Chapter 2: Simple Domain Model

The objective of this chapter is to create a simple domain model in Modelibra. There is no web application for the model. Model objects are created temporarily and not saved. The domain model is designed with ModelibraModeler and transformed manually into XML descriptions in the domain configuration file. The transformation is done manually for pedagogical reasons. Of course, it is possible from ModelibraModeler to generate XML configurations automatically. The domain configuration file is read by Modelibra, converted to Java objects and used to support user actions on the domain model.

In this chapter, the domain model has only one concept. As a consequence, the Java code in this chapter is simple. There are five Java classes essential for Modelibra: one for the domain configuration, one for the domain, one for the model, and two for the Url concept. Out of those two concept classes, one describes a single entity and the other represents a collection of entities. The five classes are essential to make the model functional. Everything else is inherited from Modelibra.

Eclipse is used to create a different project for each spiral. A project consists of Java packages and classes, and other resources such as XML configuration files. A project has a predefined structure of directories that help organize software in an effective way.

Domain Model

The domain model consists of one concept: Url. The concept has only three properties (Figure 2.1). There are no neighbors, since there are no relationships with other concepts. The properties are: oid, link and description. The oid property is an artificial object identifier that each concept must have and that does not have any meaning to a user. Its value is unique universally. The oid property is inherited from Modelibra and completely handled by the framework. The link property is required, while the

description property is not.

Figure 2.1. Url concept

Since the Url concept is the entry into the model, the || sign in front of the concept name is used in ModelibraModeler. As an entry concept, the Url concept's entities may be accessed directly from the model.

Java

Modelibra is developed in Java (version 6) Standard Edition (SE). If you do not have Java SE Development Kit (JDK), you may want to download it [Java]. JDK has a compiler and other development utilities. Java has also Java Runtime Environment (JRE) that is a part of JDK, but can be downloaded separately [JRE]. JRE is actually Java Virtual Machine used to execute compiled Java code. Internet browsers use JRE to execute Java code.

In general, a software project done with Java has multiple classes organized into different packages. A package name consists of sub-names separated by a dot. A package in Java represents a path of directories starting with the directory indicated as the source directory (often named src). A package sub-name corresponds to a directory name.

It is rare to develop a new software from scratch. Different libraries of Java code are used for different purposes. A library of compiled Java classes organized into packages is compacted into a file with the .jar extension, often called a JAR file. A JAR file is similar to a ZIP file. The library of Modelibra is modelibra.jar.

The source code used in this chapter is in the DmEduc-00 Eclipse project that may be checked out from the Modelibra repository. DmEduc is the domain name and 00 is the initial spiral number.

Eclipse Project

Eclipse is an Integrated Development Environment (IDE). It is an Open Source Software (OSS). It is done in Java and used mostly by Java developers, although, with its plug-in technology, it may be used with other programming languages and editors.

To configure your project with respect to the source code, the compiled code and Java libraries (JAR files) use *Project/Properties/Java Build Path*. The Java Build Path window has several tags. The Source tag is used to define where the source code is and where the classes will be compiled. The Libraries tag shows libraries used in the project. A new library may be added to the project by using the *Add JARs...* button. To change the compliance level to the version 6 of Java, use *Window/Preferences.../Java/Java Compiler*. However, be sure that you have the latest version of JRE, as a part of JDK, in *Window/Preferences.../Java/Installed JREs*.

Eclipse has a concept of a workspace. A workspace is a directory where you keep one or more projects. Each project within the same workspace must have a unique name. You may create a new workspace by switching to a different workspace (*File/Switch Workspace/Other.../Browse...*)

Eclipse is used to create a different project for each spiral. The DmEduc-00 project may be checked out from the SVN repository. First open the SVN Repository Exploring perspective, either by following the *Window/Open Perspective/Other...* menu items or by using the >> shortcut in the upper right corner of the Eclipse main window. Open the trunk directory and check out first the ModelibraWicketSkeleton project. This project has in the lib directory the latest versions of all necessary JAR files.

The DmEduc-00 spiral project is in the DmEduc-00 directory that is located in the Educ directory. You can check that easily by opening the root directory of the project in the operating system and by

locating the .project file. Select the DmEduc-00 project root directory in Eclipse and in the pop-up menu choose the *Checkout...* menu item. A copy of the project, linked to the server version of the project, will appear on your computer in the current workspace. You can check the local existence of the project by switching to the Java perspective by using the >> shortcut in the upper right corner of the Eclipse main window

```
DmEduc-00
      classes
      config
      doc
      lib
      logs
      mm
      src
      test
      copy-lib.xml
      .settings
      .svn
      .classpath
      .project
```

The source code is in the src directory. The compiled code is in the classes directory that you do not see directly in Eclipse. Use *Project/Properties/Java Build Path/Source* and you will find that classes is the *Default output folder*. The JAR library files will appear in the lib directory after you copy them from the ModelibraWicketSkeleton project . The copy may be done by the Ant script in the copy-lib.xml file. The script is executed by selecting the file and using in the pop-up menu the *Run As/Ant Build* item. ModelibraModeler models are in the mm directory. The configuration files are in the config directory. Error and information messages may be found in the logs directory. The project minimal documentation is in the doc directory. The Eclipse project configuration is in the .project and .classpath files. The Eclipse preferences, generated by Eclipse, are in the .settings directory. The SVN information is in the .svn directory. Files that start with a dot are not seen directly in Eclipse. You can check for their existence in a file browser of the operating system.

The source code in Java consists of three packages: dmeduc, dmeduc.weblink and dmeduc.weblink.url. The packages correspond to the local paths of directories within the project directory. All packages in the project start with the same prefix: dmeduc, which is also the domain name.

The domain classes are in the dmeduc directory. The DmEduc class (DmEduc.java file) represents the domain. The DmEducConfig class represents the domain configuration.

The model classes are in the weblink directory that is within the dmeduc directory. The WebLink class represents the model.

The concept classes are in the url directory that is within the weblink directory. A single concept has two Java classes, one that represents a single entity and the other that represents a collection of entities. The Url class represents a single web link, while the Urls class represents a group of web links.

The DmEducConfig class stands for the domain configuration. The DmEduc class accepts the domain configuration and creates the WebLink model. The following is the project's complete hierarchy of

source code directories and files.

```
src
      dmeduc
            DmEduc.java
            DmEducConfig.java
            weblink
                  WebLink.java
                  url
                        Url.java
                        Urls.java
```

The compiled Java classes are similarly organized in the classes directory.

```
classes
      dmeduc
            DmEduc.class
            DmEducConfig.class
            weblink
                  WebLink.class
                  url
                        Url.class
                        Urls.class
```

The various configuration files are located in the config directory of the project. The domain configuration is in the specific-domain-config.xml file. The domain-config.dtd file contains definitions of valid XML elements used in the specific-domain-config.xml file. It does not contain only definitions of XML elements used in this spiral, but definitions of all valid XML elements in Modelibra. This file will be identical in all spirals. Its content should not be changed. Modelibra uses the log4j [log4j] utility for logging different types of messages. The utility is configured in the log4j.xml file based on the definitions in the log4j.dtd file.

```
config
      domain-config.dtd
      log4j.dtd
      log4j.xml
      specific-domain-config.xml
```

Messages logged by Modelibra can be found in the logs directory of the project.

```
logs
      error.html
      info.html
```

The following are the JAR libraries used by the DmEduc-00 project.

```
lib
      commons-beanutils-core-1.7.jar
      commons-logging-1.1.jar
      dom4j-1.6.1.jar
      junit-4.4.jar
```

```
        log4j-1.2.13.jar
        mail.jar
        Modelibra.jar
```

The commons-beanutils-core [beanutils] is used for type conversion of values. The commons-logging [logging] and log4j [log4j] libraries are used to support the logging messages. The commons-logging is an ultra-thin interface that hides different logging implementations. The log4j is a specific logging implementation. A library that uses the commons-logging interface can be used with any logging implementation at runtime. The dom4j [dom4j] is a library for working with XML. The JUnit [JUnit] is a library to support tests from the test directory of the project. The mail library [JavaMail] provides protocol independent framework for creating email messages. It is there because the Modelibra library requires its presence.

The mm directory contains the DmEduc.mm file of the model designed by ModelibraModeler. The other two text files are the export of the model in the form of a diagram and property types used in the diagram.

```
    mm
        DmEduc.diagram
        DmEduc.mm
        DmEduc.type
```

Few notes about the spiral may be consulted in the DmEduc.txt file in the doc directory.

```
    doc
        DmEduc.txt
```

The project configuration can be found in XML files created by Eclipse. Files that start with a dot are not seen directly in Eclipse. You can check for their existence in a file browser of the operating system.

```
DmEduc-00
    .classpath
    .project
```

XML stands for E**X**tensible **M**arkup **L**anguage [XML]. It is a markup language much like HTML. It was designed to describe data. XML tags are not predefined as is the case with HTML. In XML, tag names reflect terms from a domain of your work. XML is often used for software configurations.

The DmEduc-00 project name is declared in the .project file.

```
<?xml version="1.0" encoding="UTF-8"?>
<projectDescription>
    <name>DmEduc-00</name>
    <comment></comment>
    <projects>
    </projects>
    <buildSpec>
        <buildCommand>
            <name>org.eclipse.jdt.core.javabuilder</name>
            <arguments>
```

```
                </arguments>
            </buildCommand>
        </buildSpec>
        <natures>
            <nature>org.eclipse.jdt.core.javanature</nature>
        </natures>
</projectDescription>
```

The first line of the project XML document defines the XML version and the character encoding used. In this case the document conforms to the 1.0 specification of XML and uses the UTF-8 [UTF] character set. The next line describes the root element of the project document. There must be only one root element in the XML document. The root element, by its projectDescription name, indicates that the XML document is about a project description. An element consists of an opening and closing tag.

```
<projectDescription>
...
</projectDescription>
```

The closing tag starts with the / character. There is always a closing tag for its opening tag. In XML, element names are case sensitive. They are written in small letters. If an element has a composed name (composed of different words), the first character of the second and subsequent words may be a capital letter.

The project name is declared between the tags of the name element. Note that XML elements are properly nested within each other, starting with the root element.

The DmEduc-00 project directory and file paths are specified in the .classpath file. The source code local path (local with respect to the project root directory) is src. The local directory where the source code is compiled is classes. All JAR files used in the project's lib directory are declared in the .classpath file. The content of this file is changed by Eclipse when some of those declarations are modified within Eclipse.

```
<?xml version="1.0" encoding="UTF-8"?>
<classpath>
<classpathentry kind="src" path="src"/>
<classpathentry kind="src" path="test"/>
<classpathentry kind="con" path="org.eclipse.jdt.launching.JRE_CONTAINER"/>
<classpathentry kind="lib" path="lib/commons-logging-1.1.jar"/>
<classpathentry kind="lib" path="lib/dom4j-1.6.1.jar"/>
<classpathentry kind="lib" path="lib/log4j-1.2.13.jar"/>
<classpathentry kind="lib" path="lib/mail.jar"/>
<classpathentry kind="lib" path="lib/Modelibra.jar">
<classpathentry kind="lib" path="lib/juni-4.4.jar">
<classpathentry kind="output" path="classes"/>
</classpath>
```

Attributes of XML elements are represented in name/value pairs such as kind="src". In XML, the attribute value must always be quoted. Attributes are placed inside the opening tag of an element.

If you want to create your own Eclipse project based on the DmEduc-00 project, you would need to

export the project (*File/Export...*) as an archive file (*General/Archive File*) or as a hierarchy of directories and files (*General/File System*). The archive file will be in the ZIP format. By exporting the project, you would obtain a copy of the project but without the SVN information. If you used an archive file, unzip the file into one of workspaces of Eclipse. If you want to have both projects in the same workspace, in order to avoid the project name conflict, you must change (e.g., by an ordinary text editor) the project name in the .project file, before importing the project into Eclipse.

The following are steps to import an existing project into Eclipse.

1. Open Eclipse.
2. Use the *File* menu and the *Import...* menu choice.
3. *General/Existing Projects into Workspace*, then click on the *Next* button.
4. Click on the *Browse...* button.
5. *Select the application directory* where the Eclipse project file is located, then click on the *OK* button.
6. If there are no problems, click on the *Finish* button.
7. If there are problems, rename the project before importing it into Eclipse, and restart.

Of course, you can create a project from scratch by using *File/New/Project...* in Eclipse. After that, select the project in the *Java* perspective and use *File/New/Folder*, *File/New/Package*, *File/New/Class* to create what is needed.

You can now make changes in a local project without the concern about links with the SVN repository. Anyway, you do not have a permission to commit changes (to the SVN repository at JavaForge) in the original DmEduc or any other Modelibra project. However, you can always get the latest version of the original project by using the *Update* command. In order to update the local version, select the project in the *Java* perspective and in the pop-up menu select *Team/Update*. There is also the *Synchronize with Repository* menu choice that will give you the *Team Synchronizing* perspective. In that perspective you can examine changes in detail before updating the local version.

If you want to *Commit* changes that you have made to your local project that is not linked to SVN, you must have an access to an SVN repository. The Assembla web site [Assembla] provides SVN and other free services for developers. Create your account and your first space with at least the Trac/SVN option. In the Trac/SVN tag you will find the url of your space that you can use to save projects. In *Java* perspective of Eclipse, you would select a project and in the pop-up menu you would choose *Team/Share Project...* with *SVN/Create a new repository location* where you would provide the url address from Assembla. It is important that you do not send *classes* from your local project to the SVN repository, since it would be a waste of precious SVN space. You could also use the *SVN Repository Exploring* perspective to create an SVN link by *File/New/Other.../SVN/Checkout Projects from SVN*. In general, you should have at least *trunk* and *tags* directories in your SVN space. The trunk directory is used to keep the latest versions of your projects. The tags directory is used to keep the previous versions. Once your project is linked to SVN, you would use *Team/Commit* to send changes in your local version to the repository. In order to have the same version number locally and at the SVN server, use *Team/Update* immediately after *Commit*. This need to use *Update* after each *Commit* is more obvious if you use the *Team Synchronizing* perspective.

ModelibraModeler

ModelibraModeler is a graphical tool for designing domain models and generating code from a model. You can use ModelibraModeler with the help of Java Web Start [JWS]. Java Web Start is a part of Java. It is used to install a Java software on a client computer by using the latest version of the software from a server computer. Since this book is for developers, the software in this book will be used from a perspective of a developer. Checkout the ModelibraModeler project from the code repository at JavaForge. Select the Start.java class in the org.modelibra.modeler.app.context package that has the main method, pop up the menu and use the *Run As/Java Application* item.

The main window of ModelibraModeler contains an empty table of diagrams. Use the *File* menu and the *Open...* menu item to open the DmEduc.mm file located in the mm directory of the DmEduc-00 project. The mm file extension stands for ModelibraModeler.

The main window of ModelibraModeler contains now a table of diagrams with only one WebLink diagram. Click on the *Diagram...* button to see the model. There are three icons in the tool bar bellow the menu bar: select a box or a line (a hand icon), create a box (a box icon), create a line (a line icon). For Modelibra, a box is a concept and a line is a relationship. There is only one concept in the model represented graphically as a box with three sections. The upper section shows the concept name. The middle section lists the concept's oid and other properties. The lower section is used to add new properties.

The Url concept may be selected (the black color border) by using first the hand icon in the tool bar and then by clicking on the concept title area. A concept may be deselected by clicking on an empty space in the diagram.

If you want to create a new model, use the *File/New* menu item and click on the *Add* button. Click on the *Diagram...* button to see the empty model in the new diagram window. Click first on the box icon in the tool bar to indicate that you want to create a new concept. Then click on an empty space in the diagram to create the concept. The concept name is presented in the upper area of the box. This area is used for a box title and also to move the box by keeping the mouse pressed in the title area. The new concept name is presented by default as the question mark. Double-click on the question mark to select it and change it to the concept name. Do not forget to use the *Enter* key on your keyboard so that the concept name is accepted by ModelibraModeler. If there is already another concept with the same name, the question mark will appear. By default, each model concept has the object identifier created automatically. Add concept properties by using the lower area of the box.

The box middle area is divided into multiple rows and two columns . For each new property there is a new row. The right column shows the property name (in Modelibra it is called a property code). The left column indicates if the property may be null (the o sign for null or optional). The oid property cannot be null. The oid property has the oid sign in the left column. For some other purposes, the oid name can be changed to something else, but the oid sign stays the same. By default, a new property is null. In our model, the link property cannot be null. To make a change, pop up the box menu by clicking with the right mouse button on the title section. Select *Items...* and the *Items* window will appear. In Modelibra, a box item is a concept property. Select the link property and change the minimal cardinality from 0 to 1, first by double-clicking on the value, then typing the new value over the old one, and finally by using the *Enter* key to accept the change. By default, the oid property has already the Oid type. A new property has, by default, the String type. Close the window and you will see that

the link property is not null any more .

The Url concept has only three properties, the oid property that all concepts in Modelibra have, and two properties specific to our modeling domain. The default box size is larger than what we need for only three properties. Move the mouse cursor to the box border. When the cursor changes press the the mouse left button and move the mouse in a direction that will allow you to reduce the box size.

The last thing to do is to declare the Url concept as an entry concept. When a model has only one concept, the concept must be entry into the model, since there are no other concepts that could be entry points. With the box pop-up menu check the *Entry?* menu item. In front of the concept name, the entry || sign appears. Another way to promote a concept into an entry point is to use the *Dictionary* menu and the *Boxes...* menu item. In the case of an error in modeling you can use the *Edit/Undo* menu item to come back to a previous diagram state.

With new versions of ModelibraModeler, you might loose a possibility to open old models. In ModelibraModeler a model is saved as a serializable file. If for any reason you cannot open the model file, use the *Transfer* menu in the application main window first to import data types (*Import Types...*), then to import model diagrams (*Import Diagrams...*). In this spiral, the files to import are called: DmEduc.type and DmEduc.diagram. It is a sound practice to also export types and diagrams so that you can recuperate the old models.

You may have more than one model in the same ModelibraModeler file (i.e., in the same domain). Use the *Add* button to add a new model. If you develop a model using the spiral approach, you might keep spiral models in the same file. Use the *Copy* button to copy an existing model into a new diagram, so that you can use the previous spiral model as the starting point for the new spiral model. If you have more than one model in a domain, when you generate an XML configuration for Modelibra from the main window of ModelibraModeler, configurations of all models of the same domain will appear in the same configuration file. In the next section, the domain configuration is done, for pedagogical reasons, by hand. It is shorter than what you can generate from ModelibraModeler by using the *Generation* menu in the diagram window.

Domain Configuration

The domain configuration is done in the specific-domain-config.xml file, which is located in the config directory of the Eclipse project. XML uses DTD [DTD], which stands for Document Type Definition to describe valid elements. The DOCTYPE declaration refers to the domain-config.dtd file in the config directory. In a Linux version the config/domain-config.dtd local path is not recognized and the domain-config.dtd local path must be used. In order to enable any beginner to execute the code in spirals without an error pointing to a non existing file, the DOCTYPE declaration is omitted from the code. You can always put this declaration back to your version of the code. There is also a version of the DOCTYPE declaration that refers to a DTD file located somewhere on the Web.

After the header comment, which is within <!-- -->, the domains root element indicates that it is possible to define several domains. However, in this book, only one domain will be used. The domain element has the oid attribute. In this example, the values of oid attributes have not been generated. The only restriction is that they should be numbers unique within this configuration. The domain name (or a code) is given within the code element tags. In more complex situations, there may be several

domains. In order to identify each one of them, the domain has its type. Together, the domain name and the domain type must be unique within all domains used. The domain may have several models. In this example, there is only one model with the WebLink code. In this spiral, the model is not persistent. There is only one concept in the model, although a model may have many concepts. The concept code is Url and it is an entry point into the model.

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

<!DOCTYPE models SYSTEM "config/domain-config.dtd">

      <!-- ======================================== -->
      <!-- Specific Model Config                    -->
      <!-- ======================================== -->

<domains>
    <domain oid="1101">
         <code>DmEduc</code>
           <type>Specific</type>

         <models>

               <model oid="110110">
                     <code>WebLink</code>
                     <persistent>false</persistent>

                     <concepts>

                          <concept oid="110110110">
                                <code>Url</code>
                                <entry>true</entry>

                                <properties>
                                     <property oid="110110110110">
                                          <code>link</code>
                                          <propertyClass>
                                                java.lang.String
                                          </propertyClass>
                                     <required>true</required>
                                     </property>
                                     <property oid="110110110120">
                                          <code>description</code>
                                          <propertyClass>
                                                java.lang.String
                                          </propertyClass>
                                     </property>
                                </properties>

                           </concept>

                     </concepts>

               </model>
```

```
        </models>

    </domain>

</domains>
```

There are only two specific properties in the Url concept. The predefined oid property from ModelibraModeler is not declared in the domain configuration, since it is known to Modelibra and it is not configurable. The two declared properties are link and description. Both properties are of the java.lang.String class. Note that in Modelibra. property types are Java classes that have the complete name in their declarations.

Domain Classes

There are five classes that cover the domain, the domain model and the model concept. There is one class for the domain configuration and one class for the domain. There is one class for the model, and two classes for the Url concept (entity and entities). In reality, even those classes may be generated from the domain XML configuration file by Modelibra. For pedagogical reasons, the code generation will start in one of subsequent spirals.

The first class to write is the DmEducConfig class that is almost the same for every new project. The only difference is in the domain name. Since DmEduc spirals for this book have the same domain name, the DmEducConfig class identical for all spirals.

The DmEducConfig class extends the Config class of Modelibra. The Config class, together with other configuration classes of Modelibra, converts an XML configuration into configuration objects that Modelibra consults quite frequently. The configuration objects represent the meta model of Modelibra. The root object of the meta model is obtained by the public getDomainConfig() method. From the configuration root objects it is possible to get other configurations. Both the Config and the DomainConfig classes can be found in the org.modelibra.config package of Modelibra.

```
package dmeduc;

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

public class DmEducConfig extends Config {

    private DomainConfig domainConfig;

    public DmEducConfig() {
        super();
        domainConfig = getDomainConfig("DmEduc", "Specific");
    }

    public DomainConfig getDomainConfig() {
        return domainConfig;
    }
```

```
}
```

The DmEduc class extends the Domain class from Modelibra. The constructor of the DmEduc class requires an object of the DomainConfig class. The DmEduc class represents the DmEduc domain that has one model called WebLink. The model is created in the constructor of the class. The model is referenced by the public getWebLink() method.

```
package dmeduc;

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

import dmeduc.weblink.WebLink;

public class DmEduc extends Domain {

	private WebLink webLink;

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

	public WebLink getWebLink() {
		return webLink;
	}

}
```

Model Classes

The WebLink class extends the DomainModel class from Modelibra. Generic behavior for handling any model is found in the DomainModel class. An object of the IDomain type is required for the construction of the model. IDomain is a Java interface (I stands for Interface) The WebLink class represents the WebLink model that has only one entry concept, the Urls entities, which are created in the constructor of the class. The entities are obtained by the public getUrls() method.

```
package dmeduc.weblink;

import org.modelibra.IDomain;
import org.modelibra.DomainModel;

import dmeduc.weblink.url.Urls;

public class WebLink extends DomainModel {

	private Urls urls;

	public WebLink(IDomain domain) {
```

```
            super(domain);
            urls = new Urls(this);
            setInitialized(true);
      }

      public Urls getUrls() {
            return urls;
      }

}
```

Since in this spiral the model is not persistent, the model is set, within its constructor, as initialized. In a persistent model, the model is initialized by Modelibra, but only if it is loaded properly.

Concept Classes

The Url class, from the dmeduc.weblink.url package, defines the Url concept as the concept entity. It extends (inherits from) the Entity abstract class. Generic behavior for handling any entity is found in that abstract class. An abstract class is used for the definition and not for the construction of objects. The Entity class is a generic type that is parametrized with a specific entity class as <Url>.

```
package dmeduc.weblink.url;

import org.modelibra.Entity;
import org.modelibra.IDomainModel;

public class Url extends Entity<Url> {

      private String link;

      private String description;

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

      public void setLink(String link) {
            this.link = link;
      }

      public String getLink() {
            return link;
      }

      public void setDescription(String description) {
            this.description = description;
      }

      public String getDescription() {
            return description;
```

```
	}

}
```

The Url class has a constructor with the IDomainModel parameter to indicate the domain model that an entity belongs to. IDomainModel is a Java interface. In contrast with a class, an interface does not have any method implementations. An interface defines a common behavior with the signatures of public methods.

The Url class has two private properties of the String class: link and description and the corresponding public set and get methods. The get method returns the property object. Thus, the return class of the get method is the same as the property class. The set method assigns a new object to the property using the method argument that has the same class as the property. Since the property set method does not return an object, the void keyword is used before the method name.

It is a good practice to make a property private and its access methods public. The only way to read a property outside the property class is through its get method. Outside the class, to create a property value or to modify it, the set method must be used. In this way the access to the property is controlled. Even for a stricter control, one of the access methods may be private (or non existent).

The Url class is an example of the POJO acronym [POJO] used often by practitioners of object oriented software to indicate that a design consists of simple classes.

The Urls class, from the dmeduc.weblink.url package represents a collection of entities. It extends (inherits from) the Entities abstract class. Generic behavior for handling a collection of entities is found in that abstract class. The Entities class is a generic type that is parametrized with a specific entity class as <Url>.

```
package dmeduc.weblink.url;

import org.modelibra.Entities;
import org.modelibra.IDomainModel;

public class Urls extends Entities<Url> {

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

	public Url createUrl(String link) {
		Url url = new Url(getModel());
		url.setLink(link);
		add(url);
		return url;
	}

}
```

The Urls class has a constructor with the IDomainModel parameter. A convenience method for creating a new Url object is provided. This is the first example of how Modelibra is used in the application

programming. The url object is constructed by passing the model of the current object of the Urls class. Then, the required link property is set. Finally, the new object is added to the entities. The add method is inherited from the Entities class.

JUnit Tests

JUnit is a popular testing framework [JUnit]. Tests are created in the test directory of the project. The *Project/Properties/Java Build Path/Source* path indicates that both src and test are source code directories. There are two test classes in the test directory: DmEducTest and UrlsTest. The DmEducTest class from the dmeduc package serves as the starting point for the real tests done in the UrlsTest class from the dmeduc.weblink.url package. There is a parallel between packages in the two source directories.

In the DmEducTest class, the Singleton design pattern [Singleton] is used to create one and only one domain test object. Singleton was one of the first design patterns used [Design Patterns]. The class constructor is private, so the only way to create a test object is by the public ans static getSingleton method. The static keyword is used to indicate the class level status of the method [static]. A method is static if there is no need to create an object in order to use the method. The private constructor is called only once. After that, the static dmEducTest property is initialized with the singleton object that is returned every time the getSingleton method is called again. The static keyword is used to indicate the class level status of the property [static]. The private open method is called by the constructor. It creates the domain object from the configuration. This domain object can be accessed publicly from outside this test class by the public getDmEduc method after the singleton test object is obtained:

```
	DmEduc dmEduc = DmEducTest.getSingleton().getDmEduc();
```

```
package dmeduc;

import org.modelibra.config.DomainConfig;

public class DmEducTest {

	private static DmEducTest dmEducTest;

	private DmEduc dmEduc;

	private DmEducTest() {
		super();
		open();
	}

	public static DmEducTest getSingleton() {
		if (dmEducTest == null) {
			dmEducTest = new DmEducTest();
		}
		return dmEducTest;
	}

	private void open() {
```

```
            DmEducConfig dmEducConfig = new DmEducConfig();
            DomainConfig domainConfig = dmEducConfig.getDomainConfig();
            dmEduc = new DmEduc(domainConfig);
      }

      public DmEduc getDmEduc() {
            return dmEduc;
      }

}
```

The open method creates the domain.

The real tests are done in the UrlsTest class. In this spiral there is only one test that initializes the model with two urls.

```
package dmeduc.weblink.url;

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

import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;

import dmeduc.DmEducTest;

public class UrlsTest {

      private static Urls urls;

      @BeforeClass
      public static void beforeTests() throws Exception {
            urls = DmEducTest.getSingleton().getDmEduc().getWebLink().getUrls();
      }

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

      @Test
      public void urlsCreated() throws Exception {
            Url url01 = urls.createUrl("http://www.modelibra.org/");

            assertNotNull(url01);
            assertTrue(urls.contain(url01));
            assertTrue(urls.getErrors().isEmpty());

            Url url02 = urls.createUrl("http://drdb.fsa.ulaval.ca/");

            assertNotNull(url02);
```

```
            assertTrue(urls.contain(url02));
            assertTrue(urls.getErrors().isEmpty());
        }

}
```

Classes that start with the org.junit prefix are imported from the JUnit jar library. There are also three static imports from the org.junit.Assert class. Java annotations [Annotations] are used to indicate to JUnit what to do before all tests are executed (@BeforeClass), before each test is executed (@Before) and what a test is (@Test).

In general, there may be more than one test in the same test class. Before all tests are executed, the beforeTests static method is invoked by JUnit. For JUnit, the method name is not important, only the @BeforeClass annotation must be placed in front of the method. The method task is to get the static urls property that will be used in tests. First, the singleton object of the DmEducTest class is obtained. Second, from the singleton object the domain is found by the getDmEduc method. Third, from the domain, the model is obtained. Finally, from the model the entry entities are reached by the getUrls method.

Before a single test is executed, the beforeTest method is called by JUnit. For JUnit, the method name is not important, only the @Before annotation must be placed in front of the method. Before a test is executed, the collection of errors is emptied. Different JUnit assert methods are used to assert desired test conditions. For example, the assertNotNull method asserts that object used as the parameter of the method will not be null. In Eclipse, the test class with JUnit assertions is selected and all tests are executed by the *Run As/JUnit Test* item in the pop-up menu. For each assertion that is not satisfied there is a corresponding failure in the JUnit tag of the Eclipse main window. All failures must be corrected in order to pass tests. In addition to a failure, an error may occur, which means that there is a major problem with the code.

Modelibra Interfaces

In Modelibra, the Entity class implements the IEntity interface and the Entities class implements the IEntities interface. Since the Url class extends the Entity class and the Urls class extends the Entities class, the Url class inherits all methods declared in the IEntity interface, and the Urls class inherits all methods declared in the IEntities interface. In contrast with the class concept, the interface concept does not implement any methods.

Not all methods in the two interfaces are obvious. Many of them will be gradually introduced in the following spirals. In this chapter only the following methods have been used.

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

    public IDomainModel getModel();

    public boolean add(T entity);

    public Errors getErrors();
```

```
}
```

An *interface* in Java is a protocol that classes, which accept that protocol, must implement. It is declared with the interface keyword and may contain only public method signatures and constant declarations. An interface is an abstract definition and as such it cannot be instantiated. An object may be declared to be of an interface type. In that case, the object must be either null or it must be an instance of a class that implements the interface. The keyword implements is used to indicate that a class implements the interface. In other word, the class implements the methods of the interface. A class which implements an interface must either implement all methods of the interface, or be an abstract class. As interfaces, abstract classes cannot be instantiated. Abstract classes are useful in that they can be used to define a protocol in more details at the implementation level. All classes in Java, other than the root class, must extend, explicitly or implicitly, one base class. However, a class may implement several interfaces.

The Entities abstract class implements the IEntities interface:

```
public abstract class Entities<T extends IEntity> extends Observable implements
           IEntities<T> {
      ...
}
```

A *generic type* is defined using one or more *type variables* and has one or more methods that use a type variable as a placeholder for an argument or a return type. The IEntities is a generic type that uses the T generic variable to accept any specific class that extends the IEntity interface. Thus entities cannot be objects of other types, but only of the IEntity type. For example, Urls are entities of objects of the Url type.

```
public class Urls extends Entities<Url> {
      ...
}
```

The Urls class inherits the add method from the Entities class, which in turn inherits the same method from the IEntities interface. The argument of the add method must be an object of the IEntity type such as Url.

```
public class Url extends Entity<Url> {
      ...
}
```

Both Urls and Url classes are *parameterized types* since they declare specifically a parameter of their corresponding generic type.

Summary

A simple domain model with only one concept is presented as the main theme of this chapter. The corresponding spiral project may be checked out from the code repository at JavaForge. The structure of an Eclipse project is explained. A short introduction to ModelibraModeler is made. An XML configuration of the domain model is shown. Java classes for the domain, model and concept are

presented. Test classes are introduced to enable the use of the domain model and its only concept. Modelibra interfaces used in the code of this spiral are indicated.

In the next chapter a simple domain model will be made persistent. By default, the model data will be saved in an XML file. In addition, a domain model of a new project will be designed in several steps.

Questions

1. What is the difference between a graphical model and its XML configuration?

2. Why is the XML configuration validation important?

3. Why there are two Java classes for a concept of the domain model?

4. How is the test class executed?

5. What is the difference between an interface and an abstract class in Java?

6. What is the difference between an abstract class and a (regular) class in Java?

Exercises

Exercise 2.1.

Add more valid web links as a new test in the UrlsTest class.

Exercise 2.2.

Try to add a not valid web link as a new test in the UrlsTest class.

Exercise 2.3.

Try to remove an existing web link as a new test in the UrlsTest class.

Web Links

[Annotations] Java Annotations
http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html

[Assembla] Assembla
http://www.assembla.com/

[beanutils] Commons beanutils
http://commons.apache.org/beanutils/

[Design Patterns] The Design Patterns: Java Companion
http://www.patterndepot.com/put/8/JavaPatterns.htm

[dom4j] dom4j
http://www.dom4j.org/

[DTD] Document Type Definition
http://www.comptechdoc.org/independent/web/dtd/

[Generics] Generic Types
http://www.onjava.com/pub/a/onjava/excerpt/javaian5_chap04/index.html
http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf

[Interfaces] Smarter Java development
http://www.javaworld.com/javaworld/jw-08-1999/jw-08-interfaces.html

[Java] Java
http://java.sun.com/

[JavaMail] JavaMail
http://java.sun.com/products/javamail/

[JRE] Java Runtime Environment
http://www.java.com/en/download/index.jsp

[JUnit] Junit Testing Framework
http://www.junit.org/

[JWS] Java Web Start
http://java.sun.com/products/javawebstart/

[log4j] Appache log4j
http://en.wikipedia.org/wiki/Log4j
http://logging.apache.org/log4j/

[logging] Commons logging
http://commons.apache.org/logging/

[Observable] Observer and Observable
http://www.javaworld.com/javaworld/jw-10-1996/jw-10-howto.html

[POJO] Plain Old Java Objects
http://en.wikipedia.org/wiki/POJO

[Properties] Properties
http://mindprod.com/jgloss/properties.html

[Singleton] Singleton Pattern
http://www.javacoffeebreak.com/articles/designpatterns/index.html

[static] Static fields and methods
http://mindprod.com/jgloss/static.html

[UTF] UTF-8
http://www.utf-8.com/

[XML] XML Tutorial
http://www.w3schools.com/xml/default.asp