category that does not have a super-category. There may be more than one root category. Other categories must have a super-category, since they are organized in a hierarchy (tree) of categories and sub-categories of several levels. There is no restriction on the number of tree levels. A leaf category is a category that does not have any sub-categories. The reflexive relationship is internal. In an XML data file, all categories and sub-categories are saved in the same XML data file. The reflexive concept should be an entry point into the domain model.

A reflexive relationship is at the same time an optional relationship, since the child-parent relationship direction has the minimal cardinality of 0 (since the root category does not have a super-category). A one-to-many relationship between two concepts (or one concept in the case of a reflexive relationship) is optional if there is at least one child entity without a parent. The two concepts of an optional one-to-many relationships should be both entry points into the domain model and the relationship must be external. In the XML persistency, both concepts have their own XML data files and the child concept has the reference property to the parent entity, if the parent exists. The value of the reference property is the parent's oid.

Domain Model

There is a new reflexive relationship defined on the Category concept (Figure 7.1). The relationship is one-to-many. The parent-child direction (or the child neighbor) has 0..N cardinalities, and the child-parent direction (or the parent neighbor) has 0..1 cardinalities. The child neighbor is named categories. The parent neighbor is called category. There are no new properties in the Category concept.

There is a new concept called Question. A question has a number (1, 2, 3, ...), a type (true/false, multiple choice, ...), its text, a response, a creation date (creationDate, and the number of points. Besides oid, only the number (id), text and creationDate values are required. A question may be optionally categorized. The creation date has a default value, a date at the time of the question's creation. There is also a default value for points, but the default value may be erased by a user of the web application. The question number is auto incremented. In the ModelibraWicket application, default values are presented before a user enters new data. However, the auto increment value can be seen only after the new entity is added. The value cannot be updated, since the update element of the property is defined as false.

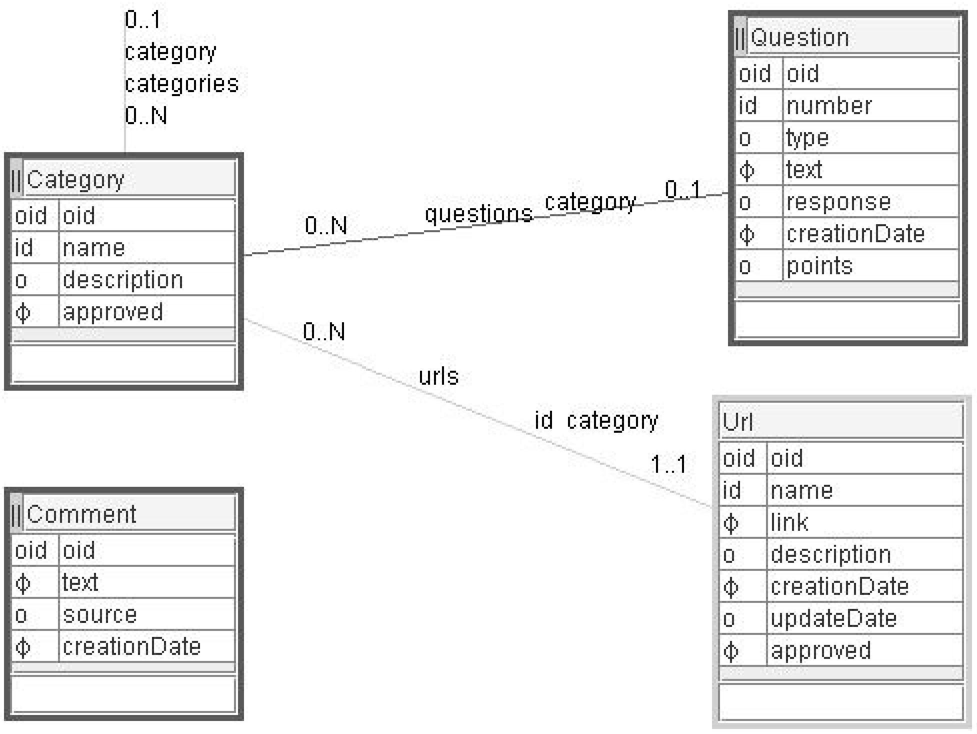


Figure 7.1. Optional one-to-many relationships

ModelibraModeler

Before the code can be generated, the domain model for the project must be designed with ModelibraModeler. Figure 7.1 represents this domain model.

By default, a concept is not entry. It is changed to entry by invoking a pop-up menu on the name of the concept and selecting the *entry* check box. A new one-to-many relationship is created by selecting the line icon, then by clicking first on the name of the parent concept, second on the name of the child concept. For a reflexive relationship both clicks are done on the name of the concept. A minimal cardinality may be changed from 1 to 0 by a pop-up menu of the concept and the choice of the second *Directions*, or by the *Dictionary* menu in the diagram window. The darker relationship between the Category and Question concepts indicates the external relationship. By default, a relationship is internal. It is changed to the external relationship by invoking a pop-up menu on the line and un-selecting the *internal* check box (Figure 7.2). Another way to change definitions of concepts and relationships is to use the *Dictionary* menu in the diagram window.

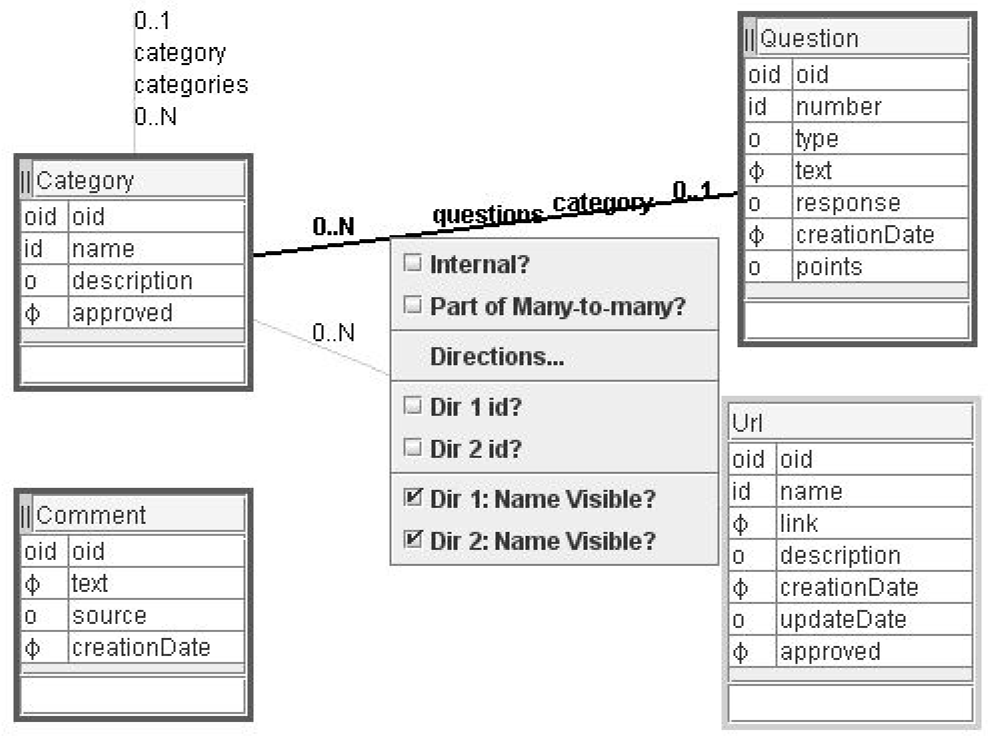


Figure 7.2. Line pop-up menu in ModelibraModeler

The mm directory of the project is used to save the model and to export types and the diagram (*Transfer* menu in the application window). It is a good practice to export the model as text files in order to be able to import the model (types first, then the diagram), if, with the new version of ModelibraModeler, the binary file of the model cannot be opened.

When the model is completely designed its minimal XML configuration is generated by the use of the *Generation* menu. The configuration is saved in the WEB-INF/config directory of the project by replacing the existing reusable-domain-config.xml file. The project is selected within Eclipse and refreshed (*File/Refresh*) to get the new version of the configuration. It is a good practice to examine the XML configuration before the code is generated within Eclipse.

Domain Configuration

The code generation facility of Modelibra has been used in this chapter. There are now two XML configuration files: reusable-domain-config.xml and specific-domain-config.xml. The reusable domain configuration is generated from ModelibraModeler. The specific domain configuration is used to customize the reusable configuration.

In the reusable-domain-config.xml, a declaration of the new Question concept may be found. The Category concept's definition has three new neighbors: category, categories and questions.

```
<concept oid="1171894920503">
     <code>Category</code>
     <entitiesCode>Categories</entitiesCode>
```

```
<entry>true</entry>

<properties>
      ...
</properties>

<neighbors>
      <neighbor oid="1171894920524">
            <code>urls</code>
            <destinationConcept>Url</destinationConcept>
            <inverseNeighbor>category</inverseNeighbor>
            <internal>true</internal>
            <partOfManyToMany>false</partOfManyToMany>
            <type>child</type>
            <min>0</min>
            <max>N</max>
      </neighbor>
      <neighbor oid="1171894961806">
            <code>categories</code>
            <destinationConcept>
                  Category
            </destinationConcept>
            <inverseNeighbor>category</inverseNeighbor>
            <internal>true</internal>
            <partOfManyToMany>false</partOfManyToMany>
            <type>child</type>
            <min>0</min>
            <max>N</max>
      </neighbor>
      <neighbor oid="1171896759590">
            <code>questions</code>
            <destinationConcept>
                  Question
            </destinationConcept>
            <inverseNeighbor>category</inverseNeighbor>
            <internal>false</internal>
            <partOfManyToMany>false</partOfManyToMany>
            <type>child</type>
            <min>0</min>
            <max>N</max>
      </neighbor>
      <neighbor oid="1171894961806">
            <code>category</code>
            <destinationConcept>
                  Category
            </destinationConcept>
            <inverseNeighbor>
                  categories
            </inverseNeighbor>
            <internal>true</internal>
            <partOfManyToMany>false</partOfManyToMany>
            <type>parent</type>
```

```
                <min>0</min>
                <max>1</max>
            </neighbor>
        </neighbors>

    </concept>

    <concept oid="1171896744338">
        <code>Question</code>
        <entitiesCode>Questions</entitiesCode>
        <entry>true</entry>

        <properties>
            <property oid="1171896759590">
                <code>categoryOid</code>
                <propertyClass>
                    java.lang.Long
                </propertyClass>
                <reference>true</reference>
                <referenceNeighbor>
                    category
                </referenceNeighbor>

                <essential>false</essential>
                <referenceDropDownLookup>
                    true
                </referenceDropDownLookup>
            </property>
            <property oid="1171896834670">
                <code>number</code>
                <propertyClass>
                    java.lang.Integer
                </propertyClass>
                <required>true</required>
                <autoIncrement>true</autoIncrement>
                <unique>true</unique>

                <essential>true</essential>
            </property>
            <property oid="1171896840936">
                <code>type</code>
                <propertyClass>
                    java.lang.String
                </propertyClass>
                <maxLength>32</maxLength>

                <essential>false</essential>
            </property>
            <property oid="1171896881469">
                <code>text</code>
                <propertyClass>
                    java.lang.String
        </propertyClass>
        <maxLength>510</maxLength>
        <required>true</required>

        <essential>true</essential>
    </property>
    <property oid="1171896883955">
        <code>response</code>
        <propertyClass>
            java.lang.String
        </propertyClass>
        <maxLength>1020</maxLength>

        <essential>false</essential>
    </property>
    <property oid="1171896850484">
        <code>creationDate</code>
        <propertyClass>
            java.util.Date
        </propertyClass>
        <maxLength>16</maxLength>
        <required>true</required>
        <defaultValue>today</defaultValue>

        <essential>false</essential>
    </property>
    <property oid="1171897384346">
        <code>points</code>
        <propertyClass>
            java.lang.Float
        </propertyClass>
        <defaultValue>0.5</defaultValue>

        <essential>false</essential>
    </property>
</properties>

<neighbors>
    <neighbor oid="1171896759590">
        <code>category</code>
        <destinationConcept>
            Category
        </destinationConcept>
        <inverseNeighbor>questions</inverseNeighbor>
        <internal>false</internal>
        <partOfManyToMany>false</partOfManyToMany>
        <type>parent</type>
        <min>0</min>
        <max>1</max>
    </neighbor>
</neighbors>
```

```
                                </concept>
```

The Question concept has a new property called categoryOid as a reference to its category neighbor. The class of the reference property is java.lang.Long, which defines the oid unique number. By default, ModelibraWicket uses the drop down choice component to lookup a category for a question. Since the drop down choice component displays only the root level categories, this default value is changed in the specific configuration to allow for a lookup of any category in a hierarchy of categories presented in a hierarchy of web pages. In future, to be more user friendly, the hierarchy of categories may be displayed in the same web page as a tree component. The specific configuration of the WebLink model extends the reusable configuration model of the same name. Similarly, concepts and properties inherit their reusable counterparts. The reusable XML elements may be overridden by providing a new value in the specific configuration. Also, a new declaration may be added to specialize a property. The same can be done with a model and a concept. Note that the domain cannot be extended.

```
<?xml version="1.0" encoding="UTF-8"?>

<domains>

     <domain oid="1189989374359">
          <code>DmEduc</code>
          <type>Specific</type>

          <models>

               <model oid="1171894920489">
                    <code>WebLink</code>
                    <extension>true</extension>
                    <extensionDomain>DmEduc</extensionDomain>
                    <extensionDomainType>Reusable</extensionDomainType>
                    <extensionModel>WebLink</extensionModel>

                    <concepts>

                         ...

                         <concept oid="1171896744338">
                              <code>Question</code>
                              <extension>true</extension>
                              <extensionConcept>Question</extensionConcept>

                              <properties>

                                   <property oid="1171896759590">
                                        <code>categoryOid</code>
                                        <extension>true</extension>
                                        <extensionProperty>
                                             categoryOid
                                        </extensionProperty>

                                        <referenceDropDownLookup>
                                             false
```

```
                                                  </referenceDropDownLookup>
                                          </property>

                                          <property oid="1171896834670">
                                                  <code>number</code>
                                                  <extension>true</extension>
                                                  <extensionProperty>
                                                          number
                                                  </extensionProperty>

                                                  <required>false</required>

                                                  <update>false</update>
                                                  <displayLength>8</displayLength>
                                          </property>

                                                  ...

                                          <property oid="1171897384346">
                                                  <code>points</code>
                                                  <extension>true</extension>
                                                  <extensionProperty>
                                                          points
                                                  </extensionProperty>

                                                  <displayLength>8</displayLength>
                                          </property>

                                  </properties>

                          </concept>

                  </concepts>

          </model>

      </models>

  </domain>

</domains>
```

The number property is required (and essential), autoIncrement and unique in the generated reusable configuration. However, in the specific configuration the number property is not required and its update declaration is false. In addition, its display length is 8.

The points property is of the java.lang.Float class and its default value is 0.5. It is specialized by adding a display length of 8.

Note that the category neighbor is **not** internal (it is external) and it is not a part of a many-to-many relationship. A many-to-many relationship will be introduced in the next chapter. The minimal cardinality is 0, enabling a question to be without its category.

The Category concept has four neighbors: urls, category, categories and questions. Only the questions neighbor is external.

Project Skeleton

ModelibraWicket has an Eclipse skeleton project, called ModelibraWicketSkeleton, which is used as the starting point in programming a new domain model and its Wicket application. The template can be checked out or downloaded from the Modelibra's repository site [Modelibra]. The following is the directory structure of the project skeleton.

```
ModelibraWicketSkeleton
     src
          dm
               gen
     test
     css
          img
     doc
     mm
     template
          domain
               model
                    concept
               wicket
                    app
                    model
                         concept
          web
     WEB-INF
          config
          lib
```

We have to prepare now a new Eclipse project so that the code can be generated. The new project is prepared from the ModelibraWicketSkeleton project. A way to do it is to export the source project without the css, template, classes, lib and logs directories to a new location. This is done in Eclipse by using *File/Export.../General/File System*. We will see that there are neither SVN directories nor SVN files in the exported project. The .project file is then opened with a text editor and the project name is changed to DmEduc-05. Finally, the new project is imported into Eclipse by *File/Import.../General/Existing Projects* into a Workspace.

The DmEduc-05 project is then transferred to the SVN repository. This is done by selecting the project and using the *Team/Share Project...* pop-up menu item. If there is no need to share the project with other people, the initial step of exporting the source project should have included the css, template, classes, lib and logs directories.

In the first transfer of files to the SVN repository, the classes directory is excluded from the transfer by unchecking the check box for the classes directory. The svn:ignore property is defined at the WEB-INF directory for the classes and logs directories. This is done by selecting the WEB-INF directory and using the *Team/Set Property...* pop-up menu item. As a consequence, both the classes and logs directories will not be transferred to the repository in future commits. In addition, the svn:externals property is defined at the WEB-INF directory for the lib directory.

lib http://svn.javaforge.com/svn/Modelibra/trunk/ModelibraWicketSkeleton/WEB-INF/lib

Those two SVN properties are committed to the repository by the *Team/Commit...* pop-up menu item. After that, the update of the local project version (*Team/Update*) by the repository version loads the library files to the lib directory from the ModelibraWicketSkeleton project. In future, if the libraries in the ModelibraWicketSkeleton project change, the corresponding libraries in the local project will be updated whenever the update of the local project version is done.

Similarly, the svn:externals property is defined at the project root directory for the css and template directories.

css http://svn.javaforge.com/svn/Modelibra/trunk/ModelibraWicketCss/css
template http://svn.javaforge.com/svn/Modelibra/trunk/ModelibraWicketSkeleton/template

The local changes are committed to the repository by *Team/Commit...* The local version is then updated by *Team/Update* and the css and template directores with their content are loaded from the ModelibraWicketCss and ModelibraWicketSkeleton projects. In this way the local project is in sync with future changes of the corresponding external projects. Of course, if those changes are not wanted, the svn:externals property is not used, and the copy of corresponding directories and files is done once at the creation of the local project.

Code Generation

Once the new project is prepared from the project skeleton, the domain model is placed in the mm directory of the project, and the reusable configuration is generated from ModelibraModeler, all is ready to generate the code for the domain model and its default web application. Once the code is generated, the specific code from the previous spiral will be transferred to the DmEduc-05 spiral.

The code generators are ready in the predefined dm.gen package. The dm.gen package has four classes: DmConfig.java, DmGenerator, DmModelibraGenerator, DmModelibraWicketGenerator. The configuration class enables access to model definitions that come from the XML configuration. The DmConfig class is simple. Its constructor is able to get the domain configuration from the the domain code and its type.

```
package dm.gen;

import org.modelibra.config.Config;
import org.modelibra.config.DomainConfig;

public class DmConfig extends Config {

	private DomainConfig domainConfig;
	public DmConfig(String domainCode, String domainType) {
		super();
		domainConfig = getDomainConfig(domainCode, domainType);
	}

	public DomainConfig getDomainConfig() {
		return domainConfig;
	}

}
```

The DmGenerator class has a constant that represents the domain code. The DOMAIN_CODE constant must be changed to reflect the new project's domain name. The code generator class delegates its work to two generator classes, one for the the domain model and the other for the default web application.

```
package dm.gen;

import org.modelibra.config.DomainConfig;

public class DmGenerator {

	public static final String DOMAIN_CODE = "DmEduc";

	public static final String DOMAIN_TYPE = "Reusable";

	private DmModelibraGenerator dmModelibraGenerator;

	private DmModelibraWicketGenerator dmModelibraWicketGenerator;

	public DmGenerator() {
		dmModelibraGenerator = new DmModelibraGenerator(DOMAIN_CODE, DOMAIN_TYPE);

		DomainConfig domainConfig = dmModelibraGenerator.getDomainConfig();
		String authors = dmModelibraGenerator.getAuthors();
		String codeDirectoryPath = dmModelibraGenerator.getCodeDirectoryPath();

		dmModelibraWicketGenerator = new DmModelibraWicketGenerator(
				domainConfig, authors, codeDirectoryPath);
	}

	public DmModelibraGenerator getDmModelibraGenerator() {
		return dmModelibraGenerator;
	}

	public DmModelibraWicketGenerator getDmModelibraWicketGenerator() {
		return dmModelibraWicketGenerator;
	}

	public static void main(String[] args) {
		DmGenerator dmGenerator = new DmGenerator();
		// *** Modelibra ***
		dmGenerator.getDmModelibraGenerator().generate();

		// *** ModelibraWicket ***
		dmGenerator.getDmModelibraWicketGenerator().generate();
	}

}
```

In the constructor, both Modelibra and ModelibraWicket generators are created. In the main method, the generate method of the DmModelibraGenerator and DmModelibraWicketGenerator classes are invoked to generate a working web application. The source code is generated in the src directory of the project. The XML data files without data are generated in the WEB-INF directory. The web application configuration is generated in the WEB-INF directory as well.

The code is never generated once. It is for sure that the initial model will change and the need to regenerate the code without loosing specific changes is ever present. In order to regenerate only the classes that start with the Gen keyword, both generate methods may be commented and the generateModelibraGenClasses and generateModelibraWicketAppProperties methods may be uncommented.

```
	public static void main(String[] args) {
		DmGenerator dmGenerator = new DmGenerator();

		// *** Modelibra ***

		// dmGenerator.getDmModelibraGenerator().generate();

		dmGenerator.getDmModelibraGenerator().generateModelibraGenClasses();

		// dmGenerator.getDmModelibraGenerator().generateModelibraPartially();

		// *** ModelibraWicket ***

		// dmGenerator.getDmModelibraWicketGenerator().generate();

		dmGenerator.getDmModelibraWicketGenerator()
			.generateModelibraWicketAppProperties();

		// dmGenerator.getDmModelibraWicketGenerator()
		//	.generateModelibraWicketPartially();
	}
```

If some specific code has been added and some data saved in XML files, the generate method can be safely used again only if there is a backup for the specific code and the data.

A programmer is encouraged to examine carefully both Modelibra and ModelibraWicket generators, especially those methods that permit a partial generation. With different code generation methods they are quite flexible to allow a continuum of code generation from the complete generation to some very specific generation. The code generation is designed in such a way that a developer, after the initial learning effort, may become very productive in starting a new project.

The code generation is done by using Velocity [Velocity]. Velocity is a free open-source Java based template engine. A simple template language references objects defined in Java code. The following is an example of the template used in the code generation of the specific domain class. Placeholders that start with the $ sign are replaced by String values from objects defined in Java code of the generator in question.

```
#include ("template/ModelibraComment.txt")

package $domainpackagecode;

import org.modelibra.config.DomainConfig;

/**
 * Creates domain and its models.
 *
 * @author $author
 * @version $today
 */
public class $DomainCode extends Gen$DomainCode {

	private static final long serialVersionUID = ${serialNumber}L;

	/**
	 * Creates the $DomainCode domain.
	 *
	 * @param domainConfig
	 *            domain configuration
	 */
	public $DomainCode(DomainConfig domainConfig) {
		super(domainConfig);
	}

	/* ============================ */
	/* ======= SPECIFIC CODE ======= */
	/* ============================ */

}
```

Compare the template with the generated code for the specific domain class.

```
/*
 * Modelibra
 *
 * Licensed under the Apache License, Version 2.0 (the "License"); you may not
 * use this file except in compliance with the License. You may obtain a copy of
 * the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS, WITHOUT
 * WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the
 * License for the specific language governing permissions and limitations under
 * the License.
 */
package dmeduc;

import org.modelibra.config.DomainConfig;

/**
 * Creates domain and its models.
 *
 * @author Dzenan Ridjanovic
 * @version 2007-10-23
 */
public class DmEduc extends GenDmEduc {

	private static final long serialVersionUID = 1189989374360L;

	/**
	 * Creates the DmEduc domain.
	 *
	 * @param domainConfig
	 *            domain configuration
	 */
	public DmEduc(DomainConfig domainConfig) {
		super(domainConfig);
	}

	/* ========================== */
	/* ======= SPECIFIC CODE ======= */
	/* ========================== */

}
```

Since Velocity has a strange way of handling spaces, the generated code, especially for more elaborate classes such as generic classes for concepts, often does not look pretty. Fortunately, Eclipse has a short-cut to format the code. Keep with the left hand the *Ctrl-Shift* keys pressed and with the right hand use the *O F S* keys in sequence to arrange the generated code. The imports will be cleaned (*O*), the code will be nicely formatted (*F*) and the changes will be saved (*S*).

Velocity templates for Modelibra and ModelibraWicket are located in the template directory of the project. You are not supposed to change those templates.

The following shows directories of the generated project.

```
DmEduc-05
	src
		dm
			gen
```

```
        dmeduc
            weblink
                category
                comment
                question
                url
        wicket
            app
            weblink
                category
                comment
                question
                url
    test
    css
    doc
    mm
    template
    WEB-INF
        config
        data
            xml
                dmeduc
                    weblink
        lib
        logs
```

The Java source code is generated in packages. There is a package for the domain. There is a package for the model. For each concept, at the model level, there is a package with four classes, two for the concept entity and two for the concept entities. There is one package for the web application. For each concept, at the application view level, there is a package with classes in which the order or selection of the concept entities may be defined as required by the view.

For the DmEduc-05 spiral of this chapter, the package name for the domain is dmeduc. The standard for package names in Java is to use lower letters for all characters. The dmeduc package has four classes.

```
dmeduc
    DmEduc
    DmEducConfig
    GenDmEduc
    PersistentDmEduc
```

The GenDmEduc abstract class contains the generic code that should not be changed by a programmer.

```java
package dmeduc;

import org.modelibra.Domain;
import org.modelibra.config.DomainConfig;

import dmeduc.weblink.WebLink;
```

```
public abstract class GenDmEduc extends Domain {

	private WebLink webLink;

	public GenDmEduc(DomainConfig domainConfig) {
		super(domainConfig);
		webLink = new WebLink(this);
	}

	public WebLink getWebLink() {
		return webLink;
	}

}
```

The DmEduc class extends the GenDmEduc class and it is a place where some specific domain code may be added by a programmer after the SPECIFIC CODE comment. This is done to allow a programmer to regenerate code in the Gen class and keep the specific code intact by not regenerating the specific class.

```
package dmeduc;

import org.modelibra.config.DomainConfig;

public class DmEduc extends GenDmEduc {

	private static final long serialVersionUID = 1168374565313L;

	public DmEduc(DomainConfig domainConfig) {
		super(domainConfig);
	}

	/* ============================ */
	/* ======= SPECIFIC CODE ======= */
	/* ============================ */

}
```

The package name for the model is dmeduc.weblink. The package has two classes

```
dmeduc.weblink
	GenWebLink
	WebLink
```

The GenWebLink abstract class contains the model entry points.

```
package dmeduc.weblink;

import org.modelibra.IDomain;
import org.modelibra.DomainModel;
import dmeduc.weblink.comment.Comments;
import dmeduc.weblink.category.Categories;
import dmeduc.weblink.question.Questions;

public abstract class GenWebLink extends DomainModel {

	private Comments comments;

	private Categories categories;

	private Questions questions;

	public GenWebLink(IDomain domain) {
		super(domain);
		comments = new Comments(this);
		categories = new Categories(this);
		questions = new Questions(this);
	}

	public Comments getComments() {
		return comments;
	}

	public Categories getCategories() {
		return categories;
	}

	public Questions getQuestions() {
		return questions;
	}

}
```

The WebLink class allows for some specific additions, which are not generated, such as the getUrls method. This specific method collects all urls for all categories. The method may be used safely to display all urls in a web page.

```
package dmeduc.weblink;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.modelibra.IDomain;

import dmeduc.weblink.category.Categories;
import dmeduc.weblink.category.Category;
import dmeduc.weblink.url.Url;
import dmeduc.weblink.url.Urls;

public class WebLink extends GenWebLink {

	public WebLink(IDomain domain) {
		super(domain);
	}

	/* =========================== */
	/* ======= SPECIFIC CODE ======= */
	/* =========================== */

	public Urls getUrls() {
		Urls allUrls = null;
		boolean dataLoaded = isInitialized();
		if (dataLoaded) {
			try {
				allUrls = new Urls(this);
				allUrls.setPersistent(false);
				allUrls.setPre(false);
				allUrls.setPost(false);
				Categories categories = getCategories();
				for (Category category : categories) {
					Urls categoryUrls = category.getUrls();
					for (Url url : categoryUrls) {
						allUrls.add(url);
					}
				}
			} catch (Exception e) {
				log.error("Error in WebLink.getUrls: " + e.getMessage());
			} finally {
				allUrls.setPersistent(true);
				allUrls.setPre(true);
				allUrls.setPost(true);
			}
		}
		return allUrls;
	}

}
```

For each concept there are four classes generated in the concept package. For example, for the Category concept, in the dmeduc.weblink.category package, there are two generic classes, GenCategory and GenCategories, and two specific classes, Category and Categories. Note that only specific classes will be used in the application programming.

Reflexive One-to-many Relationship

The reflexive one-to-many relationship on the Category concept is handled similarly to regular internal one-to-many relationships. In this chapter, only the code related to this reflexive relationship is shown in the GenCategory and GenCategories classes.

The GenCategory class, presented here only with code for the reflexive relationship, has the category neighbor for the reflexive relationship parent and the categories neighbor for the reflexive relationship child. For a root category, the constructor with the model argument should be invoked. In this constructor the empty child neighbor is created. The constructor with the parent category argument is used to create a new category with the given parent. The **this** keyword calls the constructor with the model argument. The parent category of the new category is set by the setCategory method. The set and get methods for the internal parent neighbor and the internal child neighbor of the reflexive relationship are the standard methods, as all other methods in this class.

```
package dmeduc.weblink.category;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.modelibra.Entity;
import org.modelibra.IDomainModel;

import dmeduc.weblink.WebLink;
import dmeduc.weblink.question.Questions;
import dmeduc.weblink.url.Urls;

public abstract class GenCategory extends Entity<Category> {

...

	/* ======= internal parent neighbors ======= */

	private Category category;

	/* ======= internal child neighbors ======= */

	private Categories categories;

	/* ======= base constructor ======= */

	public GenCategory(IDomainModel model) {
		super(model);
		// internal child neighbors only
		setCategories(new Categories((Category) this));
	}

	/* ======= parent argument(s) constructor ======= */

	public GenCategory(Category category) {
		this(category.getModel());
		// parents
		setCategory(category);
	}

	/* ======= internal parent set and get methods ======= */

	public void setCategory(Category category) {
		this.category = category;
	}
	public Category getCategory() {
		return category;
	}

	/* ======= internal child set and get methods ======= */

	public void setCategories(Categories categories) {
		this.categories = categories;
		if (categories != null) {
			categories.setCategory((Category) this);
		}
	}

	public Categories getCategories() {
		return categories;
	}

...

}
```

The GenCategories class has the category neighbor for the parent of the reflexive relationship. The only non-standard method in this class is the getReflexiveCategory method, which retrieves a category in the reflexive hierarchy using the given oid unique number. If a category is not found, **null** is returned. The method uses recursion to traverse the tree of categories and sub-categories.

```
package dmeduc.weblink.category;

import java.util.Comparator;
import java.util.List;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.modelibra.Entities;
import org.modelibra.IDomainModel;
import org.modelibra.ISelector;
import org.modelibra.Oid;
import org.modelibra.PropertySelector;

public abstract class GenCategories extends Entities<Category> {

...

	/* ====== internal parent neighbors ======= */

	private Category category;

	/* ====== base constructor ======= */

	public GenCategories(IDomainModel model) {
		super(model);
	}

	/* ======= parent argument constructors ======= */

	public GenCategories(Category category) {
		this(category.getModel());
		// parent
		setCategory(category);
	}

	/* ======= get entity methods based on a property ======= */

	public Category getReflexiveCategory(Long oidUniqueNumber) {
		Category category = getCategory(oidUniqueNumber);
		if (category == null) {
			for (Category currentCategory : this) {
				category = currentCategory.getCategories()
						.getReflexiveCategory(oidUniqueNumber);
				if (category != null) {
					break;
				}
			}
		}
		return category;
	}

	/* ======= internal parent set and get methods ======= */

	public void setCategory(Category category) {
		this.category = category;
	}

	public Category getCategory() {
		return category;
	}

	/* ======= create entity method ======= */

	public Category createCategory(Category categoryParent, String name) {
		Category category = new Category(getModel());
		category.setCategory(categoryParent);
		category.setName(name);
		if (!add(category)) {
			category = null;
		}
		return category;
	}

}
```

Optional One-to-many Relationship

The optional one-to-many relationship between the Category and Question concepts is an external relationship and it is handled differently from internal one-to-many relationships. Only the code related to this optional relationship is shown.

The GenCategory class has the questions neighbor. Since the neighbor is external, it is not initialized in the base constructor. There is nothing related to the external neighbor in both constructors. The set method is standard. However, the get method has additional code. When questions are requested for the first time for a category, they are **null** since they have not been initialized in the base constructor. They are derived from the model starting with the Questions entry point. First, a subset of entry questions corresponding to the current category is determined. Second, this subset is passed as the argument to the setQuestions method. The next time, when the category questions are requested, the getQuestions method will return the questions neighbor. There is no code related to questions in the GenCategories class.

```
package dmeduc.weblink.category;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.modelibra.Entity;
import org.modelibra.IDomainModel;

import dmeduc.weblink.WebLink;
import dmeduc.weblink.question.Questions;
import dmeduc.weblink.url.Urls;

public abstract class GenCategory extends Entity<Category> {

	/* ======= external child neighbors ======= */

	private Questions questions;

	/* ======= base constructor ======= */

	public GenCategory(IDomainModel model) {
		super(model);
		// internal child neighbors only
		setUrls(new Urls((Category) this));
		setCategories(new Categories((Category) this));
	}

	/* ======= parent argument(s) constructor ======= */

	public GenCategory(Category category) {
		this(category.getModel());
		// parents
		setCategory(category);
	}
	/* ======= external one-to-many child set and get methods ======= */

	public void setQuestions(Questions questions) {
		this.questions = questions;
		if (questions != null) {
			questions.setCategory((Category) this);
		}
	}

	public Questions getQuestions() {
		if (questions == null) {
			WebLink webLink = (WebLink) getModel();
			Questions questions = webLink.getQuestions();
			setQuestions(questions.getCategoryQuestions((Category) this));
		}
		return questions;
	}

}
```

The GenQuestion class has regular properties and a reference property that is used to derive questions for the given category. The reference property is called categoryOid and is of the Long class. There is also the category external parent in the GenQuestion class. There is a redundancy in having both the reference property and the external neighbor in the same class. This redundancy is necessary to derive the external neighbor from the value of the reference property. However, as with any redundancy it is crucial to manage the redundancy. This redundancy management makes the code more complicated. Fortunately, this code is generated.

When a category of the question is requested the first time by the getCategory method, it is **null**. Then, the Categories entry point is used to retrieve the requested category based on the categoryOid reference property. When the reference property is set by the setCategoryOid method, the corresponding category is retrieved in the same way. If the reference property is set to **null**, the external parent is set to **null** as well (a question is detached from its category).

```
package dmeduc.weblink.question;

import java.util.Date;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.modelibra.Entity;
import org.modelibra.IDomainModel;

import dmeduc.weblink.WebLink;
import dmeduc.weblink.category.Categories;
import dmeduc.weblink.category.Category;

public abstract class GenQuestion extends Entity<Question> {

	/*
	 * ======= properties except for the inherited code property and reference
 * properties =======
 */

private Integer number;

private String type;

private String text;

private String response;

private Date creationDate;

private Float points;

/* ======= reference properties ======= */

private Long categoryOid;

/* ======= external parent neighbors ======= */

private Category category;

/* ======= base constructor ======= */

public GenQuestion(IDomainModel model) {
      super(model);
      // internal child neighbors only
}

/* ======= parent argument(s) constructor ======= */

public GenQuestion(Category category) {
      this(category.getModel());
      // parents
      setCategory(category);
}

/*
 * ======= property (except for the code property and reference properties)
 * set and get methods =======
 */

public void setNumber(Integer number) {
      this.number = number;
}

public Integer getNumber() {
      return number;
}

public void setType(String type) {
		this.type = type;
	}

	public String getType() {
		return type;
	}

	public void setText(String text) {
		this.text = text;
	}

	public String getText() {
		return text;
	}

	public void setResponse(String response) {
		this.response = response;
	}

	public String getResponse() {
		return response;
	}

	public void setCreationDate(Date creationDate) {
		this.creationDate = creationDate;
	}

	public Date getCreationDate() {
		return creationDate;
	}

	public void setPoints(Float points) {
		this.points = points;
	}

	public Float getPoints() {
		return points;
	}

	/* ======= reference property set and get methods ======= */

	public void setCategoryOid(Long categoryOid) {
		this.categoryOid = categoryOid;
		if (categoryOid != null) {
			WebLink webLink = (WebLink) getModel();
			Categories categories = webLink.getCategories();
			category = categories.getReflexiveCategory(categoryOid);
		} else {
			category = null;
		}
	}
	public Long getCategoryOid() {
		return categoryOid;
	}

	/* ======= external parent set and get methods ======= */

	public void setCategory(Category category) {
		this.category = category;
		if (category != null) {
			categoryOid = category.getOid().getUniqueNumber();
		} else {
			categoryOid = null;
		}
	}

	public Category getCategory() {
		if (category == null) {
			WebLink webLink = (WebLink) getModel();
			Categories categories = webLink.getCategories();
			if (categoryOid != null) {
				category = categories.getReflexiveCategory(categoryOid);
			}
		}
		return category;
	}

}
```

Since, the relationship between the Category and Question concepts is external, questions of a certain category are not saved within that category in the category.xml data file. The Questions concept is an entry point into the model and all questions are saved in the question.xml data file.

```
<question oid="1193247606925">
    <categoryOid>1193171173850</categoryOid>
    <number>2</number>
    <text>How heavy is Modelibra in pounds?</text>
    <creationDate>2007-10-24</creationDate>
    <points>0.5</points>
</question>
```

The GenQuestions class has the category external parent with the standard set and get methods. The reference property has the property selection methods in order to obtain easily questions for the given category, given as an object of either the Long, Oid or Category class.

```
package dmeduc.weblink.question;

import java.util.Comparator;
import java.util.List;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.modelibra.Entities;
import org.modelibra.IDomainModel;
import org.modelibra.ISelector;
import org.modelibra.DomainModel;
import org.modelibra.Oid;
import org.modelibra.PropertySelector;

import dmeduc.weblink.category.Category;

public abstract class GenQuestions extends Entities<Question> {

	/* ======= external parent neighbors ======= */

	private Category category;

	/* ======= base constructor ======= */

	public GenQuestions(IDomainModel model) {
		super(model);
	}

	/* ======= parent argument constructors ======= */

	public GenQuestions(Category category) {
		this(category.getModel());
		// parent
		setCategory(category);
	}

	/*
	 * ======= reference property selector methods for a non many-to-many
	 * concept that has at least one external parent =======
	 */

	public Questions getCategoryQuestions(Long category) {
		PropertySelector propertySelector = new PropertySelector("categoryOid");
	propertySelector.defineEqual(category);
	return getQuestions(propertySelector);
	}

	public Questions getCategoryQuestions(Oid category) {
		return getCategoryQuestions(category.getUniqueNumber());
	}

	public Questions getCategoryQuestions(Category category) {
		return getCategoryQuestions(category.getOid());
	}

	/* ======= external parent set and get methods ======= */

	public void setCategory(Category category) {
		this.category = category;
```

```
}

public Category getCategory() {
	return category;
}

/*
 * ======= for a non many-to-many concept that has at least one external
 * parent: post add propagation =======
 */

protected boolean postAdd(Question question) {
	if (!isPost()) {
		return true;
	}
	boolean post = true;
	if (super.postAdd(question)) {
		DomainModel model = (DomainModel) getModel();
		if (model.isInitialized()) {
			Category category = getCategory();
			if (category == null) {
				Category questionCategory = question.getCategory();
				if (questionCategory != null) {
					if (!questionCategory.getQuestions()
					.contain(question)) {
						questionCategory.getQuestions()
								.setPropagateToSource(false);
						post = questionCategory.getQuestions()
								.add(question);
						questionCategory.getQuestions()
								.setPropagateToSource(true);
					}
				}
			}
		}
	} else {
		post = false;
	}
	return post;
}

/*
 * ======= for a non many-to-many concept that has at least one external
 * parent: post remove propagation =======
 */

protected boolean postRemove(Question question) {
	if (!isPost()) {
		return true;
	}
	boolean post = true;
	if (super.postRemove(question)) {
			Category category = getCategory();
			if (category == null) {
				Category questionCategory = question.getCategory();
				if (questionCategory != null) {
					if (questionCategory.getQuestions().contain(question)) {
						questionCategory.getQuestions().setPropagateToSource(
								false);
						post = questionCategory.getQuestions()
						.remove(question);
						questionCategory.getQuestions().setPropagateToSource(
								true);
					}
				}
			}
		} else {
			post = false;
		}
		return post;
	}

	/*
	 * ======= for a non many-to-many concept that has at least one external
	 * parent: post update propagation =======
	 */

	protected boolean postUpdate(Question beforeQuestion, Question afterQuestion){
		if (!isPost()) {
			return true;
		}
		boolean post = true;
		if (super.postUpdate(beforeQuestion, afterQuestion)) {
			Category category = getCategory();
			if (category == null) {
				Category beforeQuestionCategory = beforeQuestion.getCategory();
				Category afterQuestionCategory = afterQuestion.getCategory();
				if (beforeQuestionCategory == null
						&& afterQuestionCategory != null) {
					// attach
					if (!afterQuestionCategory.getQuestions().contain(
							afterQuestion)) {
						afterQuestionCategory.getQuestions()
								.setPropagateToSource(false);
						post = afterQuestionCategory.getQuestions().add(
								afterQuestion);
						afterQuestionCategory.getQuestions()
								.setPropagateToSource(true);
					}
				} else if (beforeQuestionCategory != null
						&& afterQuestionCategory == null) {
					// detach
					if (beforeQuestionCategory.getQuestions().contain(
							beforeQuestion)) {
```

```
                    beforeQuestionCategory.getQuestions()
                            .setPropagateToSource(false);
                    post = beforeQuestionCategory.getQuestions().remove(
                            beforeQuestion);
                    beforeQuestionCategory.getQuestions()
                            .setPropagateToSource(true);
                }
            } else if (beforeQuestionCategory != null
                    && afterQuestionCategory != null
                    && beforeQuestionCategory != afterQuestionCategory) {
                // detach
                if (beforeQuestionCategory.getQuestions().contain(
                        beforeQuestion)) {
                    beforeQuestionCategory.getQuestions()
                            .setPropagateToSource(false);
                    post = beforeQuestionCategory.getQuestions().remove(
                            beforeQuestion);
                    beforeQuestionCategory.getQuestions()
                            .setPropagateToSource(true);
                }
                // attach
                if (!afterQuestionCategory.getQuestions().contain(
                        afterQuestion)) {
                    afterQuestionCategory.getQuestions()
                            .setPropagateToSource(false);
                    post = afterQuestionCategory.getQuestions().add(
                            afterQuestion);
                    afterQuestionCategory.getQuestions()
                            .setPropagateToSource(true);
                }
            }
        }
    } else {
        post = false;
    }
    return post;
}

/* ======= create entity method ======= */

public Question createQuestion(Category categoryParent, String text) {
    Question question = new Question(getModel());
    question.setCategory(categoryParent);
    question.setText(text);
    if (!add(question)) {
        question = null;
    }
    return question;
}

}
```

The post methods are generated to maintain the redundancy between the entry point questions and questions as children of the parent category. This redundancy exists because the new questions object (different from the entry questions) is created for the parent category when the getQuestions method of the GenCategory class is invoked for the first time.

```
public Questions getQuestions() {
      if (questions == null) {
            WebLink webLink = (WebLink) getModel();
            Questions questions = webLink.getQuestions();
            setQuestions(questions.getCategoryQuestions((Category) this));
      }
      return questions;
}
```

The category questions are created by the getCategoryQuestions method in the GenQuestions class. This method uses the Category class as the argument. There are three overloaded methods, all three with the same name but different argument types. The first method delegates the task of finding category questions to the second method, which in turn delegates the task to the third method. In the third method, the property selector is used to accomplish the task.

```
public Questions getCategoryQuestions(Category category) {
      return getCategoryQuestions(category.getOid());
}

public Questions getCategoryQuestions(Oid category) {
      return getCategoryQuestions(category.getUniqueNumber());
}

public Questions getCategoryQuestions(Long category) {
      PropertySelector propertySelector = new PropertySelector("categoryOid");
      propertySelector.defineEqual(category);
      return getQuestions(propertySelector);
}
```

Since there is a redundancy with respect to questions, it is essential to maintain that redundancy. The redundancy exists only at the level of a collection of entities. Specific entities are not redundant. For example, it may be that there are three questions in the questions as the model entry point. Out of those three questions, two are related to a category. If both questions are under the “Framework” category, they will appear in questions of that category. It is important to realize that those two questions appear as references to the same questions as they are found in the entry questions. If a new question is created in the entry questions, and if that question is also for the “Framework” category, the question (actually the reference to that question) must be added to the questions (children) of the the “Framework” category. Otherwise, the display of questions for the “Framework” category will not show that question. A reader is encouraged to explore the issue of redundancy by commenting the following line

```
post = questionCategory.getQuestions().add(question);
```

in the postAdd method of the GenQuestions class.

Similarly, if the same question is removed from the entry questions, it must also be removed from the questions of its category. However, if the same question is updated, without detaching it from the category or attaching it to a category, there is no need to do any propagations because there is a single object for that question. If the question is detached from the category or/and attached to a category, the redundancy must be maintained by propagating remove and/or add actions.

It is important to understand that a collection of questions either has or does not have a parent category. The entry questions contain all questions, possibly with different parent categories, and the entry questions do not have a parent category. Actually, the parent category is **null**. Other collections of questions do have a parent category. Note that a collection of questions may have a parent category and a single question may have a parent category. Questions in the same collection, which has a parent category, have also the same category as their parent. A question without a parent category in the entry questions appears only in that collection.

There are some additional changes in this DmEduc-05 spiral. The specific code has been added to the specific concept classes (as in the previous spirals). Also, some JUnit tests have been added.

JUnit Tests

The four new JUnit tests have been added to test reflexive and optional one-to-many relationships. Of course, the two new relationships may be put in practice with the web application.

In the CategoriesTest class

```
package dmeduc.weblink.category;

import static org.junit.Assert.assertFalse;
import static org.junit.Assert.assertTrue;

import org.junit.After;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import org.modelibra.util.OutTester;

import dmeduc.DmEducTest;
import dmeduc.weblink.question.Question;
import dmeduc.weblink.question.Questions;

public class CategoriesTest {

	private static Categories categories;

	...

}
```

the softwareFramework method shows how the tree of categories and sub-categories is traversed.

```
@Test
public void softwareFramework() throws Exception {
	OutTester.outputText("=== Test: Software Framework ===");
	Category category = categories.getCategory("name", "Software");
	if (category != null) {
		Categories subcategories = category.getCategories();
		if (!subcategories.isEmpty()) {
			category = subcategories.getCategory("name", "Framework");
			if (category != null) {
				category.output("Framework");
			} else {
				DmEducTest.getSingleton().outputMessage(
						"Framework category does not exist.");
			}
		} else {
			DmEducTest.getSingleton().outputMessage(
					"Software category does not have subcategories.");
		}
	} else {
		DmEducTest.getSingleton().outputMessage(
				"Software category does not exist.");
	}
	assertTrue(categories.getErrors().isEmpty());
}
```

The frameworkQuestion method adds a question for the “Framework” category.

```
@Test
public void frameworkQuestion() throws Exception {
	OutTester.outputText("=== Test: Framework Question ===");
	Category category = categories
			.getReflexiveCategory("name", "Framework");
	if (category != null) {
		Questions questions = category.getQuestions();
		Question question = questions.getQuestion("text",
				"Is framework a hard work?");
		if (question == null) {
			question = questions.createQuestion(category,
					"Is framework a hard work?");
		}
		question.output("Framework Question");
	} else {
		DmEducTest.getSingleton().outputMessage(
				"Framework category does not exist.");
	}
	assertTrue(categories.getErrors().isEmpty());
}
```

In the QuestionsTest class the questionWithoutCategory method adds a simple question with the **null** category.

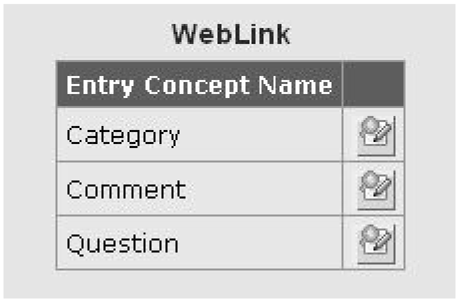
```
@Test
public void questionWithoutCategory() throws Exception {
      OutTester.outputText("=== Test: Question Without Category ===");
      Question question = questions.getQuestion("text", "1+1=?");
      if (question == null) {
           question = questions.createQuestion(null, "1+1=?");
      }
      question.output("Question Without Category");
      assertTrue(questions.getErrors().isEmpty());
}
```

The questionsWithZeroPoints method updates all questions to zero points.

```
@Test
public void questionsWithZeroPoints() throws Exception {
      OutTester.outputText("=== Test: Questions With Zero Points ===");
      for (Question question : questions) {
           Question questionCopy = question.copy();
           questionCopy.setPoints(0f);
           questions.update(question, questionCopy);
      }
      questions.output("Questions With Zero Points");
      assertTrue(questions.getErrors().isEmpty());
}
```

Web Application

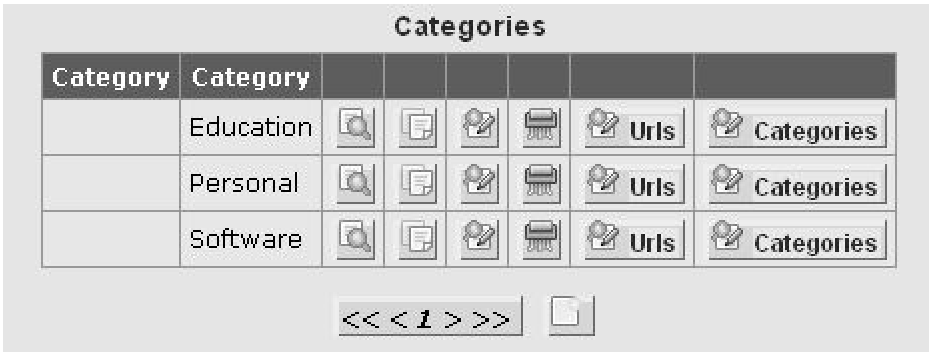
The web application has three model entry concepts: Category, Comment and Question (Fig. 7.3). It is important to realize how reflexive and optional relationships are handled in both update and display mode of the web application. In this section, only the update mode is presented through a few figures. A reader is encouraged to explore display pages as well.


WebLink

Entry Concept Name	
Category	
Comment	
Question	

Figure 7.3. Model entry concepts

The three root categories do not have a parent category, but they have subcategories (Fig. 7.4). Note that Questions do not appear in the update table of categories as children of a category, since the Category – Question relationship is external. However, the questions of a category are present in the display table of categories.

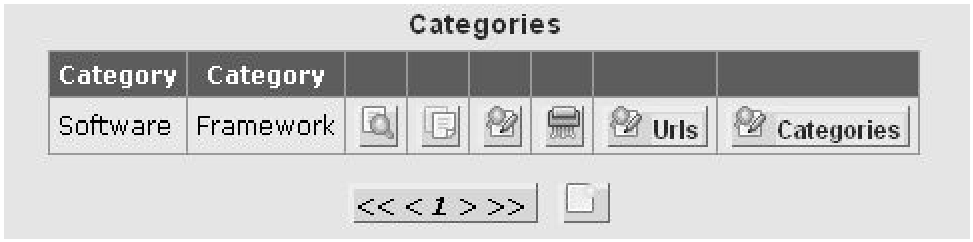

Categories

Category	Category						
	Education					Urls	Categories
	Personal					Urls	Categories
	Software					Urls	Categories

<< < 1 > >>

Figure 7.4. Root categories

The “Software Framework” category can be copied by the new button-like link (a link presented as a button by CSS) positioned just before the update button-like link (Fig. 7.5).

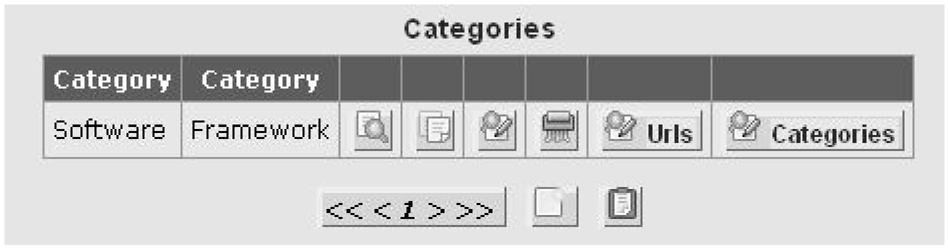

Categories

Category	Category						
Software	Framework					Urls	Categories

<< < 1 > >>

Figure 7.5. Copy button-like link

After the category is copied, the new paste link is displayed to the right of the add link.

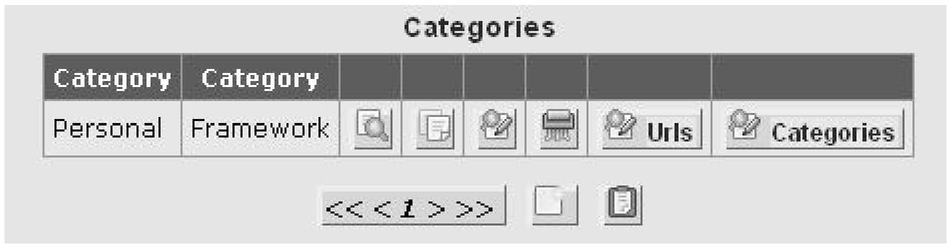

Categories

Category	Category						
Software	Framework					Urls	Categories

<< < 1 > >>

Figure 7.6. Paste button-like link

This link can be used to paste the category to another table of subcategories (Fig. 7.7). In this way the “Framework” category with all its children appears two times in the hierarchy of categories.


Categories

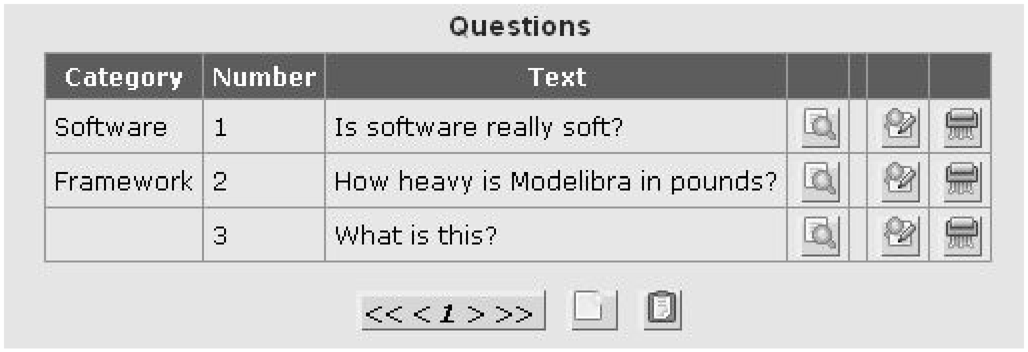
Category	Category						
Personal	Framework					Urls	Categories

<< < 1 > >>

Figure 7.7. “Personal Framework” category

After that, if we delete the “Software Framework” category, we would accomplish the detach-attach function by the copy, paste and delete actions.

There are three questions in Fig. 7.8. Two of them are related to a category, and one is without a category.



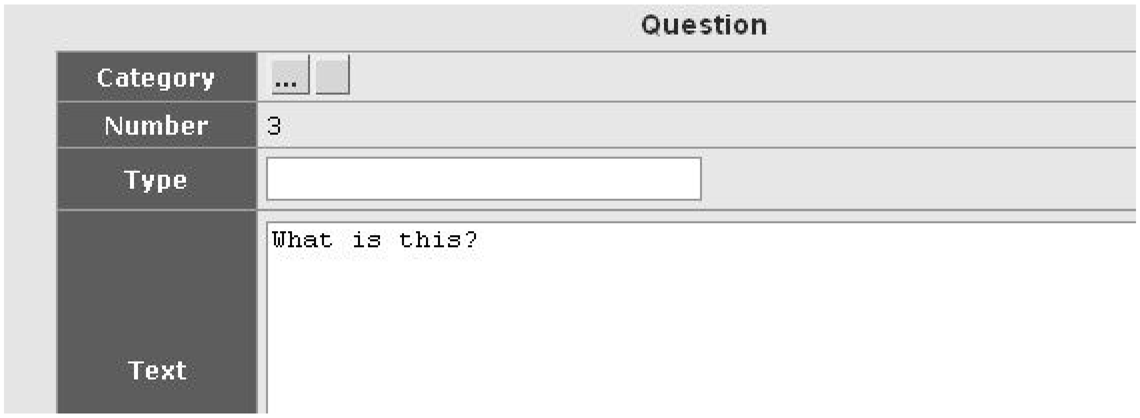
Questions

Category	Number	Text				
Software	1	Is software really soft?				
Framework	2	How heavy is Modelibra in pounds?				
	3	What is this?				

<< < 1 > >>

Figure 7.8. Questions

A question without a category may be updated to reference a category. A question with a category may be updated to be without the category. A new question may be created with or without a category. For example, the question number 3 may be updated by clicking on the link with three points in Fig. 7.9., and by choosing (looking-up) a category. The empty link may be used to detach the question from its category.



Question

Category	...
Number	3
Type	
Text	What is this?

Figure 7.9. Category lookup for the selected question

Modelibra Interfaces

There are no new methods used from the Modelibra interfaces.

Travel Impression

In the new TravelImpression-04 spiral several JUnit tests have been created. The tests, which follow the domain model in Figure 7.10, show how to create a country with one city, a traveler with one message, one note and one photo. The tests could be improved by checking if those objects have already been created.

The TravelImpression-04 spiral is the last spiral of the TravelImpression project presented in the book. A reader is invited in subsequent chapters to create new spirals based on previous spirals to practice his knowledge of Modelibra, ModelibraWicket and Wicket. The TravelImpression project can be considered as an experimental project. Of course, students that really want to learn Modelibra and Wicket are encouraged to start their own projects by designing first a model of the domain that they prefer.

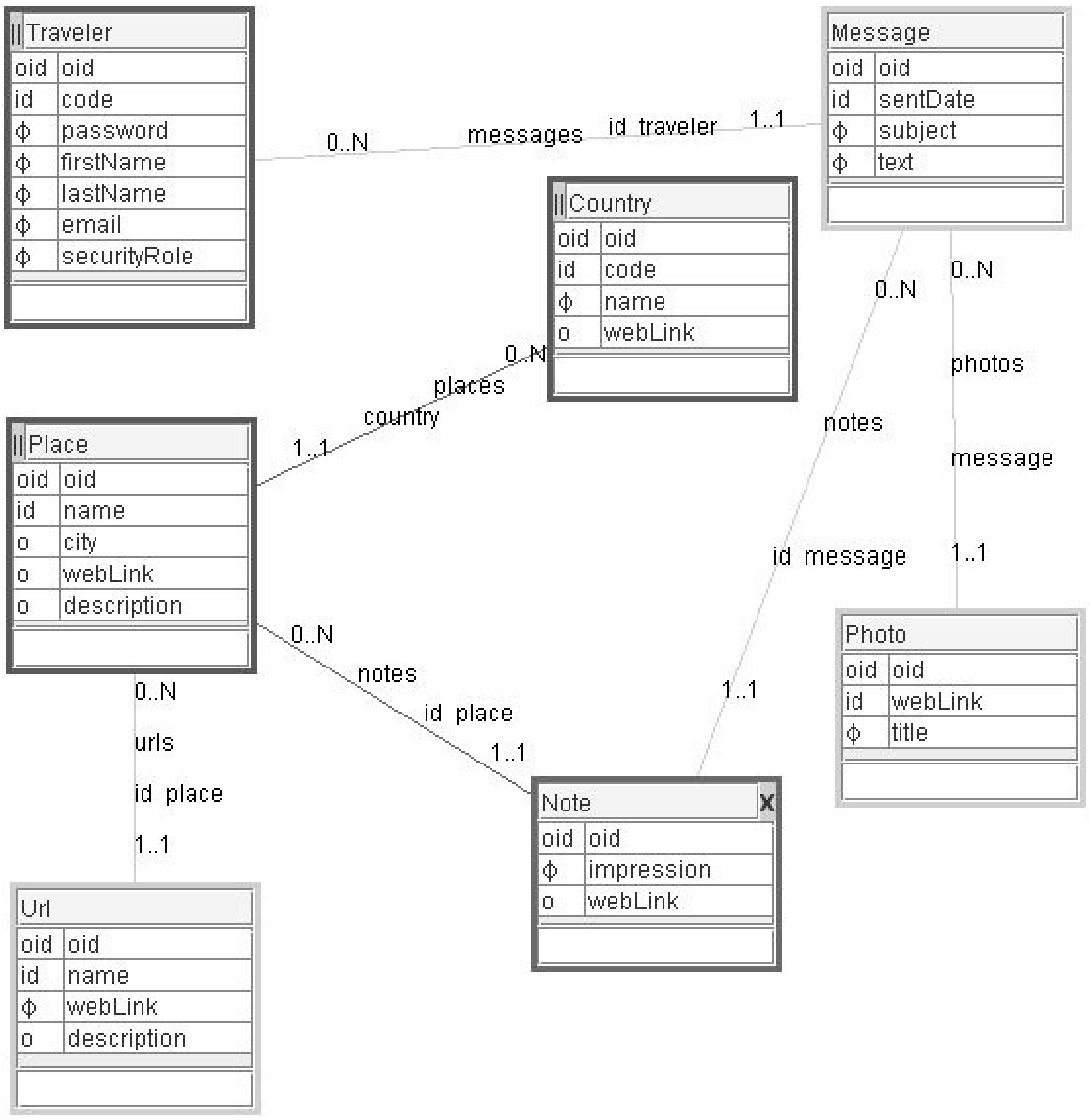


Figure 7.10. Travel Impressions model

```
package travel.impression.country;

import static org.junit.Assert.assertTrue;

import org.junit.After;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import org.modelibra.util.OutTester;

import travel.TravelTest;
import travel.impression.place.Places;

public class CountriesTest {

	private static Countries countries;

	@BeforeClass
	public static void beforeCountries() throws Exception {
		countries = TravelTest.getSingleton().getTravel().getImpression()
				.getCountries();
	}

	@Before
	public void beforeTest() throws Exception {
		countries.getErrors().empty();
	}

	@Test
	public void albaniaCountry() throws Exception {
		OutTester.outputText("=== Test: Albania Country ===");
		countries.createCountry("al", "Albania");
		assertTrue(countries.getErrors().isEmpty());
	}

	@Test
	public void tiranaCity() throws Exception {
		OutTester.outputText("=== Test: Tirana City ===");
		Country country = countries.getCountry("name", "Albania");
		Places places = country.getPlaces();
		places.createPlace(country, "Tirana");
		assertTrue(places.getErrors().isEmpty());
	}

	@After
	public void afterTest() throws Exception {
		countries.getErrors().output("Countries");
	}

}

package travel.impression.traveler;

import static org.junit.Assert.assertTrue;

import org.junit.After;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import org.modelibra.util.OutTester;

import travel.TravelTest;
import travel.impression.country.Countries;
```

```java
import travel.impression.country.Country;
import travel.impression.message.Message;
import travel.impression.message.Messages;
import travel.impression.note.Notes;
import travel.impression.photo.Photos;
import travel.impression.place.Place;
import travel.impression.place.Places;

public class TravelersTest {

	private static Travelers travelers;

	private static Countries countries;

	@BeforeClass
	public static void beforeTravelers() throws Exception {
		travelers = TravelTest.getSingleton().getTravel().getImpression()
				.getTravelers();
		countries = TravelTest.getSingleton().getTravel().getImpression()
				.getCountries();
	}

	@Before
	public void beforeTest() throws Exception {
		travelers.getErrors().empty();
	}

	@Test
	public void mayoTraveler() throws Exception {
		OutTester.outputText("=== Test: Mayo Traveler ===");
		travelers.createTraveler("md", "Mayo", "Dover", "majo@mmcc.com.ba",
				"md");
		assertTrue(travelers.getErrors().isEmpty());
	}

	@Test
	public void mayoMessage() throws Exception {
		OutTester.outputText("=== Test: Mayo Message ===");
		Traveler traveler = travelers.getTraveler("code", "md");
		Messages messages = traveler.getMessages();
		messages.createMessage(traveler, "Going south east",
				"Never been before in Albania.");
		assertTrue(messages.getErrors().isEmpty());
	}

	@Test
	public void mayoNote() throws Exception {
		OutTester.outputText("=== Test: Mayo Note ===");
		Traveler traveler = travelers.getTraveler("code", "md");
		Messages messages = traveler.getMessages();
		Message message = messages.getMessage("subject", "Going south east");
		Notes notes = message.getNotes();
		Country country = countries.getCountry("name", "Albania");
		Places places = country.getPlaces();
		Place place = places.getPlace("name", "Tirana");
		notes.createNote(message, place,
			"Nicer place than I thought it would be.");
		assertTrue(messages.getErrors().isEmpty());
	}

	@Test
	public void mayoPhoto() throws Exception {
		OutTester.outputText("=== Test: Mayo Photo ===");
		Traveler traveler = travelers.getTraveler("code", "md");
		Messages messages = traveler.getMessages();
		Message message = messages.getMessage("subject", "Going south east");
		Photos photos = message.getPhotos();
		photos.createPhoto(message, "Tirana By Night", "http://
www.trekearth.com/gallery/Europe/Albania/North/Tirane/Tirana/photo622899.htm");
		assertTrue(messages.getErrors().isEmpty());
	}

	@After
	public void afterTest() throws Exception {
		travelers.getErrors().output("Travelers");
	}

}
```

country.xml:

```
	<country oid="1194382574247">
		<code>al</code>
		<name>Albania</name>
	</country>
```

place.xml:

```
	<place oid="1194383061076">
		<countryOid>1194382574247</countryOid>
		<name>Tirana</name>
	</place>
```

traveler.xml:

```
	<traveler oid="1194383336531">
		<code>md</code>
		<firstName>Mayo</firstName>
		<lastName>Dover</lastName>
		<email>majo@mmcc.com.ba</email>
		<password>md</password>
		<securityRole>regular</securityRole>
		<messages>
			<message oid="1194383623078">
```

```
                        <sentDate>2007-11-06</sentDate>
                        <subject>Going south east</subject>
                        <text>Never been before in Albania.</text>
                        <notes>
                        <note oid="1194384110236">
                                    <placeOid>1194383061076</placeOid>
                                    <impression>
                                          Nicer place than I thought it would be.
                                    </impression>
                              </note>
                        </notes>
                        <photos>
                        <photo oid="1194384455627">
                                    <title>Tirana By Night</title>
                                    <webLink>
http://www.trekearth.com/gallery/Europe/Albania/North/Tirane/Tirana/photo622899.htm
                                    </webLink>
                        </photo>
                        </photos>
                  </message>
            </messages>
      </traveler>
```

Summary

The reflexive and optional one-to-many relationships, as special cases of the one-to-many type of relationships, are introduced. A reflexive relationship is defined on a single concept. A reflexive one-to-many relationship is a parent-child relationship between entities of the same concept. A one-to-many relationship between two concepts (or one concept in the case of a reflexive relationship) is optional if there is at least one child entity without a parent. The reflexive relationship is internal, while the optional relationship is external.

A domain model may be divided into hierarchical sub-models. The root concept of each hierarchical sub-model is an entry point into the model. All relationships within a hierarchical sub-model are internal. The hierarchical sub-models are related by external relationships. When a domain model is saved, the task of saving the model can be safely decomposed into save actions of hierarchical sub-models (for XML data files, one file per hierarchical sub-model). Since an external relationship is represented as a reference property in the child concept, the task of loading the model can be safely composed of load actions of hierarchical sub-models. After all sub-models are loaded, the domain model becomes complete by deriving internal relationships from the external relationships.

The reflexive one-to-many relationship is defined on the Category concept. The Question concept is introduced to define an optional one-to-many relationship between the Category and Question concepts.

The ModelibraWicketSkeleton project is used as the starting point for the new project. The reusable configuration of the domain model with one reflexive and one optional relationship is generated into the new project, where the code generator is used to create all necessary files for the domain model and the Wicket application based on the domain model. The specific code is added to make the default web application more user friendly. The code is generated in such a way that the specific code will not be lost in future code generations.

The next chapter will explain how a relationship of the many-to-many type is handled both in Modelibra and ModelibraWicket.

Questions

1. What is the restriction for the number of levels in a tree of entities for a reflexive relationship?

2. What is the difference between an internal an external relationship?

3. Can an optional relationship be internal?

4. Can a reflexive relationship be external?

5. Explain why there is a redundancy in an optional relationship?

6. Can you generate the code only for a domain model and all its components?

Exercises

Exercise 7.1.

Make a JUnit test to add a new question without a category. Then update the question to point to an existing category.

Exercise 7.2.

Make a JUnit test to add a new root category with a few subcategories.

Exercise 7.3.

Make a JUnit test to detach a sub-hierarchy of categories and attach them to a new parent.

Exercise 7.4.

Design a new domain model and divide it into hierarchical sub-models in three different ways by using different internal and external relationships.

Exercise 7.5.

For each version of the domain model in the Exercise 7.4, create a new Eclipse project based on the ModelibraWicketSkeleton project. Generate the code and compare the generated code for the three versions.

Web Links

[Modelibra] Modelibra
http://www.javaforge.com/proj/summary.do?proj_id=1647

[Velocity] Velocity
http://velocity.apache.org/