Chapter 5: Selection and Order of Entities

The objective of this chapter is to use selection and order of entities. Modelibra has a simple, object oriented query language. A subset of entities is selected from the source entities by defining a selector of an entity and by selecting only those entities that satisfy the selector. The result is an object of destination entities. Thus, further selections on the result may be applied again. If none of the source entities is selected, the object of destination entities is empty. The destination object is different from the source object, but the configuration is the same. A destination object is a model view of the source object. By default, add actions and remove actions on the destination entities propagate to the source entities.

The simplest selection is based on a single property and the equal relational operator. A property selector is defined for one property and one of many relational operators such as contain, greater than, match, etc. A selector may be composite consisting of several property selectors related by and, or, not logical operators.

Modelibra also offers a selection mechanism where a user provided Java selection method, which returns a boolean type value, is applied to the source entities. Every source entity that passes the method's selection expression is included in the destination entities. The selection method may have a list of parameters. The selection expression may be quite complex to satisfy various application needs.

The entities are sorted by applying a comparator. For example, urls may be ordered based on the category name with the ascending values. The simplest order is based on a single property and the ascending or descending sequence. A property comparator is defined for one property with an additional optional comparator for special cases such as a case insensitive text comparison. A comparator may be composite consisting of primary and secondary comparators, where each of them may be either a property comparator or a composite comparator.

Domain Model

The Url concept has a different identifier. A url must have the unique name while the same link value may be repeated. In this way the same web link may be used under different names. For a user, it means that the same web link may be categorized differently. For example, the web link for Modelibra may be used in both Framework and Education categories.

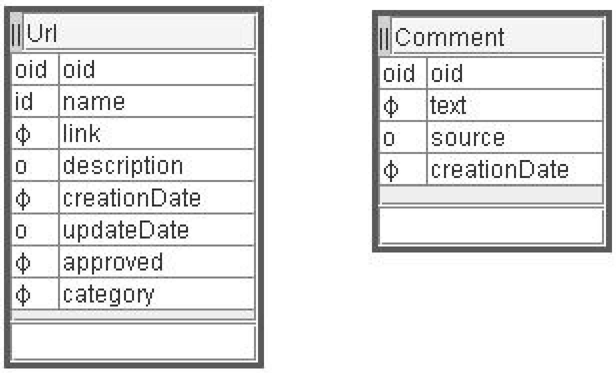

Figure 5.1. Url concept with the new name id

Domain Configuration

The specific-domain-config.xml file has been modified to include the new name property and to move the unique element with the true value from the link property to the name property.

```
<property oid="110110110100">
      <code>name</code>
      <propertyClass>java.lang.String</propertyClass>
      <maxLength>64</maxLength>
      <required>true</required>
      <unique>true</unique>
</property>
<property oid="110110110110">
      <code>link</code>
      <propertyClass>java.lang.String</propertyClass>
      <validateType>true</validateType>
      <validationType>java.net.URL</validationType>
      <maxLength>96</maxLength>
      <required>true</required>
</property>
```

Selection of Entities

There are two different ways of selecting entities. A single entity may be retrieved by one of methods in the IEntities interface whose names start with the retrieveBy prefix. A subset of entities may be selected by one of methods in the IEntities interface whose names start with the selectBy prefix. A retrieve method retrieves the first entity that satisfies a retrieval expression. If no entity is retrieved, null is returned. A select method selects a subset of entities that satisfies a selection expression. If none of entities is selected, the empty entities object is returned.

At most one url is retrieved by the getUrl method from the Urls class. The method accepts a property code and a retrieval object for that property.

```
	public Url getUrl(String propertyCode, Object property) {
		return retrieveByProperty(propertyCode, property);
	}
```

The getUrlByName method is a convenience method for the name property. It uses the getUrl method, which in turn uses the retrieveByProperty method. The retrieveByProperty method is the public method in the IEntities interface of Modelibra.

```
	public Url getUrlByName(String name) {
		return getUrl("name", name);
	}
```

The simplest selection of entities is based on a single property and the equal relational operator. The getComments method. from the Comments class, is a convenience method for selecting comments based on a property code and the property value. Only those comments that have the same property value will be selected. The selectByProperty method is the public method in the IEntities interface. Since the method returns an object of the IEntities type, the casting action transforms the result into the Comments object.

```
	public Comments getComments(String propertyCode, Object property) {
		return (Comments) selectByProperty(propertyCode, property);
	}
```

An example of the specific method that uses a specific property is given by the getSourceComments method, which returns comments whose source property is equal to the method's parameter.

```
	public Comments getSourceComments(String source) {
		return getComments("source", source);
	}
```

Similarly, the getUrls method is a convenience method for selecting urls based on the given property code and the property value. The selected urls represent the destination entities, while the current object represents the source entities.

```
	public Urls getUrls(String propertyCode, Object property) {
		return (Urls) selectByProperty(propertyCode, property);
	}
```

If none of the source entities is selected, the destination object is empty, but not null. It is important to understand that the destination object is different from the source object, but the configuration is the same (they belong to the same specific Entities class). A destination object is a model view of the source object. By default, adds and removals on the destination object propagate to the source object. This can be changed in Modelibra by passing the false value to the setPropagateToSource method inherited from the Entities class.

There are many other possible selections of entities based on a single property that use a relational operator other than equal. For those selections of entities a property selector is used. However, a property selector may be used for the equal relational operator as well. For example, a subset of urls that belong to the same category is obtained by the getCategoryUrls method that uses the PropertySelector class.

```
	public Urls getCategoryUrls(String category) {
		PropertySelector propertySelector = new PropertySelector("category");
		propertySelector.defineEqual(category);
		return getUrls(propertySelector);
	}
```

The getCategoryUrls method, which uses the PropertySelector class, is equivalent to the following convenience method.

```
	public Urls getCategoryUrls(String category) {
		return getUrls("category", category);
	}
```

The PropertySelector class is located in the org.modelibra package. The property name (or code in Modelibra's terms) is passed to the constructor of the PropertySelector class. The property value is provided to the defineEqual method of the PropertySelector class. Finally, the getUrls method with the property selector as its parameter is called.

The getUrls method with the ISelector parameter can be called for any selector type, not only for PropertySelector. The selectBySelector method is the public method in the IEntities interface. Since the method returns an object of the IEntities type, the casting action transforms the result into the Urls object.

```
	public Urls getUrls(ISelector selector) {
		return (Urls) selectBySelector(selector);
	}
```

A property selector is defined for one property and one of many relational operators such as equal, contain, greater than, match, etc. All types of selection operators are presented in the ISelector interface as public constants.

```
package org.modelibra;

public interface ISelector {

	public static final String NOT = "NOT";

	public static final String AND = "AND";

	public static final String OR = "OR";

	public static final String ALL = "ALL";

	public static final String NONE = "NONE";

	public static final String NULL = "NULL";

	public static final String EQUAL = "EQUAL";
```

```
    public static final String CONTAIN = "CONTAIN";

    public static final String CONTAIN_SOME = "CONTAIN_SOME";

    public static final String CONTAIN_ALL = "CONTAIN_ALL";

    public static final String BEGIN = "BEGIN";

    public static final String END = "END";

    public static final String MATCH = "MATCH";

    public static final String IN = "IN";

    public static final String LESS_THAN = "LESS_THAN";

    public static final String LESS_EQUAL = "LESS_EQUAL";

    public static final String GREATER_THAN = "GREATER_THAN";

    public static final String GREATER_EQUAL = "GREATER_EQUAL";

    public static final String BETWEEN = "BETWEEN";

}
```

The first three constants represent the logical operators used in a composite selector. The rest of the constants represent the relational operators used by the PropertySelector class. For each relational constant there is a corresponding public method in the PropertySelector class. For example, for the *EQUAL* constant there is the defineEqual method, and for the *GREATER_THAN* constant there is the defineGreaterThan method.

A property selector for a String property may ignore the property value case by using the setCaseSensitive method in the PropertySelector class.

```
    public void setCaseSensitive(boolean caseSensitive) {
        this.caseSensitive = caseSensitive;
    }
```

A selector may be composite consisting of several property selectors related by and, or, not logical operators. Approved urls are selected by the getApprovedUrls method, which uses the PropertySelector class,

```
    public Urls getApprovedUrls() {
        PropertySelector propertySelector = new PropertySelector("approved");
        propertySelector.defineEqual(Boolean.TRUE);
        return getUrls(propertySelector);
    }
```

while not approved urls are selected by the getNotApprovedUrls method, which uses the CompositeSelector class.

```
	public Urls getNotApprovedUrls() {
		PropertySelector propertySelector = new PropertySelector("approved");
		propertySelector.defineEqual(Boolean.TRUE);
		CompositeSelector compositeSelector = new CompositeSelector(
				ISelector.NOT, propertySelector);
		return getUrls(compositeSelector);
	}
```

However, the same result may be obtained by using the equal relational operator and by providing the FALSE constant.

```
	public Urls getNotApprovedUrls() {
		PropertySelector propertySelector = new PropertySelector("approved");
		propertySelector.defineEqual(Boolean.FALSE);
		return getUrls(propertySelector);
	}
```

The CompositeSelector class is located in the org.modelibra package. It has two constructors. The constructor with three parameters is used to compose two selectors with the AND or OR logical operator. The constructor with two parameters is used to add the NOT logical operator to a selector. By composing composite selectors, a quite complex (composite) selector may be constructed.

```
package org.modelibra;

public class CompositeSelector implements ISelector {

	private ISelector leftSelector;

	private String logicalOperator;

	private ISelector rightSelector;

	public CompositeSelector(ISelector leftSelector, String logicalOperator,
			ISelector rightSelector) {
		this.leftSelector = leftSelector;
		this.logicalOperator = logicalOperator;
		this.rightSelector = rightSelector;
	}

	public CompositeSelector(String logicalOperator, ISelector leftSelector) {
		this.logicalOperator = logicalOperator;
		this.leftSelector = leftSelector;
	}

	public ISelector getLeftSelector() {
		return leftSelector;
	}

	public String getLogicalOperator() {
		return logicalOperator;
	}

	public ISelector getRightSelector() {
		return rightSelector;
	}

}
```

Recently created urls are obtained by providing a date to the getRecentlyCreatedUrls method. Only urls that have the creationDate property value greater than the given date are selected.

```
	public Urls getRecentlyCreatedUrls(Date beforeRecentDate) {
		PropertySelector propertySelector = new PropertySelector("creationDate");
		propertySelector.defineGreaterThan(beforeRecentDate);
		return getUrls(propertySelector);
	}
```

The last month urls are selected by the getLastMonthUrls method.

```
	public Urls getLastMonthUrls() {
		Date date = new Date();
		EasyDate easyDate = new EasyDate(date);
		return getRecentlyCreatedUrls(easyDate.getPreviousMonthDate());
	}
```

The current date representing today is created by the default constructor of the Date class. Then, the date is transformed into an object of the EasyDate class that has the getPreviousMonthDate method.

The today's urls are obtained by the getTodayUrls method.

```
	public Urls getTodayUrls() {
		Date today = new Date();
		EasyDate easyToday = new EasyDate(today);
		Date yesterday = easyToday.getPreviousDayDate();
		return getRecentlyCreatedUrls(yesterday);
	}
```

The getPreviousDayDate method of the EasyDate class is used to find the yesterday's date that is used as an argument of the getRecentlyCreatedUrls method. The whole process seems complicated, but try to obtain the same result without using the EasyDate class. Try also to modify the getRecentlyCreatedUrls method using the greater or equal relational operator.

The Url class has the description property. A single keyword may be used to retrieve urls that contain the keyword in their descriptions.

```
	public Urls getKeywordUrls(String keyword) {
		PropertySelector propertySelector = new PropertySelector("description");
		propertySelector.defineContain(keyword);
		return getUrls(propertySelector);
	}
```

More complex selections based on several keywords may be performed by the getSomeKeywordUrls and getAllKeywordUrls methods. Keywords are passed to the methods as a String array. Urls that contain at least one (some) of the given keywords are selected by the getSomeKeywordUrls method.

```
public Urls getSomeKeywordUrls(String[] keywords) {
      PropertySelector propertySelector = new PropertySelector("description");
      propertySelector.defineContainSome(keywords);
      return getUrls(propertySelector);
}
```

Urls that contain all of the given keywords are selected by the getAllKeywordUrls method.

```
public Urls getAllKeywordUrls(String[] keywords) {
      PropertySelector propertySelector = new PropertySelector("description");
      propertySelector.defineContainAll(keywords);
      return getUrls(propertySelector);
}
```

Modelibra also offers a selection mechanism where a user provided Java selection method in a specific Entity class, which returns a boolean type value, is applied to the source entities of the corresponding specific Entities class. Every source entity that passes the method's selection expression is included in the destination entities. The entity selection method may have a list of parameters. This list may be null, empty or have one or more objects (but not values of primitive types). The entity selection method must return the boolean value but it may have a selection expression of any complexity.

For example, the Url class has two new methods to check if a url is created today and if a url is created and approved today.

```
public boolean isCreatedToday() {
      EasyDate easyToday = new EasyDate(new Date());
      if (easyToday.equals(getCreationDate())) {
            return true;
      }
      return false;
}

public boolean isCreatedAndApprovedToday() {
      if (isApproved() && isCreatedToday()) {
            return true;
      }
      return false;
}
```

The isCreatedAndApprovedToday method of the Url class is referenced in the getUrlsCreatedAndApprovedToday method of the Urls class to select entities by this specific method.

```
public Urls getUrlsCreatedAndApprovedToday() {
      return (Urls) selectByMethod("isCreatedAndApprovedToday", null);
}
```

The selectByMethod method is inherited from the Entities class. The method has two parameters. The first parameter provides the name of the selection method in a specific Entity class. The second argument is a List of parameters of the entity selection method. In our example, the **null** list indicates that the isCreatedAndApprovedToday method does not have any parameters. The selectByMethod method in the IEntities interface has two parameters.

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

However, the Entities class has the selectByMethod convenience method with only one parameter.

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

Order of Entities

The simplest order of entities is based on a single property and the ascending or descending sequence. The orderByProperty method is the public method in the IEntities interface of Modelibra. Since the method returns an object of the IEntities type, the casting action transforms the result into the Urls object.

```
public Urls getUrls(String propertyCode, boolean ascending) {
    return (Urls) orderByProperty(propertyCode, ascending);
}
```

For example, urls may be ordered based on the category name with the ascending values.

```
public Urls getUrlsOrderedByCategory(boolean ascending) {
    return getUrls("category", ascending);
}
```

The getUrlsOrderedByName method sorts urls by the name in the default ascending order.

```
public Urls getUrlsOrderedByName() {
    return getUrls("name", true);
}
```

Similarly, urls can be ordered by the creation date with the getOrderByCreationDate method. Note that the data file is already ordered by the creation date, since a newly created url is either older or has the same creation date as the previous urls.

```
public Urls getUrlsOrderedByCreationDate(boolean ascending) {
    return getUrls("creationDate", ascending);
}
```

The getUrlsOrderedByCreationDate method is used in the following method to find the oldest creation date.

```
	public Date getFirstCreationDate() {
		Date firstCreationDate = null;
		Urls orderByCreationDate = getUrlsOrderedByCreationDate(true);
		if (orderByCreationDate.size() > 0) {
			Url webLink = orderByCreationDate.first();
			firstCreationDate = webLink.getCreationDate();
		}
		return firstCreationDate;
	}
```

A property comparator is defined for one property with an additional optional comparator for special cases such as a case insensitive text comparison. The compare method, in the PropertyComparator class, implements the compare method of the java.util.Comparator interface. It returns 0 if two argument objects are equal, > 0 if the first object is greater than the second one, < 0 if the second object is greater than the first one. The private compareEntities method does the actual comparison. It compares two entities by comparing their properties based on the property code provided by one of two constructors.

```
package org.modelibra;

import java.util.Comparator;

public class PropertyComparator implements Comparator {

	private String propertyCode;

	private Comparator comparator;

	public PropertyComparator(String propertyCode) {
		this(propertyCode, null);
	}

	public PropertyComparator(String propertyCode, Comparator comparator) {
		this.propertyCode = propertyCode;
		this.comparator = comparator;
	}

	public int compare(Object object1, Object object2)
			throws IllegalArgumentException {
		IEntity entity1 = (IEntity) object1;
		IEntity entity2 = (IEntity) object2;
		return compareEntities(entity1, entity2);
	}

	private int compareEntities(IEntity entity1, IEntity entity2) {
		int result = 0;
		Object property1 = entity1.getProperty(propertyCode);
		Object property2 = entity2.getProperty(propertyCode);
		if (property1 != null && property2 != null) {
				if (comparator == null) {
					if (property1 instanceof Comparable) {
						result = ((Comparable) property1).compareTo(property2);
					} else if (property2 instanceof Comparable) {
						result = ((Comparable) property2).compareTo(property1);
					} else {
						String string1 = String.valueOf(property1);
						String string2 = String.valueOf(property2);
						result = string1.compareTo(string2);
					}
				} else {
					result = comparator.compare(property1, property2);
				}
			}
			return result;
		}

}
```

For example, urls may be ordered by the category name by ignoring whether the category name is in lower or upper letters.

```
	public Urls getUrlsOrderedByCategoryIgnoringCategoryCase(boolean ascending) {
		CaseInsensitiveStringComparator caseInsensitiveStringComparator = new
			CaseInsensitiveStringComparator();
		PropertyComparator propertyComparator = new PropertyComparator(
				"category", caseInsensitiveStringComparator);
		return getUrls(propertyComparator, ascending);
	}
```

A comparator may be composite consisting of primary and secondary comparators, where each of them may be either a property comparator or a composite comparator. The compare method, in the CompositeComparator class, implements the compare method of the java.util.Comparator interface. It returns 0 if two argument objects are equal, > 0 if the first object is greater than the second one, < 0 if the second object is greater than the first one. The private compareEntities method compares two entities by using first the primary comparator. If the two entities are equal, the method compares them by using the secondary comparator.

```
package org.modelibra;

import java.util.Comparator;

public class CompositeComparator implements Comparator {

	private Comparator primary;

	private Comparator secondary;

	public CompositeComparator(Comparator primary, Comparator secondary) {
		this.primary = primary;
		this.secondary = secondary;
	}

	public int compare(Object object1, Object object2)
			throws IllegalArgumentException {
		IEntity entity1 = (IEntity) object1;
		IEntity entity2 = (IEntity) object2;
		return compareEntities(entity1, entity2);
	}

	private int compareEntities(IEntity entity1, IEntity entity2)
			throws IllegalArgumentException {
		int result = primary.compare(entity1, entity2);
		if (result != 0) {
			return result;
		} else {
			return secondary.compare(entity1, entity2);
		}
	}

}
```

The getUrls method with the Comparator and boolean parameters is a convenience method for ordering urls based on any comparator. The orderByComparator method is the public method in the IEntities interface. Since the method returns an object of the IEntities type, the casting action transforms the result into the Urls object.

```
	public Urls getUrls(Comparator comparator, boolean ascending) {
		return (Urls) orderByComparator(comparator, ascending);
	}
```

For example, urls may be ordered by the category property then by the name property.

```
	public Urls getUrlsOrderedByCategoryThenName(boolean ascending) {
		CompositeComparator compositeComparator = new CompositeComparator(
			new PropertyComparator("category"), new PropertyComparator(
				"name"));
		return getUrls(compositeComparator, ascending);
	}
```

Using SQL in Modelibra

Modelibra uses another OSS called JoSQL [JoSQL] to provide the SQL support in main memory for those developers that just cannot forget SQL. Note that the JosSQL jar file is in the lib directory of the Eclipse project. The following is a quote from the home page of the JoSQL project:

> “JoSQL (SQL for Java Objects) provides the ability for a developer to apply a SQL statement to a collection of Java Objects. JoSQL provides the ability to search, order and group ANY Java objects and should be applied when you want to perform SQL-like queries on a collection of Java Objects.”

The following is a quote from the Introduction of the User Manual:

> “There are many Java applications where the use of SQL statements to perform searching/ordering/grouping and selecting of values from Java Objects would be very valuable. However often times the use of a fully-featured SQL database (free or otherwise) is either in-appropriate and/or costly.”

The getMinCreationDate method in the Urls class uses JoSQL to find a minimal creation date for all urls.

```
public Date getMinCreationDate() {
      Date minCreationDate = null;
      try {
            List<Url> urlsList = getList();
            String sqlStatement =
                  "SELECT min(creationDate) " +
                  "FROM dmeduc.weblink.url.Url " +
                  "LIMIT 1, 1";
            Query query = new Query();
            query.parse(sqlStatement);
            QueryResults queryResults = query.execute(urlsList);
            List<?> selectionList = queryResults.getResults();
            List<?> rowColumnList = (List<?>) selectionList.get(0);
            minCreationDate = (Date) rowColumnList.get(0);
      } catch (Exception e) {
            log.error("Error in Urls.getMinCreationDate: " + e.getMessage());
      }
      return minCreationDate;
}
```

There are two import statements in the Urls class that refer to two classes from JoSQL.

```
import org.josql.Query;
import org.josql.QueryResults;
```

The list of urls is obtained by the getList method from the IEntities interface. This method is used when we want to take entities and leave Modelibra to work at the pure Java level. The SQL statement is composed as a text with the Url class name.

```
SELECT min(creationDate)
FROM dmeduc.weblink.url.Url
LIMIT 1, 1
```

The LIMIT keyword indicates that the result will have only one row and one column. The Query object is constructed and the SQL statement is passed to the parse method of the Query class. The query is then executed on the list of urls and the result is obtained as an object of the QueryResults class. A list of selected urls is extracted by the getResults method. This list is a list of lists, or it is a list of rows where each row is a list of columns. The first row is obtained as the zero member of the list. The first column is obtained as the zero member of the first row. The casting is done to finally get an object of the Date class.

Concept Tests

The new concept tests are added to show how order and selection of entities behave in Modelibra.

The new creation method in the Comments class is added to support better tests in the CommentsTest class. The createComment method has two arguments. There are now two createComment methods in the Comments class. Since they have different number of arguments, they are both valid methods that represent overloading in Java.

```
public Comment createComment(String text, EasyDate easyDate) {
	Comment comment = new Comment(getModel());
	comment.setText(text);
	comment.setCreationDate(easyDate.getDate());
	if (!add(comment)) {
		comment = null;
	}
	return comment;
}
```

The creationDateDescendingOrder method in the CommentsTest class starts four calls of the createComment method. Since their creation is tested in the commentsCreated method, assertions related to the creation of comments are not repeated in this method. The created comments are then ordered by the descending creation date, but not within the comments source object. They are ordered in the orderedComments destination object that is different from the source object. The for iteration is used to show that comments are ordered as required. It is asserted that the previous date does not come before the next date. In other words, the next date comes after the previous date or the dates are equal.

```
@Test
public void creationDateDescendingOrder() throws Exception {
	comments
		.createComment("Modelibra is magic.", new EasyDate(2008, 1, 30));
	comments.createComment("Modelibra is a domain model framework.",
		new EasyDate(2008, 1, 30));
	comments.createComment("Wicket is a web framework.", new EasyDate(2008,
		8, 25));
	comments.createComment("Wicket is a small gate or door.", new EasyDate(
		2008, 4, 3));
	Comments orderedComments = comments
		.getCommentsOrderedByCreationDate(false);

	assertFalse(orderedComments.isEmpty());
	assertEquals(comments.size(), orderedComments.size());
	assertNotSame(comments, orderedComments);

	for (Iterator<Comment> iterator = orderedComments.iterator(); iterator
			.hasNext();) {
		Comment comment = iterator.next();
```

```
			Comment nextComment;
			if (iterator.hasNext()) {
				nextComment = iterator.next();
			} else {
				break;
			}
			Date commentDate = comment.getCreationDate();
			Date nextCommentDate = nextComment.getCreationDate();

			assertTrue(!commentDate.before(nextCommentDate));
		}

	}
```

The selection of comments is based on the text property. All comments that contain the magic keyword are selected (see the getKeywordComments method in the Comments class). The first for iteration asserts that each selected entity contains the given keyword. The second for iteration shows that all comments that contain the keyword are actually selected.

```
	@Test
	public void textKeywordSelection() throws Exception {
		comments
			.createComment("Modelibra is magic.", new EasyDate(2008, 1, 30));
		comments.createComment("Modelibra is a domain model framework.",
			new EasyDate(2008, 1, 30));
		comments.createComment("Wicket is a web framework.", new EasyDate(2008,
			8, 25));
		comments.createComment("Wicket is a small gate or door.", new EasyDate(
			2008, 4, 3));
		String keyword = "magic";
		Comments keywordComments = comments.getKeywordComments(keyword);

		assertFalse(keywordComments.isEmpty());
		assertTrue(comments.getErrors().isEmpty());
		assertNotSame(comments, keywordComments);

		for (Iterator<Comment> iterator = keywordComments.iterator(); iterator
				.hasNext();) {
			Comment comment = iterator.next();

			assertTrue(comment.getText().contains(keyword));
		}

		int counter = 0;
		for (Comment comment : comments) {
			if (comment.getText().contains(keyword))
				counter++;
		}

		assertEquals(keywordComments.size(), counter);
	}
```

The UrlsTest class has several methods that are similar in spirit with the test methods in the CommentsTest class.

The categoryOrder and categoryDescendingOrder methods test the ordering of urls based on the category property. The difference between those two methods is in the comparison assertions. For the ascending category values the assertion is

```
assertTrue(category.compareTo(nextCategory) < 1);
```

while the assertion for descending values is

```
assertTrue(category.compareTo(nextCategory) >= 0);
```

The approvedSelection and notApprovedSelection methods test the selection of urls based on the approved property. The value assertion for the approved selection is

```
assertTrue(url.isApproved());
```

while the assertion for not approved entities is

```
assertTrue(!url.isApproved());
```

The descriptionSomeKeywordSelection tests the selection of entities based on the description property. Only urls with either the component word or the model word found somewhere in the description of the entity are selected.

```
assertTrue(url.getDescription().contains(keyword1) ||
url.getDescription().contains(keyword2));
```

In the descriptionAllKeywordSelection method, only urls with both the component and model words in their description are chosen.

```
assertTrue(url.getDescription().contains(keyword1) &&
     url.getDescription().contains(keyword2));
```

Application Views

When the model is loaded from data in XML files, entities are ordered in the same way they appear in data files. However, this order may be changed. In order to inform ModelibraWicket about the change, some special specific classes with predefined names in predefined packages must be created. This may be done for both display and update of entities. In this chapter the order change is done only in the display of entities.

For example, when comments are displayed they are shown in the descending order of their creation dates. The first comment displayed is the most recent comment. By default, entities are displayed in a table view. For the Comment concept this has been changed in the XML configuration.

```
                    <concept oid="110110120">
                         <code>Comment</code>
                         <entry>true</entry>
                         <!-- model view -->
                         <displayType>list</displayType>
```

After the configuration change, Comments are displayed in a list view, which is the format most frequently used in reports. Since the display type is list, the predefined class name for the Comment concept is:

```
     EntityDisplayListPage.
```

The predefined package name is:

```
     dmeduc.wicket.weblink.comment.
```

This class extends the class with the same name from the org.modelibra.wicket.concept package.

```
package dmeduc.wicket.weblink.comment;

import org.modelibra.wicket.view.View;
import org.modelibra.wicket.view.ViewModel;

import dmeduc.weblink.comment.Comments;

public class EntityDisplayListPage extends
          org.modelibra.wicket.concept.EntityDisplayListPage {

     public EntityDisplayListPage(final ViewModel viewModel, final View view) {
          super(getNewViewModel(viewModel), view);
     }

     private static ViewModel getNewViewModel(final ViewModel viewModel) {
          ViewModel newViewModel = new ViewModel(viewModel);
          Comments comments = (Comments) viewModel.getEntities();
          comments = comments.getCommentsOrderedByCreationDate(false);
          newViewModel.setEntities(comments);
          return newViewModel;
     }

}
```

The static getNewViewModel method is called in the constructor to change the order of entities in the viewModel argument. Since the entities are comments they are casted into the comments object of the Comments class. Then, the comments are ordered by their creation date in the descending order (ascending is false). The ordered comments are set as entities of the new view model that is returned to the constructor and passed to ModelibraWicket in the super method. The super method represents the constructor of the EntityDisplayListPage class in the org.modelibra.wicket.concept package. ModelibraWicket checks if the EntityDisplayListPage specific class in the dmeduc.wicket.weblink.comment package exists, and if so, its constructor will be called instead of the constructor of the generic class.

In the same way that the order of entities may be changed, the selection of entities in a specific view may be done. For example, when displayed, only approved urls are shown. In addition, the approved urls are ordered (ascending is true) by their category names. In order to accomplish this, the two methods are chained. The result of the getApprovedUrls method is used as the current object for the getUrlsOrderedByCategory method. In this way, the time is not lost in sorting non-approved urls.

```
package dmeduc.wicket.weblink.url;

import org.modelibra.wicket.view.View;
import org.modelibra.wicket.view.ViewModel;

import dmeduc.weblink.url.Urls;

public class EntityDisplayTablePage extends
		org.modelibra.wicket.concept.EntityDisplayTablePage {

	public EntityDisplayTablePage(final ViewModel viewModel, final View view) {
		super(getNewViewModel(viewModel), view);
	}

	private static ViewModel getNewViewModel(final ViewModel viewModel) {
		ViewModel newViewModel = new ViewModel(viewModel);
		Urls urls = (Urls) viewModel.getEntities();
		urls = urls.getApprovedUrls().getUrlsOrderedByCategory(true);
		newViewModel.setEntities(urls);
		return newViewModel;
	}

}
```

Note that the predefined class name is EntityDisplayTablePage, since in the XML configuration of the Url concept, the default display type (table) has not been changed.

UTF-8

The following quote is from Wikipedia [UTF]:

“UTF-8 (8-bit Unicode Transformation Format) is a variable-length character encoding for Unicode created by Ken Thompson and Rob Pike. It is able to represent any universal character in the Unicode standard, yet is backwards compatible with ASCII. For this reason, it is steadily becoming the preferred encoding for email, web pages, and other places where characters are stored or streamed,”

With UTF-8, that Modelibra supports in data files, different accents can be used. For example, the following is a url about my personal web site. There are three accents in my name.

```
	<url oid="1170620652940">
		<name>Dženan Riđanović</name>
		<link>http://drdb.fsa.ulaval.ca/dr/</link>
```

```
        <description>...</description>
        <creationDate>2007-01-04</creationDate>
        <updateDate>2007-02-09</updateDate>
        <approved>true</approved>
        <category>Personal</category>
    </url>
```

In order to use different accents in the Java code within Eclipse, the default project encoding must be changed to UTF-8: *Project/Properties/Text file encoding/Other*.

Eclipse Project

The DmEduc spiral in this chapter is called DmEduc-03 and it is initially created by copying the DmEduc-02 spiral. Before the DmEduc-03 project was transferred to the Subversion repository, the lib directory had been deleted, if the directory had not been omitted in the copying action. The following two SVN properties were added to the WEB-INF container directory by using twice the *Team/Set Property...* pop-up menu item. This was done after the project was shared at the repository by using the *Team/Share Project...* pop-up menu item.

```
svn:ignore
classes
logs

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

At the beginning of the sharing process, the classes and logs directories were excluded from the transfer of files. After the transfer of files, the project was updated locally and the library files were copied from the web address given in the svn:externals property. The next time when the DmEduc project is updated, and if there is a change in the libraries of the ModelibraWicketSkeleton project, this change will be reflected in the libraries of the DmEduc project. In this way, the precious space at the JavaForge repository is not wasted with multiple copies of the jar files, nor with classes and logs.

The same is done with the css directory. After the directory had been deleted, if the directory had not been omitted in the copying action, the following property was added to the project's root directory.

```
svn:externals
css http://svn.javaforge.com/svn/Modelibra/trunk/ModelibraWicketCss/css
```

After the checkout of the the DmEduc-03 project from JavaForge, be sure that the classes and logs directories do exist in the WEB-INF directory. Also verify the *Java Build Path* in the project properties. Do not be tempted to use the *Add External JARs...* button to add the libraries from another project to the DmEduc-03 project. If you do so, you may have some strange problems with class loading.

Modelibra Interfaces

The following methods in the IEntities interface have been used in the DmEduc-03 spiral for the first time: selectBySelector, selectByMethod, orderByProperty and orderByComparator.

```
public interface IEntity<T extends IEntity> extends Serializable, Comparable<T> {

	public T copy();

}

public interface IEntities<T extends IEntity> extends Serializable, Iterable<T> {

	public IDomainModel getModel();

	public boolean add(T entity);

	public boolean update(T entity, T updateEntity);

	public T retrieveByOid(Oid oid);

	public T retrieveByProperty(String propertyCode, Object property);

	public IEntities<T> selectByProperty(String propertyCode, Object property);

	public IEntities<T> selectBySelector(ISelector selector);

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

	public IEntities<T> orderByProperty(String propertyCode, boolean ascending);

	public IEntities<T> orderByComparator(Comparator comparator,
			boolean ascending);

	public Errors getErrors();

}
```

Travel Impression

By using the TravelImpression-01 spiral from the previous chapter, the model has changed slightly. The subject property in the Message concept is now mandatory. When messages were displayed in a table view, it became obvious that it would be more informative to show a message subject and not the beginning of the message text. However, if the message subject was not required, the table view of messages would be even harder to grasp. Similarly, the title property in the Photo concept is now mandatory.

The new spiral is created (TravelImpression-02) and the code is generated. Since we have not changed the XML configuration in the previous spiral, nothing is lost. However, in order to facilitate future changes of the model and preserve specific adjustments in the XML configuration, the inheritance is used for XML configurations.

The XML configuration for the model in Figure 5.2 is generated by ModelibraModeler. The configuration is saved in the config directory of the TravelImpression-02 project as the reusable-domain-config.xml file. From now on, this configuration file will not be modified manually. However, if the model changes, the new reusable configuration will be regenerated by ModelibraModeler. Any adjustments to the XML configuration of the domain model will be done manually in the specific-domain-config.xml file that is also located in the same config directory.

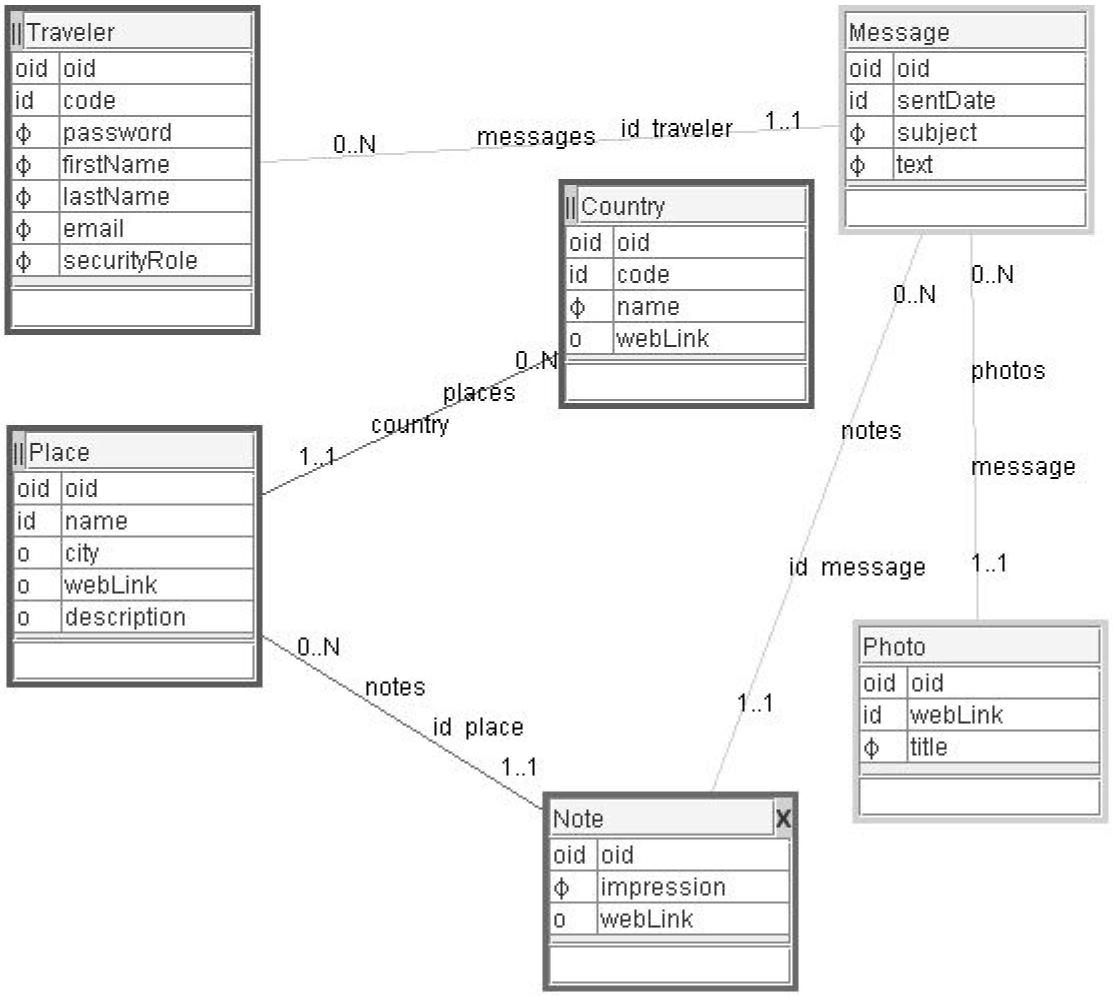

Figure 5.2. Travel Impression model

The specific model configuration inherits the reusable model configuration.

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

<domains>

    <domain oid="1189015928269">
```

```
<code>Travel</code>
<type>Specific</type>

<models>

     <model oid="1190053285158">
           <code>Impression</code>
           <extension>true</extension>
           <extensionDomain>Travel</extensionDomain>
           <extensionDomainType>Reusable</extensionDomainType>
           <extensionModel>Impression</extensionModel>

           <concepts>

                 <concept oid="1189698214159">
                       <code>Traveler</code>
                       <extension>true</extension>
                       <extensionConcept>Traveler</extensionConcept>

                       <properties>
                             <property oid="1189698214161">
                                   <code>code</code>
                                   <extension>true</extension>
                                   <extensionProperty>code</extensionProperty>

                                   <essential>false</essential>
                             </property>
                             <property oid="1189698214163">
                                   <code>firstName</code>
                                   <extension>true</extension>
                                   <extensionProperty>
                                         firstName
                                   </extensionProperty>

                                   <essential>false</essential>
                             </property>
                             <property oid="1189698214164">
                                   <code>lastName</code>
                                   <extension>true</extension>
                                   <extensionProperty>
                                         lastName
                                   </extensionProperty>

                                   <essential>false</essential>
                             </property>
                             <property oid="1189698214165">
                                   <code>name</code>
                                   <propertyClass>
                                         java.lang.String
                                   </propertyClass>
                                   <derived>true</derived>
```

```
                    <essential>true</essential>
                </property>
                <property oid="1189698214166">
                    <code>email</code>
                    <extension>true</extension>
                    <extensionProperty>email</extensionProperty>

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

        </concept>

        <concept oid="1189698214190">
            <code>Note</code>
            <extension>true</extension>
            <extensionConcept>Note</extensionConcept>

            <properties>
                <property oid="1189698214199">
                    <code>placeOid</code>
                    <extension>true</extension>
                    <extensionProperty>
                        placeOid
                    </extensionProperty>

                    <referenceDropDownLookup>
                        false
                    </referenceDropDownLookup>
                </property>
            </properties>

        </concept>

        <concept oid="1189698214194">
            <code>Message</code>
            <extension>true</extension>
            <extensionConcept>Message</extensionConcept>

            <properties>
               <property oid="1189698214196">
                    <code>sentDate</code>
                    <extension>true</extension>
                    <extensionProperty>
                        sentDate
                    </extensionProperty>

                    <essential>true</essential>
                 </property>
               <property oid="1189698214197">
                    <code>subject</code>
                    <extension>true</extension>
```

```
                  <extensionProperty>
                        subject
                  </extensionProperty>
                  <required>true</required>

                  <essential>true</essential>
            </property>
            <property oid="1189698214198">
                  <code>text</code>
                  <extension>true</extension>
                  <extensionProperty>text</extensionProperty>

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

</concept>

<concept oid="1189698236077">
      <code>Photo</code>
      <extension>true</extension>
      <extensionConcept>Photo</extensionConcept>

      <properties>
            <property oid="1189698788794">
                  <code>title</code>
                  <extension>true</extension>
                  <extensionProperty>title</extensionProperty>
                  <required>true</required>

                  <essential>true</essential>
            </property>
      </properties>

</concept>

<concept oid="1189698402632">
      <code>Country</code>
      <extension>true</extension>
      <extensionConcept>Country</extensionConcept>

      <properties>
            <property oid="1189698432666">
                  <code>code</code>
                  <extension>true</extension>
                  <extensionProperty>code</extensionProperty>

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

</concept>
```

```
                        </concepts>

                    </model>

                </models>

            </domain>

        </domains>
```

The Travel domain code stays the same in both reusable and specific configuration files. The only difference is in the domain type. In the reusable configuration the domain type is Reusable and in the specific configuration the domain type is Specific. The only Modelibra requirement is that all domains used in the same project must have a different combination of domain code and domain type. However, the type value may be anything that easily differentiates one domain from another. In this case the most natural values were Reusable and Specific. Using the inheritance terminology, the Reusable type could have been renamed as the Generic type. Note that the domain code is the same in both configurations since it is the same domain.

There is no inheritance at the domain level. The inheritance starts at the model level. The Impression model in the specific configuration extends the Impression model in the reusable configuration. The inheritance link between the two models is established through their unique values for the combination of domain code and domain type.

```
                <model oid="1190053285158">
                    <code>Impression</code>
                    <extension>true</extension>
                    <extensionDomain>Travel</extensionDomain>
                    <extensionDomainType>Reusable</extensionDomainType>
                    <extensionModel>Impression</extensionModel>
```

The Traveler concept in the specific Impression model inherits the Traveler concept in the reusable Impression model. The context of a concept inheritance is determined by the model inheritance.

```
                        <concept oid="1189698214159">
                            <code>Traveler</code>
                            <extension>true</extension>
                            <extensionConcept>Traveler</extensionConcept>
```

The code property in the specific Traveler concept extends the code property in the reusable Traveler concept. The context of a property inheritance is determined by the concept inheritance.

```
                                <property oid="1189698214161">
                                    <code>code</code>
                                    <extension>true</extension>
                                    <extensionProperty>code</extensionProperty>

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

The essential XML element declaration appears in both specific and reusable configurations. In the reusable configuration the essential declaration is true, while in the specific configuration the essential declaration is false. In this way, the specific essential declaration overrides the reusable essential declaration. Similarly, for both lastName and firstName properties, the specific essential declaration overrides the reusable essential declaration. The last and first names will not be displayed in a table view of travelers. Instead, the new name property will appear. The name property is a derived property. A derived property will not be saved. As such, it does not have the set method. The name property will be derived each time the specific getName method from the Traveler class is called.

```
package travel.impression.traveler;

import org.modelibra.IDomainModel;

public class Traveler extends GenTraveler {

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

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

	public String getName() {
		String name = "";
		String lastName = getLastName();
		if (lastName != null) {
			name = lastName;
		}
		String firstName = getFirstName();
		if (firstName != null) {
			name = name + ", " + firstName;
		}
		return name;
	}

}
```

The getName method is added after the SPECIFIC CODE comment. The next time when the model changes, only the Gen classes will be regenerated. In this way, the added specific code such as the getName method will stay intact. Similarly, the next time when the model changes in ModelibraModeler, only the reusable-domain-config.xml will be regenerated. Hence, the added specific configuration such as the name property declaration will not be lost.

The Note concept in the specific configuration inherits all property and neighbor declarations from the reusable configuration. Its placeOid reference property, which represents the external place neighbor, inherits all declarations from the corresponding property in the reusable configuration. The only declaration that is overridden with the new false value is referenceDropDownLookup. This means that a lookup of a place for a note will not be done with the drop-down choice component. The lookup will be done in a new page that contains a table of places to choose from.

All other new declarations are for the essential element to override a value from the reusable configuration.

For a concept that has not changed in any way at the concept, property or neighbor level, the concept is not repeated in the specific configuration, since it is inherited by the model. This is the case with the Place concept that does not appear in the specific configuration, but appears as the entry point in the Web application. The same is true for a property or a neighbor. In our example, none of neighbors has been modified. Thus, no neighbor is repeated in the specific configuration, since all neighbors are inherited by their source concepts.

The Place concept is fully defined in the reusable configuration. It has the countryOid reference property that represents the external country neighbor. A lookup of a country will be done by the drop-down choice component (true for the referenceDropDownLookup element).

```
<concept oid="1189698214168">
      <code>Place</code>
      <entitiesCode>Places</entitiesCode>
      <entry>true</entry>

      <properties>
            <property oid="1189698484746">
                  <code>countryOid</code>
<propertyClass>
                        java.lang.Long
                  </propertyClass>
                  <required>true</required>
                  <reference>true</reference>
                  <referenceNeighbor>
                        country
                  </referenceNeighbor>

                  <essential>false</essential>
                  <referenceDropDownLookup>
                        true
                  </referenceDropDownLookup>
            </property>
```

By default, the drop-down choice component displays the neighbor concept's unique combination. The Country identifier is the code property. In order to replace the country code by the country name as a display value in the drop-down choice component, the toString method in the Country class is overridden.

```
public String toString() {
      return getName();
}
```

The specific configuration may be generated at the beginning by the generateSpecificDomainConfig method of the DmModelibraGenerator class. After the first generation, the specific configuration is never regenerated. Specific changes are added by hand.

The next time when the model is changed in ModelibraModeler, the new configuration will be generated and the old reusable-domain-config.xml will be replaced by the new model configuration. The Eclipse project will be refreshed to have the access to the new version of the reusable-domain-config.xml file. The code will be generated by using the main method of the DmGenerator class in the dm.gen package. However, this time the new methods will be called.

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

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

      // dmGenerator.getDmModelibraWicketGenerator().generate();
      dmGenerator.getDmModelibraWicketGenerator()
           .generateModelibraWicketAppProperties();
}
```

The new methods are generateModelibraGenClasses and generateModelibraWicketAppProperties. For the changed model, only the Gen classes will be generated. However, it is important to verify if there is a need to adapt the specific code to the new changes in the model.

Often, a change in the model will require new properties in the application properties file. In order to keep specific changes at the application properties level, the new application properties file is created. In our case, the new file is called TravelApp_en. The _en stands for the English version. The French version would have the _fr extension. All specific changes will be done in files with the _ postfix. Using the inheritance terminology, the properties in files with the _ postfix are specific, while the properties in the application properties file without the _ postfix are generic (or reusable to make a parallel with the XML configuration inheritance). At the application level, Wicket looks for a property key in one of the files with the _ postfix, and if the key is not found, Wicket will consult the application properties file. In this way, the specific property overrides the generic property. The regeneration of application properties will be done only in the TravelApp.properties file. Therefore, the specific changes will not be lost by the regeneration process.

The following is the (re)generated content of the TravelApp.properties file.

```
Travel=Travel
Travel.title=Travel Domain
Travel.description=Modelibra - Travel Domain Models.

Impression=Impression
Impression.title=Impression Model
Impression.description=Modelibra - Travel Domain - Impression Model

Traveler=Traveler
Travelers=Travelers
Traveler.id=Traveler identifier: ([code] [])
Traveler.id.unique=Traveler identifier ([code] []) is not unique.
Traveler.code=Code
Traveler.code.required=Code is required.
Traveler.code.length=Code is longer than 16.
Traveler.password=Password
Traveler.password.required=Password is required.
Traveler.password.length=Password is longer than 16.
Traveler.firstName=FirstName
Traveler.firstName.required=FirstName is required.
Traveler.firstName.length=FirstName is longer than 32.
Traveler.lastName=LastName
Traveler.lastName.required=LastName is required.
Traveler.lastName.length=LastName is longer than 32.
Traveler.name=Name
Traveler.email=Email
Traveler.email.required=Email is required.
Traveler.email.length=Email is longer than 80.
Traveler.email.validation=Email is not a valid org.modelibra.type.Email value.
Traveler.securityRole=SecurityRole
Traveler.securityRole.required=SecurityRole is required.
Traveler.securityRole.length=SecurityRole is longer than 16.
Traveler.messages=Messages

Place=Place
Places=Places
Place.id=Place identifier: ([name] [])
Place.id.unique=Place identifier ([name] []) is not unique.
Place.countryOid=CountryOid
Place.countryOid.required=CountryOid is required.
Place.name=Name
Place.name.required=Name is required.
Place.name.length=Name is longer than 64.
Place.city=City
Place.city.length=City is longer than 32.
Place.webLink=WebLink
Place.webLink.length=WebLink is longer than 96.
Place.webLink.validation=WebLink is not a valid java.net.URL value.
Place.description=Description
Place.description.length=Description is longer than 510.
Place.notes=Notes
Place.country=Country

Note=Note
Notes=Notes
Note.id=Note identifier: ([] [message, place])
Note.id.unique=Note identifier ([] [message, place]) is not unique.
Note.placeOid=PlaceOid
Note.placeOid.required=PlaceOid is required.
Note.impression=Impression
Note.impression.required=Impression is required.
Note.impression.length=Impression is longer than 1020.
Note.webLink=WebLink
Note.place=Place
Note.message=Message
```

```
Message=Message
Messages=Messages
Message.id=Message identifier: ([sentDate] [traveler])
Message.id.unique=Message identifier ([sentDate] [traveler]) is not unique.
Message.sentDate=SentDate
Message.sentDate.required=SentDate is required.
Message.sentDate.length=SentDate is longer than 16.
Message.subject=Subject
Message.subject.required=Subject is required.
Message.subject.length=Subject is longer than 64.
Message.text=Text
Message.text.required=Text is required.
Message.text.length=Text is longer than 4080.
Message.notes=Notes
Message.photos=Photos
Message.traveler=Traveler

Photo=Photo
Photos=Photos
Photo.id=Photo identifier: ([webLink] [])
Photo.id.unique=Photo identifier ([webLink] []) is not unique.
Photo.webLink=WebLink
Photo.webLink.required=WebLink is required.
Photo.webLink.length=WebLink is longer than 96.
Photo.webLink.validation=WebLink is not a valid java.net.URL value.
Photo.title=Title
Photo.title.required=Title is required.
Photo.title.length=Title is longer than 64.
Photo.message=Message

Country=Country
Countries=Countries
Country.id=Country identifier: ([code] [])
Country.id.unique=Country identifier ([code] []) is not unique.
Country.code=Code
Country.code.required=Code is required.
Country.code.length=Code is longer than 16.
Country.name=Name
Country.name.required=Name is required.
Country.name.length=Name is longer than 64.
Country.webLink=WebLink
Country.webLink.length=WebLink is longer than 96.
Country.webLink.validation=WebLink is not a valid java.net.URL value.
Country.places=Places
```

The following is the content of the TravelApp_en.properties file with only overridden properties.

```
Traveler.firstName=First Name
Traveler.firstName.required=First Name is required.
Traveler.firstName.length=First Name is longer than 32.
Traveler.lastName=Last Name
Traveler.lastName.required=Last Name is required.
Traveler.lastName.length=Last Name is longer than 32.
Traveler.securityRole=Security Role
Traveler.securityRole.required=Security Role is required.
Traveler.securityRole.length=Security Role is longer than 16.

Place.countryOid=Country
Place.countryOid.required=Country is required.
Place.name=Place
Place.name.required=Place is required.
Place.name.length=Place is longer than 64.
Place.webLink=Web Link
Place.webLink.length=Web Link is longer than 96.
Place.webLink.validation=Web Link is not a valid java.net.URL value.

Note.placeOid=Place
Note.placeOid.required=Place is required.
Note.webLink=Web Link

Message.sentDate=Sent Date
Message.sentDate.required=Sent Date is required.
Message.sentDate.length=Sent Date is longer than 16.

Photo.webLink=Web Link
Photo.webLink.required=Web Link is required.
Photo.webLink.length=Web Link is longer than 96.
Photo.webLink.validation=Web Link is not a valid java.net.URL value.

Country.name=Country
Country.name.required=Country name is required.
Country.name.length=Country name is longer than 64.
Country.webLink=Web Link
Country.webLink.length=Web Link is longer than 96.
Country.webLink.validation=Web Link is not a valid java.net.URL value.
```

After the properties are regenerated in the TravelApp.properties file by the generateModelibraWicketAppProperties method, it is important to verify if specific changes done in the TravelApp_en.properties file hould be adapted to the newly regenerated content of the TravelApp_en.properties file.

Summary

Selection and order of entities is the main theme of this chapter. A subset of entities is selected from the source entities by defining a selector. Only those entities that satisfy the selector are selected. The result is an object of destination entities. Thus, further selections on the result may be applied again. A property selector is defined for one property and one of many relational operators such as contain, greater than, match, etc. A selector may be composite consisting of other selectors related by and, or, not logical operators. There is also a selection mechanism where a user provided Java selection method, which returns a boolean type value, is applied to the source entities. The selection method may be quite complex to satisfy various application needs.

Entities may be reordered by applying a comparator. The simplest order is based on a single property and the ascending or descending sequence. A comparator may be composite consisting of primary and secondary comparators, where each of them may be a composite comparator again.

Different model tests are done to show how entities can be selected and ordered in different ways.

The TravelImpression model is slightly changed to show how the reusable configuration can be regenerated leaving the specific configuration unchanged. As specific classes inherit generic classes, the specific configuration inherits the reusable configuration. Similarly, generic properties of the web application are inherited by properties of specific natural languages. In all cases, the generic (or reusable) code is generated and regenerated, while the specific code is not changed by the code generation but by developers to customize both the domain model and its views.

The next section of the book is about relationships in a domain model. The next chapter will explain how a relationship of the one-to-many type is handled both in Modelibra and ModelibraWicket.

Questions

1. What are different ways of selecting entities in Modelibra?

2. What are different types of selectors in Modelibra?

3. How to select entities based on the parent concept?

4. What are different ways of ordering entities in Modelibra?

5. What are different types of comparators in Modelibra?

6. Should SQL be used in Modelibra?

Exercises

Exercise 5.1.

Make a test to retrieve only one url.

Exercise 5.2.

Make a test to select entities of your choice by using a new composite selector.

Exercise 5.3.

Make a test to select entities of your choice by using a new selection method.

Exercise 5.4.

Make a test to order entities of your choice by using a new composite comparator.

Web Links

[JoSQL] SQL for Java Objects
http://josql.sourceforge.net/

[UTF] UTF definition from Wikipedia
http://en.wikipedia.org/wiki/UTF-8