Chapter 8: Many-to-many Relationship

The objective of this chapter is to explain many-to-many relationships. A many-to-many relationship is defined on two (or more) concepts. For example, there is a many-to-many relationship between the Course and Student concepts. A course has many students and a student may be enrolled in several courses. In general, in object oriented technologies, a many-to-many relationship is represented directly between the involved concepts. In relational databases, a many-to-many relationship is implemented through a third table representing participating tables. In Modelibra, a many-to-many relationship is represented with three concepts and two one-to-many relationships. This is done for several reasons. It happens often that there is at least one property specific to the many-to-many relationship. In that case, the third concept cannot be omitted. It is the fact that relational databases will stay around for quite some time. It is a common practice to work with an object model and save objects as table rows. Since one of the objectives of Modelibra is to provide an easy transfer of data from XML files to relational or object databases, representing a many-to-many relationship with three concepts and two one-to-many relationships is a rule.

With at least one many-to-many relationship a domain model becomes a network of concepts (at least one concept has two parent neighbors). In order to save a network model in hierarchically organized XML data files, a network must be decomposed in multiple hierarchies, each starting with an entry concept. This is accomplished by entry concepts as roots of hierarchies and internal one-to-many relationships. A hierarchy of concepts starts with an entry concept and continues with the internal neighbors only. A single hierarchy is saved in a single XML data file. An external relationship is used to reference a concept in another hierarchy (another XML data file). A hierarchy of concepts provides also a natural traversal of entities in views of the model.

Domain Model

There is a new entry concept that represents members of this web application. The Member concept has several properties. The code and password properties will be used in subsequent spirals to allow a member to sign in. The code property is a user oriented id. In addition to oid, the code property is predefined in Modelibra. It is a property of the Entity class. The code property may be ignored and not configured. However, if the code property is used then its configuration must be defined in the XML configuration file. The basic member's properties are firstName and lastName. A member, must have an email, but she may decide to receive or not emails. The default value of the receiveEmail property is false. The beginning of a membership is indicated by the startDate property. Not all members have the same role. The most frequent role is regular. A member with the admin role has all access rights. The access rights will be handled together with the sign in procedure in one of future spirals.

There is a new many-to-many relationship defined on the Category and Member concepts. The third concept, representing the many-to-many relationship (the X sign in the upper right corner of the concept) is Interest. The concept is not entry. It has one specific property called description. There are two one-to-many relationships from the participating concepts to the many-to-many concept. The relationship between Member and Interest is internal (lighter line), while the relationship between Category and Interest is external (darker line). Thus, interests will be saved together with their Member parent in the member.xml data file. The concept id is defined by the member and category neighbors.

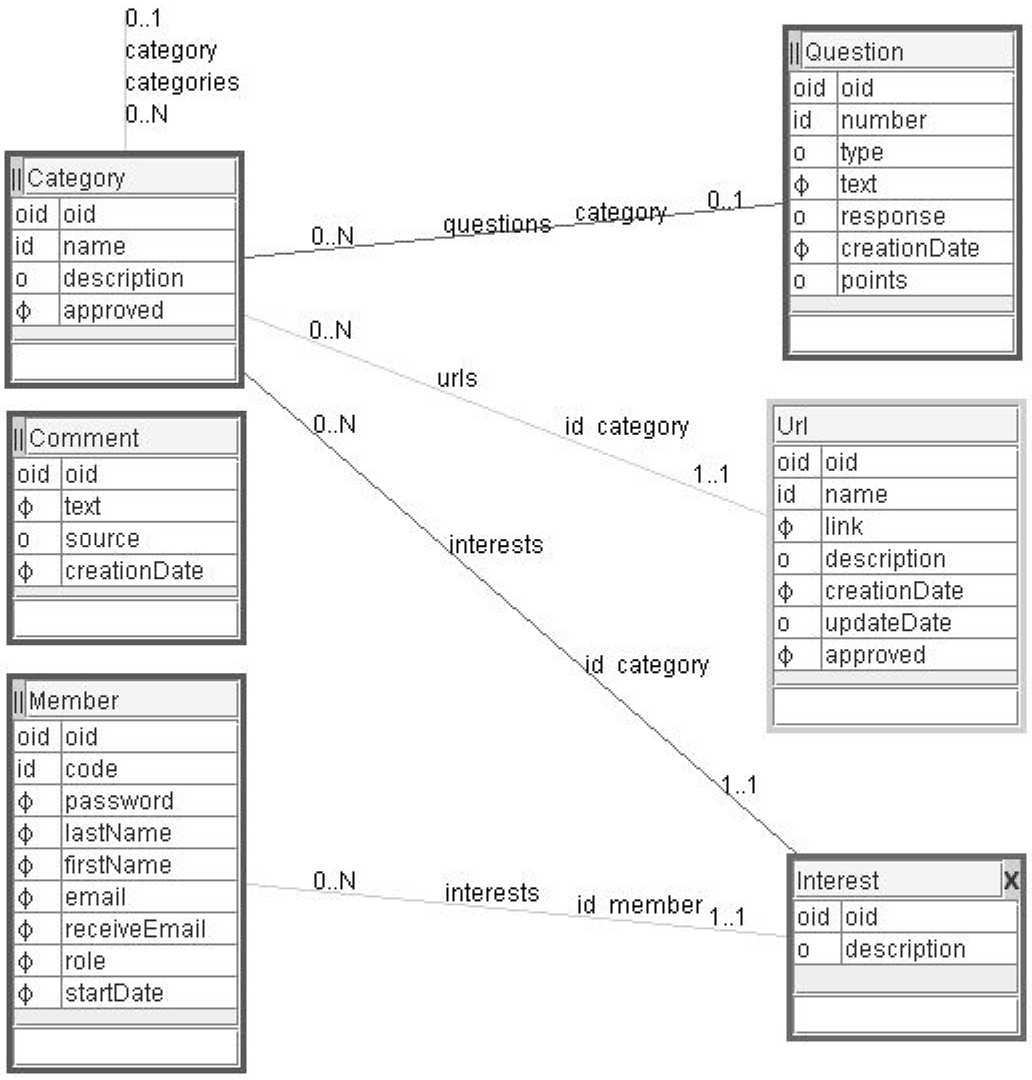


Figure 8.1. Many-to-many relationship

ModelibraModeler

When the Interest concept was created with two parent neighbors, for ModelibraModeler it was an intersection concept between the Category and Member concepts. An intersection concept has at least two parent neighbors, internal by default. As such, ModelibraModeler displays the ? sign in the right corner of the title area. The ? sign begs a designer to change one of the two participating relationships to external. This is done by invoking a pop-up menu on the line and un-selecting the *internal* check box (another way to change definitions of concepts and relationships is to use the *Dictionary* menu in the diagram window.). If only one parent neighbor is internal and all others are external, ModelibraModeler displays the V sign, for the valid intersection concept, in the right corner of the title area. To change the intersection concept to the many-to-many concept, in a pop-up menu on each participating line, the *Part of Many-to-many?* check box must be selected to obtain the X sign in the right section of the title area. If an intersection concept is really the many-to-many concept, it is better to promote it to obtain the X sign. In this way, the generated code will be more precise to handle appropriately the many-to-many relationship.

Domain Configuration

The reusable-domain-config.xml file has been regenerated from ModelibraModeler. There are new Member and Interest concept declarations with property and neighbor elements. The Category concept definition has one new external neighbor: interests.

```
<concept oid="1172170727628">
      <code>Member</code>
      <entitiesCode>Members</entitiesCode>
      <entry>true</entry>

      <properties>
            <property oid="1172170799726">
                  <code>code</code>
                  <propertyClass>
                        java.lang.String
                  </propertyClass>
                  <maxLength>16</maxLength>
                  <required>true</required>
                  <unique>true</unique>

                  <essential>true</essential>
            </property>
            <property oid="1172170801885">
                  <code>password</code>
                  <propertyClass>
                        java.lang.String
                  </propertyClass>
                  <maxLength>16</maxLength>
                  <required>true</required>

                  <essential>false</essential>
                  <scramble>true</scramble>
            </property>
            <property oid="1172170806938">
                  <code>lastName</code>
                  <propertyClass>
                        java.lang.String
                  </propertyClass>
                  <maxLength>32</maxLength>
                  <required>true</required>

                  <essential>true</essential>
            </property>
            <property oid="1172170809433">
                  <code>firstName</code>
                  <propertyClass>
                        java.lang.String
                  </propertyClass>
                  <maxLength>32</maxLength>
                  <required>true</required>

                  <essential>true</essential>
            </property>
            <property oid="1172170826332">
                  <code>email</code>
                  <propertyClass>
                        java.lang.String
                  </propertyClass>
                  <validateType>true</validateType>
                  <validationType>
                        org.modelibra.type.Email
                  </validationType>
                  <maxLength>80</maxLength>
                  <required>true</required>

                  <essential>true</essential>
            </property>
            <property oid="1172170834118">
                  <code>receiveEmail</code>
                  <propertyClass>
                        java.lang.Boolean
                  </propertyClass>
                  <required>true</required>
                  <defaultValue>false</defaultValue>

                  <essential>false</essential>
            </property>
            <property oid="1172170850713">
                  <code>role</code>
                  <propertyClass>
                        java.lang.String
                  </propertyClass>
                  <maxLength>16</maxLength>
                  <required>true</required>
                  <defaultValue>regular</defaultValue>

                  <essential>false</essential>
            </property>
            <property oid="1172170853960">
                  <code>startDate</code>
                  <propertyClass>
                        java.util.Date
                  </propertyClass>
                  <maxLength>16</maxLength>
                  <required>true</required>
                  <defaultValue>today</defaultValue>

                  <essential>false</essential>
```

```
            </property>
      </properties>

      <neighbors>
            <neighbor oid="1172171109453">
                  <code>interests</code>
                  <destinationConcept>
                        Interest
                  </destinationConcept>
                  <inverseNeighbor>member</inverseNeighbor>
                  <internal>true</internal>
                  <partOfManyToMany>true</partOfManyToMany>
                  <type>child</type>
                  <min>0</min>
                  <max>N</max>
            </neighbor>
      </neighbors>

</concept>

<concept oid="1172170900466">
      <code>Interest</code>
      <entitiesCode>Interests</entitiesCode>

      <properties>
            <property oid="1172171104844">
                  <code>categoryOid</code>
                  <propertyClass>
                        java.lang.Long
                  </propertyClass>
                  <required>true</required>
                  <reference>true</reference>
                  <referenceNeighbor>
                        category
                  </referenceNeighbor>

                  <essential>false</essential>
<referenceDropDownLookup>
                        true
                  </referenceDropDownLookup>
            </property>
            <property oid="1172171069177">
                  <code>description</code>
                  <propertyClass>
                        java.lang.String
                  </propertyClass>
                  <maxLength>510</maxLength>

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

```
            <neighbors>
                  <neighbor oid="1172171104844">
                        <code>category</code>
                        <destinationConcept>
                              Category
                        </destinationConcept>
                        <inverseNeighbor>interests</inverseNeighbor>
                        <internal>false</internal>
                        <partOfManyToMany>true</partOfManyToMany>
                        <type>parent</type>
                        <min>1</min>
                        <max>1</max>
                        <unique>true</unique>
                  </neighbor>
                  <neighbor oid="1172171109453">
                        <code>member</code>
                        <destinationConcept>
                              Member
                        </destinationConcept>
                        <inverseNeighbor>interests</inverseNeighbor>
                        <internal>true</internal>
                        <partOfManyToMany>true</partOfManyToMany>
                        <type>parent</type>
                        <min>1</min>
                        <max>1</max>
                        <unique>true</unique>
                  </neighbor>
            </neighbors>

      </concept>

      <concept oid="1171894920503">
            <code>Category</code>

                  <neighbor oid="1172171104844">
                        <code>interests</code>
                        <destinationConcept>
                              Interest
                        </destinationConcept>
                        <inverseNeighbor>category</inverseNeighbor>
                        <internal>false</internal>
                        <partOfManyToMany>true</partOfManyToMany>
                        <type>child</type>
                        <min>0</min>
                        <max>N</max>
                  </neighbor>

      </concept>
```

All Member properties are required. The code property is unique. The password property has true for the scramble view element. That means that the password value will be scrambled in the web application. The email property is of the java.lang.String class but its value will be validated by the org.modelibra.type.Email class from Modelibra. The role property has regular as the default value. The default value of the startDate property is today. The interests child neighbor is between the Member and Interest concepts. It is internal and a part of the many-to-many relationship. This fact is initially indicated in ModelibraModeler by invoking the pop-up menu on the relationship line.

Since the relationship between Interest and Category concepts is external, the Interest child concept has the categoryOid reference property pointing to the parent category. The description property is not required (false by default if the element is not present), but if there is a value, its maximal length is 510 characters.

Code Generation

As in the previous spiral, the code for the domain model and the web application is generated. The specific code from the previous spiral has been preserved by a selective code generation.

Since there are two new concepts, Member and Interest, and one new relationship linked to an existing concept, Category – Interest, the generic classes of the WebLink model and the Category concept must be regenerated and all classes for the Member and Interest concepts must be generated from scratch. In addition, since the Member concept is an entry into the model its empty XML data file must be generated. For convenience reasons, it would be helpful to update the specific-domain-config.xml file to include minimal declarations for the Member and Interest concepts as the basis for some specific changes, but without loosing already entered specific declarations for other concepts. This is accomplished by regenerating all specific minimal declarations and by including again the already entered specific declarations from the previous spiral. All this is related to the model changes. The code generation is done in the main method of the DmGenerator class in the first *** 2. section. There are two *** 2. sections, one for the model and one for the web application. When generating code, the choice must be made between the generation options that are presented as numbered sections. Another way to do it is to use the generateModelibraGenClasses method to regenerate all Gen classes and then generate the specific code related to the new concepts only.

The model changes effect also the code of the web application. The second *** 2. section of the main method generates page classes for the Member and Interest concepts. The application properties are regenerated to include text for the new concepts.

```
package dm.gen;

import org.modelibra.config.DomainConfig;

public class DmGenerator {

...

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

		// *** Modelibra ***

		// *** 1. Generate all ***
		// dmGenerator.getDmModelibraGenerator().generate();

		// *** 2. Generate new with preserving specific ***
		// dmGenerator.getDmModelibraGenerator()
		//		.generateModelibraGenClasses();
		// OR
		dmGenerator.getDmModelibraGenerator().generateModelGenClass("WebLink");
		dmGenerator.getDmModelibraGenerator().generateConceptGenClasses(
				"WebLink", "Category");
		dmGenerator.getDmModelibraGenerator().generateConceptClasses(
				"WebLink", "Member");
		dmGenerator.getDmModelibraGenerator().generateConceptClasses(
					"WebLink", "Interest");
		dmGenerator.getDmModelibraGenerator()
				.generateEntryConceptEmptyXmlDataFile("WebLink", "Member");
		// *** Optional: If done, be sure to have a backup of
		// specific-domain-config.xml ***
		dmGenerator.getDmModelibraGenerator().generateSpecificDomainConfig();

		// *** 3. Generate what you do not want by using comments ***
		// dmGenerator.getDmModelibraGenerator()
		//		.generateModelibraPartially();

		// *** ModelibraWicket ***

		// *** 1. Generate all ***
		// dmGenerator.getDmModelibraWicketGenerator().generate();

		// *** 2. Generate new with preserving specific ***
		// dmGenerator.getDmModelibraWicketGenerator()
		//		.generateConceptPageClasses("WebLink");
		// OR
		dmGenerator.getDmModelibraWicketGenerator()
				.generateConceptPageClasses("WebLink", "Member");
		dmGenerator.getDmModelibraWicketGenerator()
				.generateConceptPageClasses("WebLink", "Interest");
		dmGenerator.getDmModelibraWicketGenerator()
				.generateModelibraWicketAppProperties();

		// *** 3. Generate what you do not want by using comments ***
		// dmGenerator.getDmModelibraWicketGenerator()
		// .generateModelibraWicketPartially();
	}

}
```

The DmModelibraGenerator class has methods that allow a selective model code generation based on model changes.

```
package dm.gen;

import org.modelibra.config.ConceptConfig;
import org.modelibra.config.DomainConfig;
import org.modelibra.config.ModelConfig;
import org.modelibra.gen.DomainGenerator;
import org.modelibra.gen.DomainModelGenerator;
import org.modelibra.gen.ModelibraGenerator;

public class DmModelibraGenerator {

	private DomainConfig domainConfig;

	private ModelibraGenerator modelibraGenerator;

	private String authors;

	private String sourceDirectoryPath;

	private String testDirectoryPath;

	public DmModelibraGenerator(String domainCode, String domainType) {
		DmConfig dmConfig = new DmConfig(domainCode, domainType);
		domainConfig = dmConfig.getDomainConfig();
		modelibraGenerator = new ModelibraGenerator(domainConfig);

		authors = modelibraGenerator.getAuthors();
		sourceDirectoryPath = modelibraGenerator.getSourceDirectoryPath();
		testDirectoryPath = modelibraGenerator.getTestDirectoryPath();
	}

...

	public void generateModelGenClass(String modelCode) {
		ModelConfig modelConfig = domainConfig.getModelConfig(modelCode);
		String codeDirectoryPath = modelibraGenerator.getCodeDirectoryPath();
		DomainModelGenerator modelGenerator = new DomainModelGenerator(
				modelConfig, codeDirectoryPath);
		modelGenerator.generateGenModel();
	}

	public void generateConceptClasses(String modelCode, String conceptCode) {
		ModelConfig modelConfig = domainConfig.getModelConfig(modelCode);
		String codeDirectoryPath = modelibraGenerator.getCodeDirectoryPath();
		DomainModelGenerator modelGenerator = new DomainModelGenerator(
				modelConfig, codeDirectoryPath);
		ConceptConfig conceptConfig = modelConfig.getConceptConfig(conceptCode);
		modelGenerator.generateConcept(conceptConfig);
	}

	public void generateConceptGenClasses(String modelCode, String conceptCode) {
		ModelConfig modelConfig = domainConfig.getModelConfig(modelCode);
		String codeDirectoryPath = modelibraGenerator.getCodeDirectoryPath();
		DomainModelGenerator modelGenerator = new DomainModelGenerator(
				modelConfig, codeDirectoryPath);
		ConceptConfig conceptConfig = modelConfig.getConceptConfig(conceptCode);
		modelGenerator.generateGenConcept(conceptConfig);
	}

	public void generateEntryConceptEmptyXmlDataFile(String modelCode,
			String conceptCode) {
		ModelConfig modelConfig = domainConfig.getModelConfig(modelCode);
		String codeDirectoryPath = modelibraGenerator.getCodeDirectoryPath();
		DomainModelGenerator modelGenerator = new DomainModelGenerator(
				modelConfig, codeDirectoryPath);
		ConceptConfig conceptConfig = modelConfig.getConceptConfig(conceptCode);
		modelGenerator.generateEmptyXmlDataFile(conceptConfig);
	}

}
```

Similarly, the DmModelibraWicketGenerator class has methods that allow a selective web application code generation based on the same model changes.

```
package dm.gen;

import org.modelibra.config.ConceptConfig;
import org.modelibra.config.DomainConfig;
import org.modelibra.config.ModelConfig;
import org.modelibra.wicket.gen.DomainWicketGenerator;
import org.modelibra.wicket.gen.ModelibraWicketGenerator;

public class DmModelibraWicketGenerator {

	private ModelibraWicketGenerator modelibraWicketGenerator;

	private DomainConfig domainConfig;

	public DmModelibraWicketGenerator(DomainConfig domainConfig,
			String authors, String sourceDirectoryPath) {
		this.domainConfig = domainConfig;
		modelibraWicketGenerator = new ModelibraWicketGenerator(
				domainConfig, authors, sourceDirectoryPath);
	}

...

	public void generateModelibraWicketAppProperties() {
		modelibraWicketGenerator.getDomainWicketGenerator()
				.generateDomainAppProperties();
	}

	public void generateConceptPageClasses(String modelCode) {
		ModelConfig modelConfig = domainConfig.getModelConfig(modelCode);
		DomainWicketGenerator domainWicketGenerator = modelibraWicketGenerator
				.getDomainWicketGenerator();
		for (ConceptConfig conceptConfig : modelConfig.getConceptsConfig()) {
				domainWicketGenerator
						.generateDomainModelConceptUpdateTablePage(conceptConfig);
				domainWicketGenerator
						.generateDomainModelConceptDisplayTablePage(conceptConfig);
				domainWicketGenerator
						.generateDomainModelConceptDisplayListPage(conceptConfig);
				domainWicketGenerator
						.generateDomainModelConceptDisplaySlidePage(conceptConfig);
			}
		}

		public void generateConceptPageClasses(String modelCode, String conceptCode) {
			ModelConfig modelConfig = domainConfig.getModelConfig(modelCode);
			DomainWicketGenerator domainWicketGenerator = modelibraWicketGenerator
					.getDomainWicketGenerator();
			ConceptConfig conceptConfig = modelConfig.getConceptConfig(conceptCode);
			domainWicketGenerator.generateDomainModelConcept(conceptConfig);
		}

}
```

The selective code generation is a very important feature of Modelibra. It allows developers to accept model changes without loosing the productivity edge. Of course, it is always preferable not to have those changes in later phases of the project, but often reality is different and the selective code generation is a way to avoid the specification freeze, which does not serve users of the web application, and it is also a mechanism to avoid the development stress, which does not serve none of the stakeholders.

It is important to clean up the generated code by using the *Ctrl-Shift* and *O F S* keyboard keys in Eclipse to leave the code in a readable format.

After the code generation, some new specific code has been added to the specific classes. Finally, the new JUnit tests have been included in concepts' test classes.

Many-to-many Relationship

For the sake of space, only the code related to the many-to-many relationship will be presented. There is nothing new to explain in the generated GenMember abstract class.

```
package dmeduc.weblink.member;

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.interest.Interests;
```

```
public abstract class GenMember extends Entity<Member> {

...

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

	private Interests interests;

	public GenMember(IDomainModel model) {
		super(model);
		// internal child neighbors only
		setInterests(new Interests((Member) this));
	}

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

	public void setInterests(Interests interests) {
		this.interests = interests;
		if (interests != null) {
			interests.setMember((Member) this);
		}
	}

	public Interests getInterests() {
		return interests;
	}

}
```

The generated GenMembers abstract class has only one method related to the many-to-many relationship. The method is called getCategoryInterests and has one parameter of the Category entity class called category. The method is used to retrieve interests of the given category based the Member entry concept. Recall that the Category – Interest is an external relationship, while the Member – Interest is an internal relationship. The new interests object of the Interests entities class is created with the given category as its parent. For each member, her interests are obtained and the specific interest with the member and category parents is looked for. If found, the interest object is added to the interests object. After all members are checked, the interests object with all interests for the given category is returned as the result. If the category does not have any interests , the empty interests object is returned. This method is called in the getInterests method of the GenCategory class.

```
package dmeduc.weblink.member;

import java.util.Comparator;

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;

import dmeduc.weblink.category.Category;
import dmeduc.weblink.interest.Interest;
import dmeduc.weblink.interest.Interests;

public abstract class GenMembers extends Entities<Member> {

...

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

...

	/*
	 * ======= for each internal (part of) many-to-many child: get entities
	 * method based on the many-to-many external parent =======
	 */

	public Interests getCategoryInterests(Category category) {
		Interests interests = new Interests(category);
		for (Member member  : this) {
			Interest interest = member.getInterests().getInterest(member,
					category);
			if (interest != null) {
				interests.add(interest);
			}
		}
		return interests;
	}

...

}
```

The GenCategory class has the interests child neighbor. As external, the neighbor is not created in the constructor as the internal child neighbors. Thus, after the construction of the GenCategory object, the interests neighbor stays null. When the getInterests method is invoked for the first time, the interests object is null. First, the members entry is obtained from the model. Then, the getCategoryInterests method on the members object is called to obtain the category interests. Finally, the category interests are then set and returned. In this way, the external relationship is recreated from the objects already present in the model and, after that, it will behave as if it were the internal relationship. Hence, the next time, the interests object will not be null and it will be returned directly.

```
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.interest.Interests;
import dmeduc.weblink.member.Members;
import dmeduc.weblink.question.Questions;
import dmeduc.weblink.url.Urls;

public abstract class GenCategory extends Entity<Category> {

...

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

	private Interests interests;

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

	/* ====== external (part of) many-to-many child set and get methods ====== */

	public void setInterests(Interests interests) {
		this.interests = interests;
		if (interests != null) {
			interests.setCategory((Category) this);
		}
	}

	public Interests getInterests() {
		if (interests == null) {
			WebLink webLink = (WebLink) getModel();
			Members members = webLink.getMembers();
			setInterests(members.getCategoryInterests((Category) this));
		}
		return interests;
	}

}
```

There is nothing in the GenCategories class related to the many-to-many relationship.

The generated GenInterest abstract class has two parent neighbors. The member parent neighbor is internal, while the category parent neighbor is external. There is also the categoryOid reference property that is used to derive the category external parent. Recall that only the categoryOid number is saved in the member.xml data file. Both parent neighbors are parameters of the second constructor. They are set by the set parent methods. The second constructor first invokes the first constructor with the this keyword. The second constructor should be used in the application programming, whenever this is possible. During the loading process, done by Modelibra, the category is not immediately known, but the categoryOid reference property is available. The setCategoryOid method does not derive the corresponding category from the categories entry. This is done in the getCategory method by obtaining the categories entry object and by invoking the getReflexiveCategory method, with the categoryOid argument, on the categories entry object. The redundancy between the reference property and the external parent neighbor is maintained by their set and get methods.

```
package dmeduc.weblink.interest;

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;
import dmeduc.weblink.member.Member;

public abstract class GenInterest extends Entity<Interest> {

...

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

	private Long categoryOid;

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

	private Member member;

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

	private Category category;

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

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

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

	public GenInterest(Member member, Category category) {
		this(category.getModel());
		// parents
		setMember(member);
		setCategory(category);
	}
	/* ======= reference property set and get methods ======= */

	public void setCategoryOid(Long categoryOid) {
		this.categoryOid = categoryOid;
		category = null;
	}

	public Long getCategoryOid() {
		return categoryOid;
	}

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

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

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

}
```

The GenInterests class has also two parents, the member internal parent and the category external parent. The difference between the GenInterests and GenInterest classes is that in the GenInterest class both parents are not null after the model initialization, while in the GenInterests class only one of the parents is not null. When interests are of the category parent, the member parent is null. When interests are of the member parent, the category parent is null. In other words, the interests object is created either by the second constructor, which has the member parameter, or the third constructor, which has the category parameter. If the interests object is created with the first constructor, both parents are null. This is the reason that the first constructor should not be lightly used. It is there mostly for Java reflection [Reflection] used often in Modelibra.

Since each interest must (again, ignore the constructor without two parents) have both Member and Category parents, and since the id of the Interest concept is determined by those two parents, it is possible to obtain at most one interest by the getInterest method. If such an interest does not exist, null is returned.

```
package dmeduc.weblink.interest;

import java.util.Comparator;

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 dmeduc.weblink.category.Category;
import dmeduc.weblink.member.Member;

public abstract class GenInterests extends Entities<Interest> {

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

	private Member member;

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

	private Category category;

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

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

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

	public GenInterests(Member member) {
		this(member.getModel());
		// parent
		setMember(member);
	}

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

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

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

	public Category getCategory() {
		return category;
	}

	/*
	 * ======= for a many-to-many concept: get entity method based on all
	 * many-to-many parents =======
	 */

	public Interest getInterest(Member member, Category category) {
		for (Interest interest : this) {
			if (interest.getMember() == member
					&& interest.getCategory() == category) {
				return interest;
			}
		}
		return null;
	}

	/* ======= for a many-to-many concept: post add propagation ======= */

	protected boolean postAdd(Interest interest) {
		if (!isPost()) {
			return true;
		}
		boolean post = true;
		if (super.postAdd(interest)) {
			DomainModel model = (DomainModel) getModel();
			if (model.isInitialized()) {
				Member member = getMember();
				if (member == null) {
					Member interestMember = interest.getMember();
					if (!interestMember.getInterests().contain(interest)) {
						post = interestMember.getInterests().add(interest);
					}
				}
				Category category = getCategory();
				if (category == null) {
					Category interestCategory = interest.getCategory();
					if (!interestCategory.getInterests().contain(interest)) {
						post = interestCategory.getInterests().add(interest);
					}
```

```
                    }
                }
            } else {
                post = false;
            }
            return post;
        }

        /* ======= for a many-to-many concept: post remove propagation ======= */

        protected boolean postRemove(Interest interest) {
            if (!isPost()) {
                return true;
            }
            boolean post = true;
            if (super.postRemove(interest)) {
                Member member = getMember();
                if (member == null) {
                    Member interestMember = interest.getMember();
                    if (interestMember.getInterests().contain(interest)) {
                        post = interestMember.getInterests().remove(interest);
                    }
                }
                Category category = getCategory();
                if (category == null) {
                    Category interestCategory = interest.getCategory();
                    if (interestCategory.getInterests().contain(interest)) {
                        post = interestCategory.getInterests().remove(interest);
                    }
                }
            } else {
                post = false;
            }
            return post;
        }

        /* ======= for a many-to-many concept: post update propagation ======= */

        protected boolean postUpdate(Interest beforeInterest, Interest afterInterest){
            if (!isPost()) {
                return true;
            }
            boolean post = true;
            if (super.postUpdate(beforeInterest, afterInterest)) {
                Member beforeInterestMember = beforeInterest.getMember();
                Member afterInterestMember = afterInterest.getMember();

                if (beforeInterestMember != afterInterestMember) {
                    post = beforeInterestMember.getInterests().remove(
                                beforeInterest);
                    if (post) {
                        post = afterInterestMember.getInterests()
```

```
                                    .add(afterInterest);
                        if (!post) {
                              beforeInterestMember.getInterests()
                                    .add(beforeInterest);
                        }
                  }
            }
            Category beforeInterestCategory = beforeInterest.getCategory();
            Category afterInterestCategory = afterInterest.getCategory();

            if (beforeInterestCategory != afterInterestCategory) {
                  post = beforeInterestCategory.getInterests().remove(
                              beforeInterest);
                  if (post) {
                        post = afterInterestCategory.getInterests().add(
                                    afterInterest);
                        if (!post) {
                              beforeInterestCategory.getInterests().add(
                                          beforeInterest);
                        }
                  }
            }
      } else {
            post = false;
      }
      return post;
   }

...

}
```

When a new interest, with the member and category parameters in the constructor, is created and the added to the interests object of the member parent, the same interest object must be added to the interests objects of the corresponding category. This is done by the postAdd protected method. The add method of Modelibra calls the postAdd method. If the post action should be done (the default is true), the generated postAdd method calls first the postAdd method of Modelibra in which Modelibra validates the id constraint. If the new interest satisfies the id constraint, the post add propagation is done but only if the model has been initialized. This is done to avoid unnecessary validations in the model initialization (loading) process. If the interests objects belongs to a category, the member neighbor of the interests object is null, and the new interest is added to the not null member obtained from the interest parameter. If the interests objects belongs to a member, the category neighbor of the interests object is null, and the new interest is added to the not null category obtained from the interest parameter. In this way, the model stays consistent. Otherwise, a user would find that an interest that relates a certain member and a certain category, exists in the collection of interests of the member, but not in the collection of interests of the category. In this situation, a user would loose confidence in the model and in Modelibra.

The postRemove and postUpdate protected methods follow a similar reasoning.

The generic code that maintains the redundancy of a many-to-many relationship is rather complex.

Fortunately, this code is generated. However, the understanding of the generated code increases the confidence of a programmer for Modelibra.

Specific Code

The Members class has a few specific methods.

The getMemberByCode method calls the getMember generated method, which in turns calls the retrieveByProperty method from Modelibra.

```
public Member getMemberByCode(String code) {
    return getMember("code", code);
}
```

The following three specific methods use the PropertySelector class from Modelibra to define a selection of members based on a property.

```
public Members getReceiveEmailMembers() {
    PropertySelector propertySelector = new PropertySelector("receiveEmail");
    propertySelector.defineEqual(Boolean.TRUE);
    return getMembers(propertySelector);
}

public Members getRecentMembers(Date beforeRecentDate) {
    PropertySelector propertySelector = new PropertySelector("startDate");
    propertySelector.defineGreaterThan(beforeRecentDate);
    return getMembers(propertySelector);
}

public Members getAdminMembers() {
    PropertySelector propertySelector = new PropertySelector("role");
    propertySelector.defineEqual("admin");
    return getMembers(propertySelector);
}
```

The getRegularMembersThatReceiveEmail specific method calls the selectByMethod method from Modelibra to select all members that satisfy the isRegularReceiveEmail specific method defined in the Member class.

```
public Members getRegularMembersThatReceiveEmail() {
    return (Members) selectByMethod("isRegularReceiveEmail");
}

public boolean isRegularReceiveEmail() {
    boolean selected = false;
    if (getRole().equals("regular") && isReceiveEmail()) {
        selected = true;
    }
    return selected;
```

```
	}
```

The getMembersOrderedByLastFirstName specific method sorts members first by the last name then by the first name. The method uses the CompositeComparator class from Modelibra to define the composite order that is then passed to the getMembers method of the GenMembers class.

```
	public Members getMembersOrderedByLastFirstName(boolean ascending) {
		CompositeComparator compositePropertyComparator = new
	CompositeComparator(
				new PropertyComparator("lastName"), new PropertyComparator(
						"firstName"));
		return getMembers(compositePropertyComparator, ascending);
	}
```

There are two specific methods related to emails. The getEmails method returns a list of emails of members. The emailMessage method may be used to send the same message to members. The actual sending of an email is delegated to the EmailConfig utility class that can be found in the org.modelibra.util package. The emailMessage method has not been used in this spiral.

```
	public List<String> getEmails() {
		List<String> emails = new ArrayList<String>();
		for (Member member  : this) {
			emails.add(member.getEmail().toString());
		}
		return emails;
	}

	public void emailMessage(EmailConfig emailConfig, String subject,
			String message) {
		try {
			for (Member member : this) {
				emailConfig
					.send(member.getEmail().toString(), subject, message);
			}
		} catch (DmException e) {
			log.error("Error in Members.emailMessage: " + e.getMessage());
		}
	}
```

In order to support the sending of messages, there must be a valid email configuration in the email-config.xml file located in the config directory. If there is more than one email configuration, the last configuration is used.

```
<emails>
	<email oid="1">
		<code>vlgiiora</code>
		<toSendEmail>yes</toSendEmail>
		<from>dzenan.ridjanovic@videotron.ca</from>
		<outServer>relais.videotron.ca</outServer>
		<password>xxxxxxxx</password>
	</email>
	<email oid="2">
		<code>ridjanod</code>
		<toSendEmail>yes</toSendEmail>
		<from>dzenan.ridjanovic@fsa.ulaval.ca</from>
		<outServer>hermes.ulaval.ca</outServer>
		<password>xxxxxxxx</password>
	</email>
</emails>
```

The validation of member roles is done with the postAdd and postUpdate specific methods (of the specific Members class). The actual validation is done by the hasValidRole method of the Member class. The valid roles are currently admin and regular. If a member has a role that is not valid, the post method will fail and the action will not be completed. If there is a validation error, the Member.role.validation key is used to keep the error message in the current object of the Members class.

```
	protected boolean postAdd(Member member) {
		if (!isPost()) {
			return true;
		}
		boolean validMember = false;
		if (super.postAdd(member)) {
			validMember = member.hasValidRole();
			if (!validMember) {
				getErrors().add("Member.role.validation",
						"Role must be regular or admin.");
			}
		}
		return validMember;
	}

	protected boolean postUpdate(Member beforeMember, Member afterMember) {
		if (!isPost()) {
			return true;
		}
		boolean validMember = false;
		if (super.postUpdate(beforeMember, afterMember)) {
			validMember = afterMember.hasValidRole();
			if (!validMember) {
				getErrors().add("Member.role.validation",
						"Role must be regular or admin.");
			}
		}
		return validMember;
	}

	protected boolean hasValidRole() {
		boolean valid = false;
		if ((getRole().equals("regular"))
				|| (getRole().equals("admin"))) {
			valid = true;
		}
		return valid;
	}
```

In addition¸ the same error key is added to the application specific properties file. The international version of the web application will be handled in one of future spirals. Until then, the specific properties file will be in English. The application generic file, which is generated, is called DmEducApp.properties and it is placed in the dmeduc.wicket.app package. The application specific file in English is named DmEducApp_en.properties, and is located in the same package. The specific properties file is not generated. Its purpose is to customize text messages that appear in web pages. Its content is presented here as key=property pairs.

```
Url.creationDate=Creation Date
Url.creationDate.required=Creation Date is required.
Url.creationDate.length=Creation Date is longer than 16.
Url.updateDate=Update Date
Url.updateDate.length=Update Date is longer than 16.

Comment.creationDate=Creation Date
Comment.creationDate.required=Creation Date is required.
Comment.creationDate.length=Creation Date is longer than 16.

Category.name=Category
Category.name.required=Category is required.
Category.name.length=Category is longer than 64.

Question.categoryOid=Category
Question.creationDate=Creation Date
Question.creationDate.required=Creation Date is required.
Question.creationDate.length=Creation Date is longer than 16.

Member.lastName=Last Name
Member.lastName.required=Last Name is required.
Member.lastName.length=Last Name is longer than 32.
Member.firstName=First Name
Member.firstName.required=First Name is required.
Member.firstName.length=First Name is longer than 32.
Member.receiveEmail=ReceiveEmail
Member.receiveEmail.required=ReceiveEmail is required.
Member.role.validation=Role is not valid: must be regular or admin.
Member.startDate.required=Start Date is required.
Member.startDate.length=Start Date is longer than 16.

Interest.categoryOid=Category
```

There is no specific code in the Interest and Interests classes.

JUnit Tests

The MembersTest class has seven tests. The emptyMember test tries to add a member with empty but not null properties. This test can be executed in Eclipse, without other tests from this class, by selecting the method in the Package Explorer section of Eclipse and using the pop-up menu to *Run As/JUnit Test*. The test assertion that there will be an error is true (or more precisely, the test assertion that the error collection will be empty is false). The error appears in the log.

```
      Member.email.validation = Property Member.email (Email) is not valid based on          the
org.modelibra.type.Email validation type.
```

Note that required String properties accept the "" empty text (not null) as a valid value. The emptyMemberWithEmail test shows that when the valid email is provided there is no error. The test assertion that the error collection will be empty is true. The test uses a valid email but other required properties are still empty. This time there is no error and the new member is created with default values for the receiveEmail, role and startDate properties. This shows a need for an additional validation for String properties either in Modelibra or at least in the specific postAdd and postUpdate methods.

```
<member oid="1194560481216">
    <code></code>
    <password></password>
    <lastName></lastName>
    <firstName></firstName>
    <email>empty.member@empty.com</email>
    <receiveEmail>false</receiveEmail>
    <role>regular</role>
    <startDate>2007-11-08</startDate>
</member>
```

However, when the empty space is replaced by null in the nullMemberWithEmail test, there is an error.

The allMembersReceiveEmail test updates all members to receive email in future. The profRole test has the false assertion since the “prof” role does not exist. The specific postUpdate method in the Members class does not accept the “prof” role. The validation error message, which appears in the log, is

```
      Member.role.validation = Role must be regular or admin.
```

In contrast, the adminRole test has the true assertion. Thus, the test succeeds in updating the “role” property to the valid “admin” value.

The nonEmptyMember test finds a member with the empty code and updates the member to some valid values.

```
package dmeduc.weblink.member;

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;

public class MembersTest {

	private static Members members;

	@BeforeClass
	public static void beforeMembers() throws Exception {
		members = DmEducTest.getSingleton().getDmEduc().getWebLink()
				.getMembers();
	}

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

	@Test
	public void emptyMember() throws Exception {
		OutTester.outputText("=== Test: Empty Member ===");
		String code = "";
		String password = "";
		String lastName = "";
		String firstName = "";
		String email = "";
		members.createMember(code, password, lastName, firstName, email);
		assertFalse(members.getErrors().isEmpty());
	}

	@Test
	public void emptyMemberWithEmail() throws Exception {
		OutTester.outputText("=== Test: Empty Member With Email ===");
		String code = "";
		String password = "";
		String lastName = "";
		String firstName = "";
		String email = "empty.member@empty.com";
		members.createMember(code, password, lastName, firstName, email);
		assertTrue(members.getErrors().isEmpty());
	}

	@Test
	public void nullMemberWithEmail() throws Exception {
		OutTester.outputText("=== Test: Null Member With Email ===");
		String code = null;
		String password = null;
		String lastName = null;
		String firstName = null;
		String email = "empty.member@empty.com";
		members.createMember(code, password, lastName, firstName, email);
		assertFalse(members.getErrors().isEmpty());
	}

	@Test
	public void allMembersReceiveEmail() throws Exception {
		OutTester.outputText("=== Test: All Members Receive Email ===");
		for (Member member : members) {
			Member memberCopy = member.copy();
			memberCopy.setReceiveEmail(true);
			members.update(member, memberCopy);
		}
		assertTrue(members.getErrors().isEmpty());
	}

	@Test
	public void profRole() throws Exception {
		OutTester.outputText("=== Test: Prof Role  ===");
		Member dr = members.getMemberByCode("dr");
		if (dr != null) {
			Member memberCopy = dr.copy();
			memberCopy.setRole("prof");
			members.update(dr, memberCopy);
			assertFalse(members.getErrors().isEmpty());
		}
	}

	@Test
	public void adminRole() throws Exception {
		OutTester.outputText("=== Test: Admin Role  ===");
		Member dr = members.getMemberByCode("dr");
		if (dr != null) {
			Member memberCopy = dr.copy();
			memberCopy.setRole("admin");
			members.update(dr, memberCopy);
			assertTrue(members.getErrors().isEmpty());
		}
	}

	@Test
	public void nonEmptyMember() throws Exception {
		OutTester.outputText("=== Test: Non Empty Member ===");
		Member member = members.getMemberByCode("");
		if (member != null) {
			Member memberCopy = member.copy();
			memberCopy.setCode("tj");
			memberCopy.setPassword("tj");
			memberCopy.setLastName("Jacobs");
			memberCopy.setFirstName("Tom");
			memberCopy.setEmail("tom.jacobs@gmail.com");
			members.update(member, memberCopy);
			assertTrue(members.getErrors().isEmpty());
		}
	}

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

}
```

The InterestsTest class has two tests. The webMeetingInterest test is done for the member with the “dr” code. This member is found in the beforeInterests method that has the @BeforeClass annotation. The beforeInterests method is called only once before all tests of this class. It finds members and categories entry points into the model. The interests non-entry is found for the “dr” member. The webMeetingInterest test adds a new interest for the member with the “dr” code. The interest is for the “Web Meeting” category. The “Web Meeting” category is a sub-category of the “Education” root category. The “Web Meeting” is obtained by the getCategoryByName method of the Categories class. The getCategoryByName method delegates the task of retrieving a sub-category to the getReflexiveCategory method of the Categories class.

The educationInterests test is not done for the member with the “dr” code. It displays, in the log, interests of the “Education” category. Since the interests property of the test class is not related any more to the member with the “dr” code, the beforeTest method finds the member's interests again, thus providing a valid pre-condition for an eventual new test based on the chosen member's interests.

```
package dmeduc.weblink.interest;

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.category.Categories;
import dmeduc.weblink.category.Category;
import dmeduc.weblink.member.Member;
import dmeduc.weblink.member.Members;

public class InterestsTest {

	private static Members members;

	private static Member member;

	private static Categories categories;

	private static Interests interests;
```

```
@BeforeClass
public static void beforeInterests() throws Exception {
      members = DmEducTest.getSingleton().getDmEduc().getWebLink()
                  .getMembers();
      member = members.getMemberByCode("dr");
      interests = member.getInterests();
      categories = DmEducTest.getSingleton().getDmEduc().getWebLink()
                  .getCategories();
}

@Before
public void beforeTest() throws Exception {
      if (interests.getMember() == null) {
            interests = member.getInterests();
      }
      interests.getErrors().empty();
}

@Test
public void webMeetingInterest() throws Exception {
      OutTester.outputText("=== Test: Web Meeting Interest ===");
      Category category = categories.getCategoryByName("Web Meeting");
      if (category != null) {
            Interest interest = interests.getInterest(member, category);
            if (interest == null) {
                  interest = new Interest(member, category);
                  interest.setDescription("I teach over Internet.");
                  interests.add(interest);
            } else {
                  OutTester.outputText("--- " + interest.getOid()
                              + " interest already exists. ---");
            }
            assertTrue(interests.getErrors().isEmpty());
      }
}

@Test
public void educationInterests() throws Exception {
      OutTester.outputText("=== Test: Education Interests ===");
      Category category = categories.getCategoryByName("Education");
      if (category != null) {
            interests = category.getInterests();
            interests.output(category.getName() + " (" + category.getOid()
                        + ") Interests");
            assertTrue(interests.getErrors().isEmpty());
      }
}

@After
public void afterTest() throws Exception {
      interests.getErrors().output("Interests");
}
```

```
}
```

The CategoriesTest has a new test called modelibraFramework. This test creates the “Modelibra” category as a sub-category of the “Framework” category, but only if the “Modelibra” category does not already exist.

```
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;

      @BeforeClass
      public static void beforeCategories() throws Exception {
            categories = DmEducTest.getSingleton().getDmEduc().getWebLink()
                        .getCategories();
      }

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

      ...

      @Test
      public void modelibraFramework() throws Exception {
            OutTester.outputText("=== Test: Modelibra Framework ===");
            Category category = categories.getCategoryByName("Framework");
            if (category != null) {
                  Categories subcategories = category.getCategories();
                  categories = subcategories;
                  Category modelibraCategory = subcategories
                              .getCategoryByName("Modelibra");
                  if (modelibraCategory == null) {
                        Category subcategory = new Category(category);
                        subcategory.setName("Modelibra");
                        subcategory.setApproved(true);
```

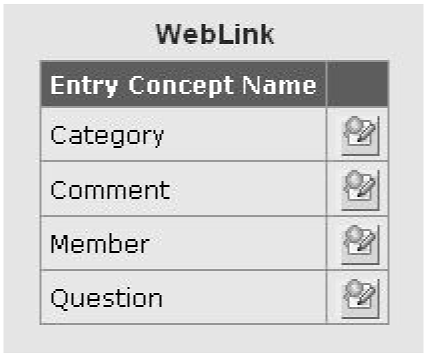
```
                    subcategories.add(subcategory);
                    assertTrue(subcategories.getErrors().isEmpty());
               } else {
                    OutTester
                    .outputText("Modelibra framework category already exists.");
               }
          } else {
               OutTester.outputText("Framework category does not exist.");
          }
     }

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

}
```

Web Application

The web application has four model entry concepts: Category, Comment, Member and Question (Fig. 8.2).

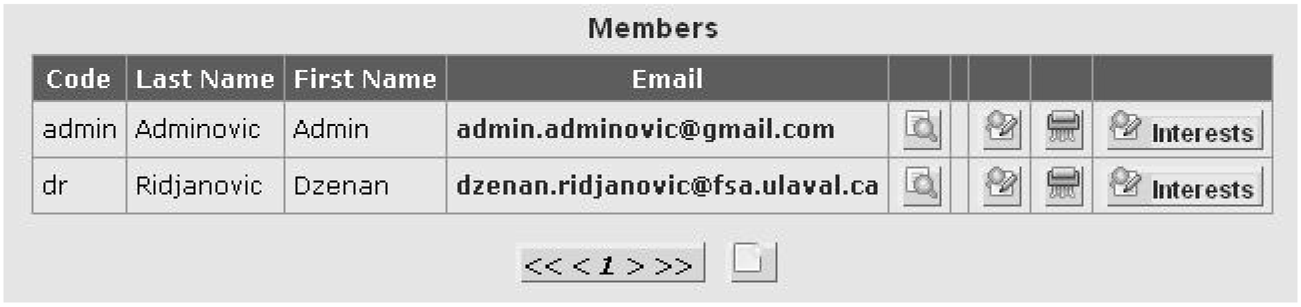


WebLink

Entry Concept Name	
Category	
Comment	
Member	
Question	

Figure 8.2. Model entry concepts

There are two members, one with the “admin” role and the other with the “regular” role (Fig. 8.3). Since the role property is configured as not essential, it is not shown in the table of members. However, by the member codes or names, it is easy to guess who is an administrator.



Members

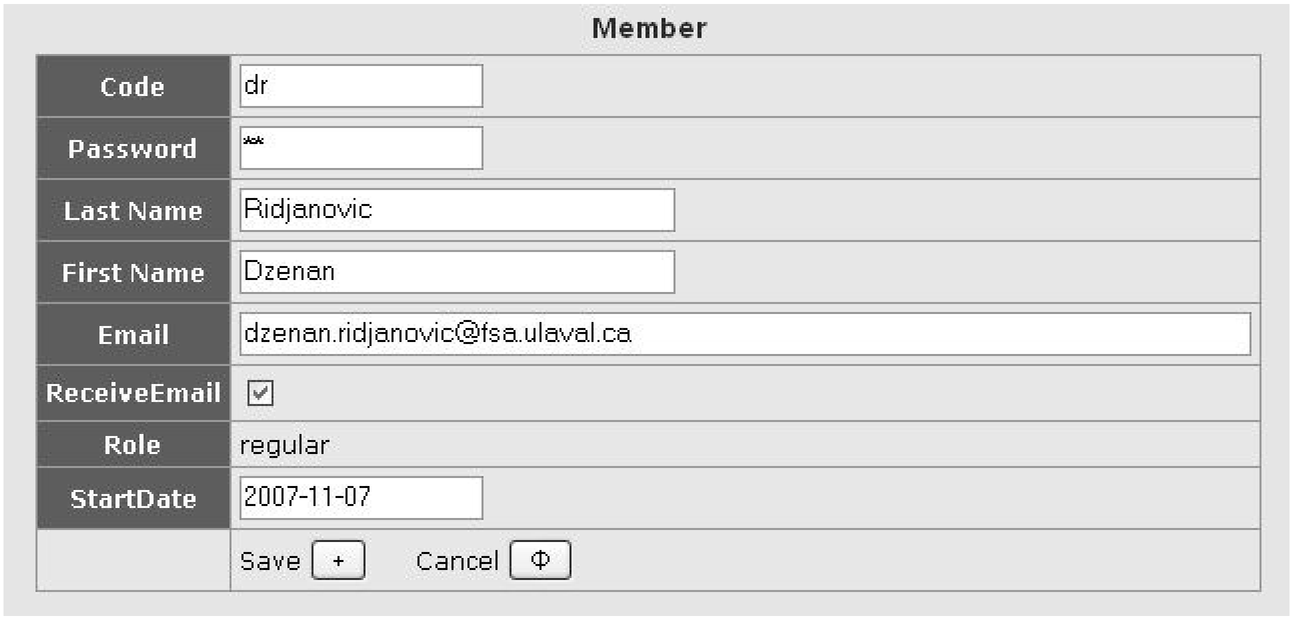
Code	Last Name	First Name	Email					
admin	Adminovic	Admin	admin.adminovic@gmail.com					Interests
dr	Ridjanovic	Dzenan	dzenan.ridjanovic@fsa.ulaval.ca					Interests

Figure 8.3. Members

In a member form in Figure 8.4., the role property cannot be updated, for obvious reasons, from the default “regular” value to the “admin” value. This is accomplished by changing the XML declaration of the role property of the Member concept in the specific-domain-config.xml file. The default true value of the update element, in the reusable-domain-config.xml file, is changed to false in the specific-domain-config.xml file.

```
<property oid="1172170850713">
      <code>role</code>
      <extension>true</extension>
      <extensionProperty>role</extensionProperty>

      <update>false</update>
</property>
```

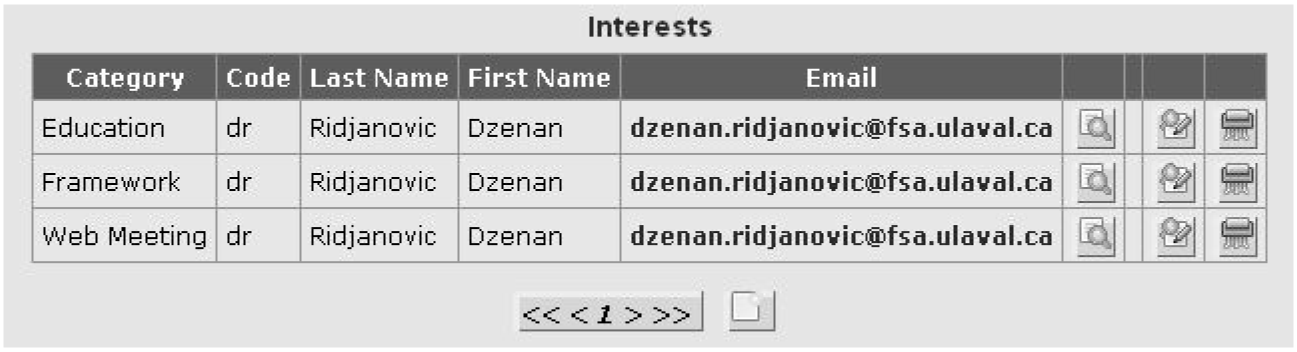


Member

Code	dr
Password	**
Last Name	Ridjanovic
First Name	Dzenan
Email	dzenan.ridjanovic@fsa.ulaval.ca
ReceiveEmail	✓
Role	regular
StartDate	2007-11-07
	Save + Cancel Φ

Figure 8.4. Member

The interests of the member with the “dr” code are shown in Fig. 8.5.



Interests

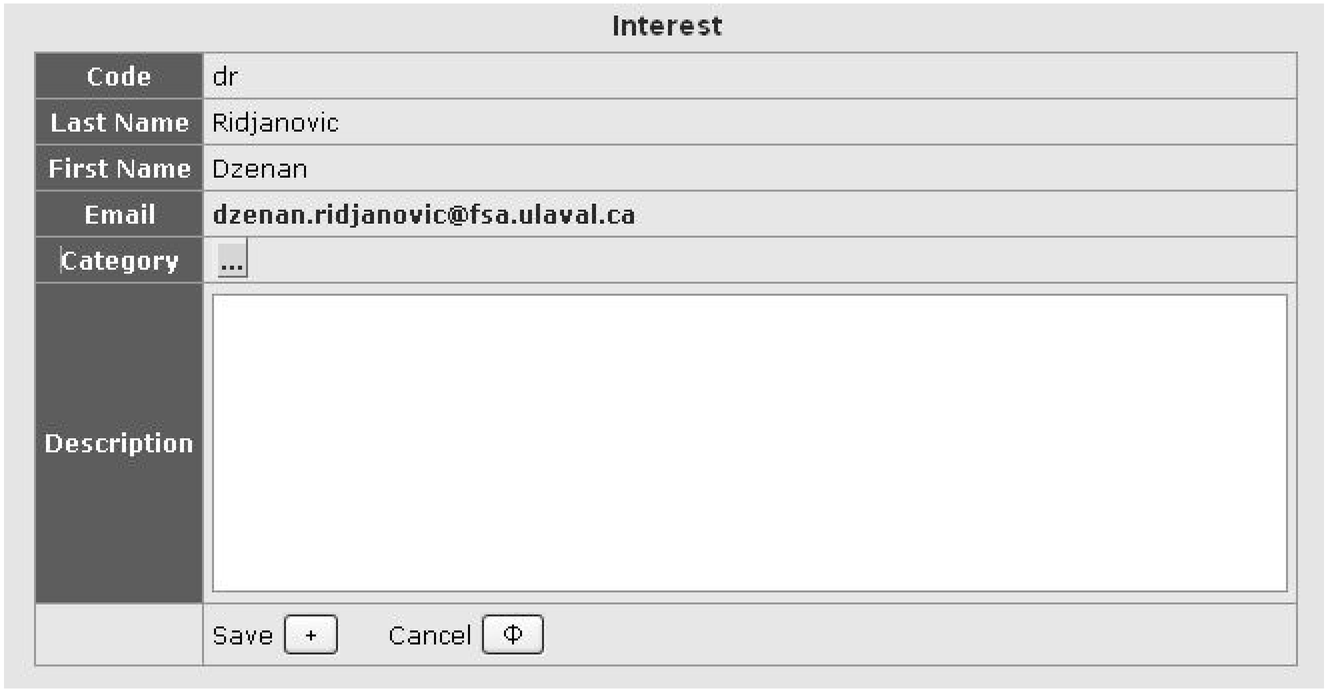
Category	Code	Last Name	First Name	Email			
Education	dr	Ridjanovic	Dzenan	dzenan.ridjanovic@fsa.ulaval.ca			
Framework	dr	Ridjanovic	Dzenan	dzenan.ridjanovic@fsa.ulaval.ca			
Web Meeting	dr	Ridjanovic	Dzenan	dzenan.ridjanovic@fsa.ulaval.ca			

<< < 1 > >>

Figure 8.5. My Interests

Note that the member properties that are declared as essential in the XML configuration appear in the table of members in Figure 8.3, and also in the table of member interests, as "absorbed" properties of the member parent. By changing the essential element from true to false, some of those properties may disappear from the table display.

By clicking on the *Add* link in the table of member interests, the form in Figure 8.6. appears. The member essential properties are displayed without possibility to change them. The Category parent is not assigned but there is a link with three points to look-up a category for this new interest.


Interest

Code	dr
Last Name	Ridjanovic
First Name	Dzenan
Email	dzenan.ridjanovic@fsa.ulaval.ca
Category	...
Description	
	Save + Cancel Φ

Figure 8.6. My New Interest

In the specific-domain-config.xml file, the Interest concept has a reference property called categoryOid. The referenceDropDownLookup element in the reusable-domain-config.xml is true by default. In the specific configuration, it is set to false to allow a look-up of a category in the table display of categories. In this way, by navigating through pages of sub-categories, a sub-category may be chosen to be a parent category of the interest.

```
<property oid="1172171104844">
      <code>categoryOid</code>
      <extension>true</extension>
      <extensionProperty>categoryOid</extensionProperty>

      <referenceDropDownLookup>false</referenceDropDownLookup>
</property>
```

Modelibra Interfaces

There is one new method used from the IEntities interfaces: selectByMethod.

```
      public IEntities<T> selectByMethod(String entitySelectMethodName,
        List<?> parameterList);
```

The method is used to select a subset of entities based on the user provided (specific) method. If the specific method does not have parameters, null can be used for the second argument. There is a convenience method in the Entities class for user provided methods without parameters.

```
    public IEntities<T> selectByMethod(String entitySelectMethodName) {
        return selectByMethod(entitySelectMethodName, null);
    }
```

Since the specific entities class, such as Members, inherits its behavior from the Entities abstract class, the convenience selectByMethod method with only one argument may be freely used.

Summary

A many-to-many relationship is defined on two concepts. In object oriented technologies, a many-to-many relationship is represented directly between the involved concepts. In relational databases, a many-to-many relationship is implemented through a third table representing participating tables. In Modelibra, a many-to-many relationship is represented with three concepts and two one-to-many relationships. Since one of the objectives of Modelibra is to provide an easy transfer of data from XML files to relational or object databases, representing a many-to-many relationship with three concepts and two one-to-many relationships is a rule. One of the relationships must be internal and the other external.

With at least one many-to-many relationship a domain model becomes a network of concepts. In order to save a network model in hierarchically organized XML data files, a network must be decomposed in multiple hierarchies, each starting with an entry concept. This is accomplished by entry concepts as roots of hierarchies and internal one-to-many relationships. An external relationship is used to reference a concept in another hierarchy. This is accomplished by the reference property in the child concept of the external relationship. The presence of both the parent neighbor and the reference property in the child concept of the external relationship produces redundancy that must be maintained. The generic code that maintains the redundancy of a many-to-many relationship is rather complex. Fortunately, the generic code is generated.

The next section of the book is about web components. The next chapter will explain how web components are created in ModelibraWicket. Of course, ModelibraWicket uses Wicket for its web components. The difference between ModelibraWicket and Wicket components is that ModelibraWicket has an intimate knowledge of Modelibra, while Wicket does not know anything about Modelibra. This knowledge of Modelibra enables ModelibraWicket to do more than Wicket can.

Questions

1. Is Modelibra an object oriented technology if it uses three concepts to represent a many-to-many relationship?

2. Explain why Modelibra uses three concepts to represent a many-to-many relationship?

3. Should an application programmer use a reference property in Modelibra?

4. Explain why there is a redundancy in a many-to-many relationship?

5. How would you represent a many-to-many relationship that has three participating concepts in addition to the many-to-many concept?

Exercises

Exercise 8.1.

Make a JUnit test to add a new interest in the collection of interests of a specific category. Display in the log both collections of interests (for the category and for the member of the new interest).

Exercise 8.2.

Introduce a new validation in the model, but which is not present in the configuration of the model. Make a JUnit test to verify the validation.

Exercise 8.3.

Add some specific code in both Interest and Interests classes. Make a JUnit test to use the specific code.

Exercise 8.4.

Override the post actions in the GenInterests class by creating the corresponding post actions with an empty body in the specific sub-class. Use the web application and explain the consequences.

Web Links

[Reflection] Java Reflection
http://java.sun.com/docs/books/tutorial/reflect/index.html