

SECTION I: Domain Model Concepts

Chapter 1: Modelibra

This book is for Java programmers who want to learn how to develop a web application based on a domain model. A web application is developed by using two open source frameworks: Modelibra, a domain model framework, and Wicket, a web framework.

The objective of this chapter is to introduce Modelibra. Since Modelibra is a domain model framework, a simple domain model is first presented as a graph of concepts and relationships. The graphical representation is used to design a model. Once the model is designed, it is represented in Java as an XML configuration. This configuration is then used to generate Java classes that correspond to the model.

The use of spiral approach to learning new technologies and developing software is motivated. The importance of open source software for students is underlined. The rationale for the use of frameworks in software development is presented.

Spiral Approach

The spiral approach to software development [Spiral Model], which preserves a project history as a series of code snapshots, is used in this book. A simple web application, which will be used as a case throughout the book, is developed in multiple spirals. For each spiral, there is a complete software application and a chapter explaining the spiral. The web application is about different categories of web links that are of interest to certain members. Categories, web links, members and member interests are concepts of the application domain model. The application is dynamic because the content of its web pages changes with the content of the domain model.

Learning new software concepts and technologies is a challenging task. Learning in spirals, from simple to more advanced concepts but with concrete software applications, helps students get a reasonable confidence level early on [Spiral Education], and motivates them to learn by providing more useful applications. With each new spiral, the project grows and new concepts are introduced. A new spiral is explained with respect to the previous one. The difference between two consecutive spirals is that the next spiral has the new code introduced and the old code modified or deleted. This is called learning by anchoring to what we already understand. With a new spiral, we can come back to what we did previously and improve it. In this way, learning in spirals can touch the same topic several times, but each time with more details in a better version.

There are many books where students have to learn quite a lot before applying new concepts, and even then, it is not obvious how to develop a complete software application. It took me more than ten years of learning and teaching to find out that the initial learning of a new technology must be task driven and not topic (subject) driven. Most software books are topic driven. It takes a quick look at the Table of Contents of almost any software book to realize that each chapter introduces a major topic. In a spiral, there may be more than one topic and all of them relate to what we want to accomplish with the spiral.

Students like the spiral approach to learning software. Early spirals are easy to digest and they prepare them for more complex functions later on. Experienced students like the spiral approach as well, since it allows them to be selective in their learning. By following a subset of spirals, or parts of a single spiral, they can reach their goal faster in a more productive way.

I have used a similar spiral approach with Silverrun CASE (Computer Aided Software Engineering) tools [Silverrun], first to introduce Java to the development team, and second to develop commercial products in the role of Director of Research and Development. The spiral approach allowed me to focus on Java essentials in a more controlled way.

Open Source Software

Open Source Software (OSS) [OSS] is a software for which the programming code is available on the Web so that developers can modify and redistribute it. This contrasts with most commercial software, for which the source code is a closely guarded trade secret. OSS has a license agreement for its use that may cover rights to modify and redistribute it, even for commercial purposes. Thus, OSS does not necessarily mean "free" software. The basic idea behind OSS is simple. When developers on the Web can read and modify freely the source code, the software evolves. Programmers improve it and this can happen at a speed that, if one is used to the slow pace of conventional software development, seems astonishing.

For students, OSS is important for multiple reasons. Most of it is legally free. They can create an exciting development environment on their personal computers. There are many OSS projects that can be an incredible source of learning. A student can open a software and see how programmers have organized it, in a similar way that a student of literature learns the most by reading classics. An advanced student may even join an OSS project. OSS is the best way for a young software developer to get an international recognition. For companies, organizations, governments and students in developing countries, OSS provides a way to prevent the widespread illegal copying of software, and an opportunity to raise the level of software development to international standards.

Wicket

A dynamic web application is seen in a web browser through web pages that are created dynamically on a server. A request from a browser's user is sent to a server, where it is processed, usually with some data retrieved and embedded dynamically in a web page that is returned as a response to the user request. For most companies and organizations their own web site is strategically important. The more it is important, the more it becomes dynamic. The subject of developing dynamic web applications is exciting for students, since they can easily show on the Web what they are capable of doing. In addition, web applications have been popular both in business and education. Unfortunately for students, the subject is considered to be an advanced topic in both computer science and information systems. By using the spiral approach, students can be exposed to this important subject even at the beginning of their studies.

The software architecture of a dynamic web application is most often defined using the Model View Controller (MVC) design pattern [MVC]. A design pattern is a named (for example, MVC) description

of a professional solution (a professional web application is most often organized using MVC) to a recurring problem (most organizations face the problem of increasing complexity of web applications) that arises within a certain context (organizational data, business rules and web pages must be combined in a useful web application). In MVC, M communicates with data, C controls the interaction between a user request and an application response, and V produces dynamically web pages as application responses. All these activities happen on a server side, only a user request comes from a client, and a prepared web page is sent to a client by an application server.

It takes time to develop a dynamic web application. If we had a software that already knows how to organize a web application in MVC, we could focus on application data, on data actions, and on making web pages appealing to users. Fortunately, such a software exists in different forms. The most popular category of software that improves the productivity of software developers is called framework [Frameworks]. A framework is an integrated set of components that collaborate to produce a reusable architecture for a family of applications. More specifically, a framework is a predefined architecture in a vertical domain (such as web applications), consisting of both generic and specific building blocks. Areas that are specific must be refined by application developers by means of extending existing framework objects to provide application specific features.

Recently we have seen the proliferation of web frameworks whose main objective is to simplify the development of complex web applications. There are many web frameworks [OSS Web] developed in Java. The most popular is Struts [Struts] from the Apache Jakarta OSS Project. Struts relies more on external configuration files and less on Java code to speed up the web application development. Struts uses one or more XML configuration files to specify the flow of a web application. It is an action based framework. A user action is the focus of attention in Struts. As a consequence, the control part of Struts is rather elaborate for developers. Learning Struts and developing web applications with Struts are difficult tasks. There are many books on this subject and most of them are only suitable for advanced students.

I have decided to use a new web component based framework called Wicket [Wicket], to show how easy is to develop a dynamic web application with Wicket once there is a well designed domain model. Wicket, brings plain object oriented programming to web applications. It is a web application framework for creating dynamic web pages by using web components. It uses only two technologies: Java and HTML. Wicket pages can be designed by a visual HTML editor. Dynamic content processing and form handling is all handled in Java code. A web component, such as a web page or a page section, is in the center of preoccupations in Wicket. The control part of Wicket is largely hidden from developers.

Wicket provides a generic support for web applications. Web application concepts have corresponding classes in the Wicket framework and in the web application. For example, for the web application concept, Wicket has the `WebApplication` class. For a web page, Wicket has the `WebPage` class. Each web page in the web application has its own class that extends `WebPage`.

Although Wicket is used in this book to develop a small web application, the framework is designed to support very large web applications. By default, Wicket manages the state of a web application by memorizing states of web pages and page components. For small applications there is no need to write some special state management code. However, Wicket allows developers to take control of the state management.

I have decided to develop a new domain model framework, called Modelibra [Modelibra], to provide

an easy support for (application) domain models. The framework uses the 80/20 rule where 80% of the model classes are generic (part of Modelibra) and 20% are specific.

Modelibra

Modelibra is a domain model framework. It is an open source software. Its objective is to provide an easy to learn and easy to use framework. Its recommended use is in prototypes and small applications. It may be used in client and server applications. It has been already used in several web applications. I have also decided to develop in spirals a dynamic web application with Modelibra and Wicket to allow Java programmers to learn gradually the basics of web development based on a domain model. This book is the result of that effort.

In computer terms, a domain model is a model of specific domain classes that describe the core data and their behavior [Domain Model]. From an organizational perspective, a domain model is a model of the domain. Within the domain, an organization conducts its business [Domain Modeling]. The memory of any organization may be abstracted as a domain model. The backbone of any software is a domain model. When a model is well designed and when it can be easily represented and managed in an object oriented language, a developer may focus on views of the software and they are what users care about the most.

In a small application, a domain model is the core part of the application software. Modelibra has been designed to help developers of small applications in representing and using application domain models in a restricted way. The Modelibra restrictions minimize the number of decisions that a domain designer must make. This makes Modelibra easy to learn. Since Modelibra is an OSS, developers of small OSS will find Modelibra useful for developing their software around a domain model, for providing an easy installation for their users and for developing a web application to introduce their software to the public audience.

In small OSS projects, often, there is only one developer or a small team of programmers. Similarly, in software engineering education, students work on individual assignments or participate in small teams. In both contexts, Modelibra helps developers focus their efforts on their task at hand. Modelibra has a companion web framework called ModelibraWicket. ModelibraWicket consists of generic web components that make a default Wicket application out of a domain model. This default application or a prototype may help developers validate and consequently refine the domain model. In addition, the generic web components may be easily reused in a specific web application. Their reuse consists of collecting data from a domain model by using Modelibra and by supplying the data as parameters to the web components. The generic web components from ModelibraWicket use Modelibra for a domain model and Wicket for application views.

If a generic web component does not satisfy the needs of a developer, a specific web component may be developed by using Modelibra and Wicket. The difference between a generic web component and a specific web component is that a generic web component already exists in ModelibraWicket, while a specific web component must be developed. If variations of the same specific web component are used in the same or different web applications, a more experienced software engineer may create a new generic web component. In this way, a catalog of web components in ModelibraWicket would grow and the interest to Modelibra and Wicket would grow as well.

In most cases, data of a domain model must be saved in an external memory. If a database system is used for that purpose, an OSS requires at least several complex steps in order to install the software. Modelibra uses XML files to save data of the current domain model. There is no need for any special software installation. However, Modelibra allows the use of both relational and object databases. The upgrade of an application from XML data files to a database does not require a single line of programming code to be changed. It is enough to change a domain configuration in XML, and, of course, create a database and configure it to run the same application again. Modelibra also provides the data migration from XML files to a database.

Even for large OSS such as web frameworks, it is useful to have Modelibra around for creating quickly software demos that are easy to install, so that the learning process may start quickly with the focus on the software at hand. Similar arguments may be used for developing prototypes.

Although the focus of Modelibra is a small software application (or rather a software application with a small amount of data), the framework provides some advanced features such as transactions and undo, required in a professional software.

A domain may have several models. One of them may be a reference model where common data, common to all models within a domain, are kept. A domain model may also inherit some of its definitions from another model in the same domain.

In Modelibra, a part of the base model may be exported to another model, which can be taken for an off-line work, then returned back to synchronize changes with the base model.

The closest software to Modelibra is the Eclipse Modeling Framework (EMF). EMF is a Java framework and code generation facility for building tools and other applications based on a structured model [EMF]. It is a professional framework that may not be easy to learn for developers of small OSS and students of software engineering or information systems.

Domain Model

A domain model is a representation of user concepts, concept properties and relationships between concepts. The easiest way to present a domain model is through a graphical representation. Figure 1. represents a domain model of our case application. It features web links that are of interest to certain members. The domain model is created by the ModelibraModeler design and code generation tool. The tool provides different colors for different model components. However, a model may be displayed in black and white only with a scale of different gray colors, as is the case in this book.

In our case, the domain model's concepts are: Url, Question, Category, Member, Interest and Comment. Url (Uniform Resource Locator [URL]) describes a web link. Urls are categorized. Categories are organized in a tree of subcategories. Question is a frequently asked question about the use of the web application. Questions are optionally categorized. Members express their interests in categories of web links. Comments can be made about anything related to the web application.

A concept is described by its properties and neighbors, called together the concept's attributes. The Url concept has a (summary) name, a (web) link, a (short) description, a creation date (creationDate), the last update date (updateDate) and whether it is approved or not. The description and updateDate

properties may be null (the o symbol in front of a property name). The Url concept has only one neighbor, the Category concept. However, the Category concept has four neighbors: Url, Question, Interest and Category concepts. A relationship between two concepts is represented by two neighbor directions, displayed together as a line. A neighbor direction is a concept special (neighbor) property, with a name and a range of cardinalities. The Category --> Url direction is named urls; its minimal cardinality is 0 and its maximal cardinality is N (many). In other words, a category may have from 0 to N urls. A url has exactly one category. The Question --> Category direction is named category; its minimal cardinality is 0 and its maximal cardinality is 1. Thus, a question may not be categorized. In summary, a concept is described by its properties and neighbors. A neighbor is either a child or a parent. A child neighbor has the maximum cardinality of N (or a number greater than 1). A parent neighbor has the maximum cardinality of 1.

A concept is represented as a list of entities. The retrieval of entities starts with the entry concepts of the domain model. In the WebLinks domain model, the entry concepts are Category, Question, Member and Comment. They all have a darker border and the || symbol in the upper left corner of the concept. Once an entity of the entry concept is retrieved in the list of entities, the retrieval of neighbor entities may start. A child neighbor is represented as a list of entities. A parent neighbor is represented as a single entity. The Url concept is not an entry concept. Hence, entities of the Url concept may be reached only through its parent Category concept. As a non-entry the Url concept has a lighter border. The Interest concept has two parents. Thus, interests may be retrieved either from the Member concept or the Category concept. A concept that has more than one parent is called an intersection concept. An intersection concept that represent a many-to-many relationship has the X sign in the upper right corner of the concept.

Every concept has a predefined property called oid. The oid property is mandatory. It is used as an artificial concept identifier (object id) and is completely managed by Modelibra. Its value is unique universally. In addition, a concept may have at most one user oriented (semantical) identifier (id) that consists of the concept's properties and/or neighbors. A simple id has only one property. In an entry concept, all entities must have a unique value for the concept id. However, in a non-entry child concept, the id is often unique only within the child parent. For example, the id of the Url concept consists of the name property and the category neighbor. Thus, a name must be unique only within its category.

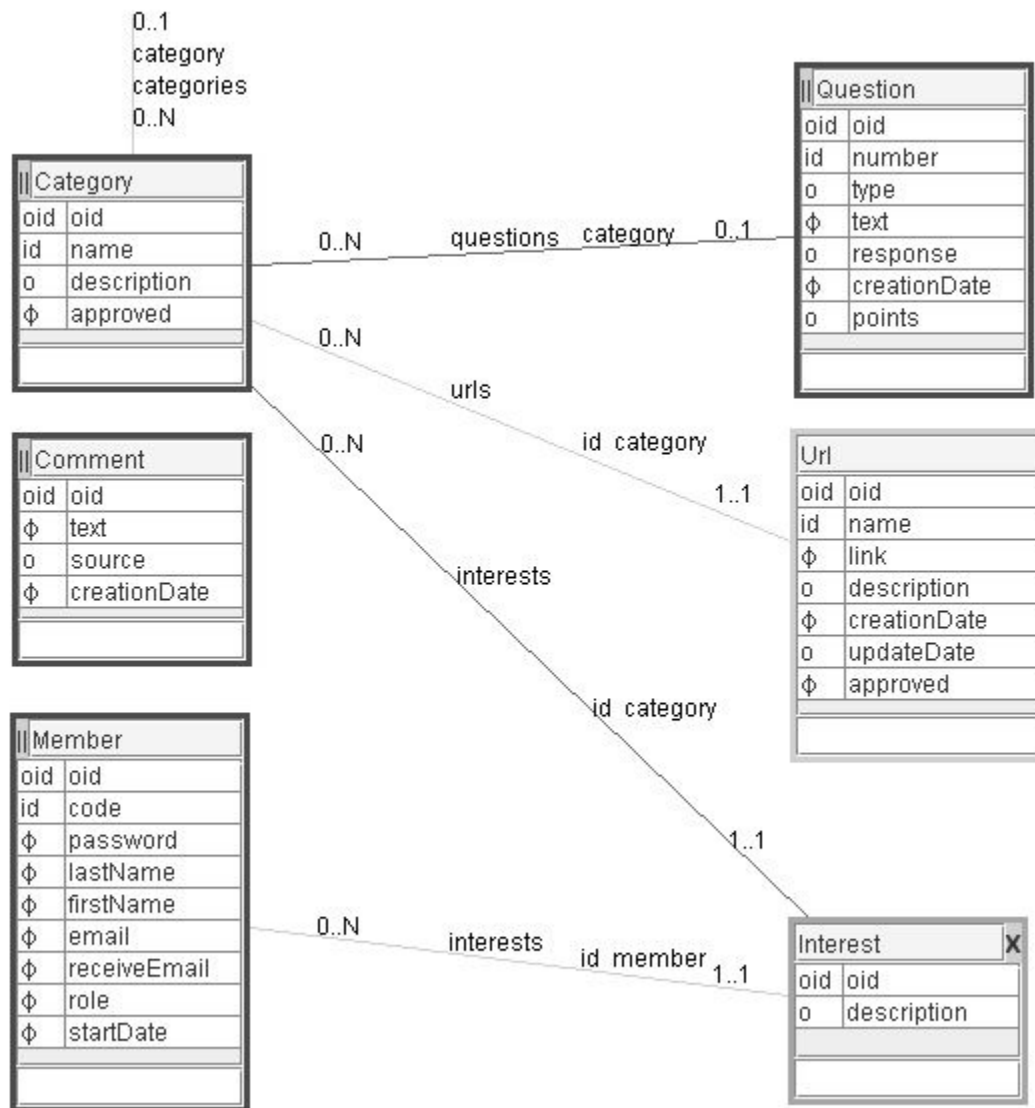


Figure 1.1. WebLinks domain model

Java Classes

A concept is represented in Modelibra as two POJO [POJO] classes. Both POJOs are specific to the domain. One extends the generic Entity class and the other extends the generic Entities class. Both generic classes are part of Modelibra and may be reused in different domains. The Entity class implements the IEntity interface, and the Entities class implements the IEntities interface.

For example, the Category concept has two specific classes: Category and Categories. The Category class describes the Category concept and the Categories class represents all categories. A specific category may have from 0 to N urls. The urls for that category, and other urls for other categories are represented by the Urls class. The Url class is used to describe the Url concept.

For the sake of space, only Category and Url concepts with their relationship will be considered in this example (as if the domain model had only those two concepts and only that relationship). The example

is used to show the flavor of Modelibra. Do not worry if you do not understand all details; they will be explained gradually in subsequent chapters. In addition, the basic content of the domain model's specific classes may be generated by Modelibra.

The Category class extends the abstract Entity class passing itself as a generic type parameter. It contains three properties and one child neighbor with the corresponding set and get methods. All properties are typed by Java classes and not by Java base types such as int or boolean. A Boolean property has also an is method that returns a boolean base value for convenience reasons. The class constructor passes the domain model to its inheritance parent. The internal neighbor is created in the class constructor. The neighbor set method sets the current object as the neighbor parent.

```
public class Category extends Entity<Category> {

    private String name;

    private String description;

    private Boolean approved;

    // Url child neighbor (internal)
    private Urls urls;

    public Category(IDomainModel domainModel) {
        super(domainModel);
        // internal child neighbors only
        urls = new Urls(this);
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

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

    public String getDescription() {
        return description;
    }

    public void setApproved(Boolean approved) {
        this.approved = approved;
    }

    public Boolean getApproved() {
        return approved;
    }
}
```

```

    public boolean isApproved() {
        return getApproved().booleanValue();
    }

    /*
     * Neighbors
     */

    public void setUrls(Urls urls) {
        this.urls = urls;
        urls.setCategory(this);
    }

    public Urls getUrls() {
        return urls;
    }
}

```

The Categories class extends the abstract Entities class passing itself as a generic type parameter. The class constructor passes the domain model to its inheritance parent. A category may be retrieved either by the oid or by a property. If a property is not unique, the first category whose property is equal to the property argument is retrieved. If there is no such a category, null is returned. A subset of categories may also be selected. This selection produces a new entities object.

```

public class Categories extends Entities<Category> {

    public Categories(IDomainModel domainModel) {
        super(domainModel);
    }

    public Category getCategory(Oid oid) {
        return retrieveByOid(oid);
    }

    public Category getCategory (Long oidUniqueNumber) {
        return getCategory (new Oid(oidUniqueNumber));
    }

    public Category getCategory(String propertyCode, Object property) {
        return retrieveByProperty(propertyCode, property);
    }

    public Categories getCategories(String propertyCode, Object property) {
        return (Categories) selectByProperty(propertyCode, property);
    }
}

```

The Url class is also a POJO. It contains six properties and one parent neighbor with the corresponding

set and get methods. Note that in Modelibra both relationship directions are represented as neighbors.

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

    private String name;

    private String link;

    private String description;

    private Date creationDate;

    private Date updateDate;

    private Boolean approved;

    // Category parent neighbor (internal)
    private Category category;

    public Url(IDomainModel domainModel) {
        super(domainModel);
        // no internal child neighbors
    }

    public Url(Category category) {
        this(category.getModel());
        // parent
        this.category = category;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    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;
    }
}
```

```

    }

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

    public Date getCreationDate() {
        return creationDate;
    }

    public void setUpdateDate(Date updateDate) {
        this.updateDate = updateDate;
    }

    public Date getUpdateDate() {
        return updateDate;
    }

    public void setApproved(Boolean approved) {
        this.approved = approved;
    }

    public Boolean getApproved() {
        return approved;
    }

    public boolean isApproved() {
        return getApproved().booleanValue();
    }

    /*
     * Neighbors
     */

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

    public Category getCategory () {
        return category;
    }

}

```

The `Urls` class has several selection methods. It also has the protected `post update` method that overrides the corresponding method in the `Entities` class. `Modelibra` has pre and post action methods to do either some specific validations or action propagations. In this example, the post action sets an update date. The `EasyDate` type is used to simplify date handling, since `Date` is a rather complex type in Java.

```

public class Urls extends Entities<Url> {

```

```

// Category parent neighbor (internal)
private Category category;

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

public Urls(Category category) {
    this(category.getModel());
    // parent
    this.category = category;
}

public Url getUrl(Oid oid) {
    return retrieveByOid(oid);
}

public Url getUrl(Long oidUniqueNumber) {
    return getUrl(new Oid(oidUniqueNumber));
}

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

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

@Override
protected boolean postUpdate(Url beforeUrl, Url afterUrl) {
    if (super.postUpdate(beforeUrl, afterUrl)) {
        Date today = new Date();
        Date updateDate = afterUrl.getUpdateDate();
        EasyDate easyToday = new EasyDate(today);
        EasyDate easyUpdateDate = new EasyDate(updateDate);
        if (!easyToday.equals(easyUpdateDate)) {
            Url afterAfterEntity = afterUrl.copy();
            afterAfterEntity.setUpdateDate(today);
            return update(afterUrl, afterAfterEntity);
        } else {
            return true;
        }
    }
    return false;
}

/*
 * Neighbors
 */

```

```

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

    public Category getSubject() {
        return category;
    }
}

```

Modelibra Interfaces

The easiest way to get a feeling about what Modelibra offers is to look at its main Java interfaces [Interface]. An interface is a group of related methods without the implementation code. What is left in a method without its implementation is called a method signature. The four main Modelibra interfaces are: IDomain, IDomainModel, IEntity and IEntities. In Modelibra, the name of an interface starts with I to distinguish it easily from a class that implements the interface and has the same name without I. Thus, there is also the Domain class that implements the IDomain interface. The interfaces in Modelibra are serializable.

The IDomain interface provides method signatures that indicate what can be done with a domain. A domain has a configuration and it can be obtained by the getDomainConfig method. A domain may have several models. One of those models may be a reference model, where concepts common to all models may be kept. A domain is closed by closing its models.

```

public interface IDomain extends Serializable {

    public DomainConfig getDomainConfig();

    public IDomainModel getModel(String modelCode);

    public IDomainModel getReferenceModel();

    public IDomainModels getModels();

    public void close();
}

```

A model is a part of its domain. It has a configuration. A model is empty if all its concepts are empty. A model may have several entries where the search for entities may start. A model may be exported as a taken model with or without sensitive data. This means that a replica of the base model is made. The content of the replica may be modified with additions and updates and later on returned and synchronized with the base model with the help of the taken model. Some of the content of the replica may be removed and the base model may be later on cleaned based on those removals. A model may have a session. When a model is closed its content is not available any more. If a model is persistent, a new model must be constructed to reach the persistent data.

```

public interface IDomainModel extends Serializable {

    public IDomain getDomain();

    public ModelConfig getModelConfig();

    public boolean isEmpty();

    public IEntities getEntry(String entryCode);

    public List<IEntities> getEntryList();

    public void export(IDomainModel takenModel, boolean exportSensitive);

    public void synchronize(IDomainModel takenModel, IDomainModel returnedModel,
        boolean synchronizeSensitive);

    public void clean(IDomainModel takenModel, IDomainModel returnedModel);

    public boolean isSession();

    public ModelSession getNewSession();

    public void close();

}

```

An entity belongs to its model. It has a configuration. An entity must have an artificial identifier called oid. An entity may have at most one user oriented identifier (id) that can be obtained by the getUniqueCombination method. If the index is configured to be used, by default, there is a unique index for oid and a unique index for the id combination of properties and/or neighbors. In addition a non-unique index combination may be configured. Properties and neighbors of an entity may be updated by the update method. Properties of an entity may be updated by the updateProperties method. An entity may be copied. Only properties of an entity may be copied. A deep copy of an entity may be made by copying the internal tree that starts with the entity as the root of the tree. An entity may be compared with another entity to verify if they are equal based on different criteria. There are set and get methods for properties, parent neighbors and child neighbors.

```

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

    public IDomainModel getModel();

    public ConceptConfig getConceptConfig();

    public void setOid(Oid oid);

    public Oid getOid();

    public UniqueCombination getUniqueCombination();

```

```

    public IndexCombination getIndexCombination();

    public boolean update(T entity);

    public boolean updateProperties(T entity);

    public T copy();

    public T copyProperties();

    public T deepCopy(IDomainModel model);

    public boolean equalOid(T entity);

    public boolean equalUnique(T entity);

    public boolean equalProperties(T entity);

    public boolean equalParentNeighbors(T entity);

    public boolean equalContent(T entity);

    public void setProperty(String propertyCode, Object property);

    public Object getProperty(String propertyCode);

    public void setParentNeighbor(String neighborCode, IEntity<?> neighborEntity);

    public IEntity<?> getParentNeighbor(String neighborCode);

    public void setChildNeighbor(String neighborCode,
                                IEntities<?> neighborEntities);

    public IEntities<?> getChildNeighbor(String neighborCode);
}

```

A collection of entities belongs to its model. Entities have a configuration. An iterator over entities may be defined. The number of entities determines its size. If the size is zero, entities are empty. An entity may be added to entities. An entity may be removed from entities. An entity that belongs to entities may be updated by using another entity. Properties of an entity may be updated by using another entity. It can be verified if entities contain an entity. An entity may be retrieved by its oid, by the unique combination, by the index, or by a property and its value. A subset of entities may be selected by a method that returns true when applied to an entity, by an index, by a property based on its value, by a parent neighbor, or by a selector for a more elaborate selection. Entities may be ordered by a property, or by a more elaborate comparator. A union or an intersection may be made out of two collections of entities. It can be verified if one collection of entities is a subset of another collection of entities. In Modelibra, a selection of entities or an order of entities produces a new collection of destination entities. The source entities are reachable from the destination entities. In Modelibra, it is preferable to work directly with entities. However, a Java list of entities may always be obtained. There are positional methods to find the first and last entities, to find the next and the prior entity based on the given entity,

to locate an entity based on its position. A random entity may also be returned. Entities have a corresponding collection of errors. A copy or a deep copy of entities may be made. Entities may be exported, synchronized and cleaned.

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

    public IDomainModel getModel();

    public ConceptConfig getConceptConfig();

    public Iterator<T> iterator();

    public int size();

    public boolean isEmpty();

    public boolean add(T entity);

    public boolean remove(T entity);

    public boolean update(T entity, T updateEntity);

    public boolean updateProperties(T entity, T updateEntity);

    public boolean contain(T entity);

    public T retrieveByOid(Oid oid);

    public T retrieveByUnique(UniqueCombination uniqueCombination);

    public T retrieveByIndex(IndexCombination indexCombination);

    public T retrieveByProperty(String propertyCode, Object property);

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

    public IEntities<T> selectByIndex(IndexCombination indexCombination);

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

    public IEntities<T> selectByParentNeighbor(String neighborCode,
        IEntity<?> neighbor);

    public IEntities<T> selectBySelector(ISelector selector);

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

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

    public IEntities<T> union(IEntities<T> entities);
```

```

    public IEntities<T> intersection(IEntities<T> entities);

    public boolean isSubsetOf(IEntities<T> entities);

    public IEntities<T> getSourceEntities();

    public List<T> getList();

    public T first();

    public T last();

    public T next(T entity);

    public T prior(T entity);

    public T locate(int position);

    public T random();

    public Errors getErrors();

    public IEntities<T> copy();

    public IEntities<T> deepCopy(IDomainModel model);

    public void export(IEntities<T> takenEntities, boolean exportSensitive);

    public void synchronize(IEntities<T> takenEntities,
        IEntities<T> returnedEntities, boolean synchronizeSensitive);

    public void clean(IEntities<T> takenEntities, IEntities<T> returnedEntities);
}

```

XML Configuration

A domain with its models must be configured in an XML configuration file. This configuration reflects the model's POJO classes. However, it provides more information about the model's default behavior used heavily in web components of ModelibraWicket. The XML configuration is loaded up-front and converted into meta entities. Those meta entities are consulted by Modelibra quite often. The following is a minimal version of the configuration. There are more elements that may even show a default behavior at the model's view level. The XML configuration may be generated from the domain model in ModelibraModeler.

The domain is called DmEduc. It is a specific configuration that may be changed by a developer. The domain has only one model with the WebLink name (code for Modelibra). For the sake of space, the model has only two concepts: Category and Url. The Category concept is an entry into the model.

Since the plural noun of the Category concept is irregular, its Categories name is indicated. Each property is declared by its name, class and whether it is required. There are also some additional elements for unique values, for default values, and validation types. Each neighbor is defined by its name, destination concept, whether it is a parent or a child, by minimal and maximal cardinalities, and whether it is a part of the unique combination.

```
<domains>
  <domain oid="1101">
    <code>DmEduc</code>
    <type>Specific</type>
    <models>
      <model oid="110110">
        <code>WebLink</code>
        <author>Dzenan Ridjanovic</author>
        <concepts>
          <concept oid="110110100">
            <code>Category</code>
            <entitiesCode>Categories</entitiesCode>
            <entry>true</entry>
            <properties>
              <property oid="110110100110">
                <code>name</code>
                <propertyClass>
                  java. lang. String
                </propertyClass>
                <required>true</required>
                <unique>true</unique>
              </property>
              <property oid="110110100120">
                <code>description</code>
                <propertyClass>
                  java. lang. String
                </propertyClass>
              </property>
              <property oid="110110100130">
                <code>approved</code>
                <propertyClass>
                  java. lang. Boolean
                </propertyClass>
                <required>true</required>
                <defaultValue>false</defaultValue>
              </property>
            </properties>
          </concept>
        </concepts>
      </model>
    </models>
  </domain>
</domains>
```

```

        <max>N</max>
    </neighbor>
</neighbors>
</concept>
<concept oid="110110110">
    <code>Url</code>
    <entry>false</entry>
    <properties>
        <property oid="110110110100">
            <code>name</code>
            <propertyClass>
                java. lang. String
            </propertyClass>
            <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>
            <required>true</required>
        </property>
        <property oid="110110110120">
            <code>description</code>
            <propertyClass>
                java. lang. String
            </propertyClass>
        </property>
        <property oid="110110110130">
            <code>creationDate</code>
            <propertyClass>
                java. util. Date
            </propertyClass>
            <required>true</required>
            <defaultValue>today</defaultValue>
        </property>
        <property oid="110110110140">
            <code>updateDate</code>
            <propertyClass>
                java. util. Date
            </propertyClass>
        </property>
        <property oid="110110110150">
            <code>approved</code>
            <propertyClass>
                java. lang. Boolean
            </propertyClass>
        </property>
    </properties>
</concept>

```

```

        <required>true</required>
        <defaultValue>>false</defaultValue>
    </property>
</properties>
<neighbors>
    <neighbor oid="110110110810">
        <code>category</code>
        <destinationConcept>
            Category
        </destinationConcept>
        <type>parent</type>
        <min>1</min>
        <max>1</max>
        <unique>true</unique>
    </neighbor>
</neighbors>
</concept>
</concepts>
</model>
</models>
</domain>
</domains>

```

XML Data

The following is an XML content of our simplified example that consists only of the Category and Url concepts. There are two categories: Frameworks and Personal with their corresponding urls. Note that both categories and urls are stored in the same XML file declared for the Categories entry. Entities of an internal relationship between two concepts are saved in the same XML data file, starting with the entry concept.

```

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

<categories>
    <category oid="1146767279506">
        <name>Frameworks</name>
        <description>Vertical domain productivity software.</description>
        <approved>true</approved>
        <urls>
            <url oid="1146767279553">
                <name>Modelibra</name>
                <link>http://drdb.fsa.ulaval.ca/modelibra/</link>
                <description>
                    Modelibra is a domain model framework.
                </description>
                <creationDate>2006-05-04</creationDate>
                <updateDate>2006-05-06</updateDate>
                <approved>true</approved>
            </url>
            <url oid="1146942566939">

```

```

        <name>Wicket</name>
        <link>http://wicket.sourceforge.net/</link>
        <description>
            Wicket is a web application framework.
        </description>
        <creationDate>2006-05-06</creationDate>
        <approved>true</approved>
    </url>
</urls>
</category>
<category oid="1146767279586">
    <name>Personal</name>
    <description>Personal web sites.</description>
    <approved>true</approved>
    <urls>
        <url oid="1146767279602">
            <name>Dzenan Ridjanovic</name>
            <link>http://drdb.fsa.ulaval.ca/</link>
            <description>
                Dzenan Ridjanovic's home page.
            </description>
            <creationDate>2006-05-04</creationDate>
            <approved>true</approved>
        </url>
    </urls>
</category>
</categories>

```

Persistent Model

A domain model is usually persistent. Model entities may be saved to an XML file or model changes may be registered in a relational or an object database. By default, entities are saved in XML files. At the beginning of its use, a domain model is completely loaded from an external memory. Hence, all model entities are available to application views. When the model is loaded, all model actions are done in the main memory of a computer. A software application programmer works with a domain model and does not have to write a single line of code to persist data. When specific entities, such as Categories change, by adding a new entity, by removing an entity, or by updating an entity, the entities are saved back to the corresponding XML data file by Modelibra.

An XML persistent domain model is loaded from one or more XML data files. There is one XML data file for each entry concept of the domain model. What is saved in an entry data file depends on the relationship type: internal or external. In general, a model is a network graph where each concept is a node and each relationship is a line. For the purpose of persistence, the network model is divided into hierarchical sub-models, where each entry concepts represents a root of a hierarchy of concepts and internal relationships. Starting with the entry concept, the internal neighbor directions of the child type determine a scope of the hierarchy. External relationships are used to connect hierarchical sub-models. An internal neighbor is saved within its parent, while an external neighbor is not. From the entry concept, its internal neighbor directions are followed to save the entry with its children hierarchy in the XML data file of the entry concept. External neighbor directions determine the entry hierarchy

boundary. For convenience reasons, the XML persistence is used in this book.

There are two external relationships in the WebLinks domain model, Category – Interest and Category – Question, and they are represented visually as lines darker than lines of internal relationships. In an external relationship, the child concept has a reference property that is used to recreate a normal (internal) relationship out of the external relationship when data are loaded into the main memory of a computer. Once all external relationships are converted to internal ones, the model becomes a regular network of entities. In the Category -- Interest external relationship, the Interest class has the categoryOid reference property. Since the Category – Interest relationship is external and the Member – Interest relationship is internal, interests are not saved within their category, but within the parent member. Similarly, questions are not saved within their category, since there are questions that may not have a category. The Question class has also the categoryOid reference property that is used to regroup questions under their category.

The recommended use of Modelibra is in prototypes and applications with the small amount of data. It may be used both in client and server applications. However, since its model data are all available at the same time, Modelibra may be used as a main memory database for those software applications that require a high availability of data and a high response time.

ModelibraWicket

A web application is a collection of web pages. A dynamic web application has some pages that are generated dynamically based on some data. When those data change, the content of the corresponding web pages changes as well. A web application is executed by a web server. A web server listens to user requests that come from web browsers and delegates those requests to the corresponding web application. A delegated user request is used by the web application as the starting point to generate a response web page that is then passed to the web server. The server will send the response page to the user that has requested it.

ModelibraWicket makes a domain model alive as a web application, so that model data may be displayed as web pages and updated through forms. The model is metamorphosed into a web application with the help of its XML configuration. This web application can be considered a default application of the domain. With some small changes in the XML configuration, the web application may be somewhat customized. It is important to realize that this web application is not a version that one would like to install as a web site. Its main purpose is to validate a domain model by designers and future users of the web application and consequently refine the domain model. In addition, ModelibraWicket has a collection of generic web components that may be easily reused in professional web applications to display or update entities.

The home page of the default web application provides a button to enter *Modelibra* (Figure 1.2). The *DmEduc* domain name appears in the the title section.

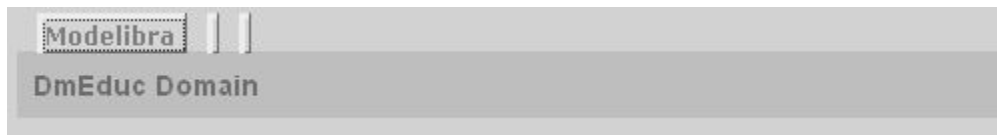


Figure 1.2. Home page

The *Modelibra* page is the application domain page with a table of domain models (Figure 1.3). The domain title is *Modelibra – Education Domain* and the only model in the domain is *WebLink*. The model has two buttons, one for displaying model entries and another for updating model entries. The *Home* button displays the home page of the application.

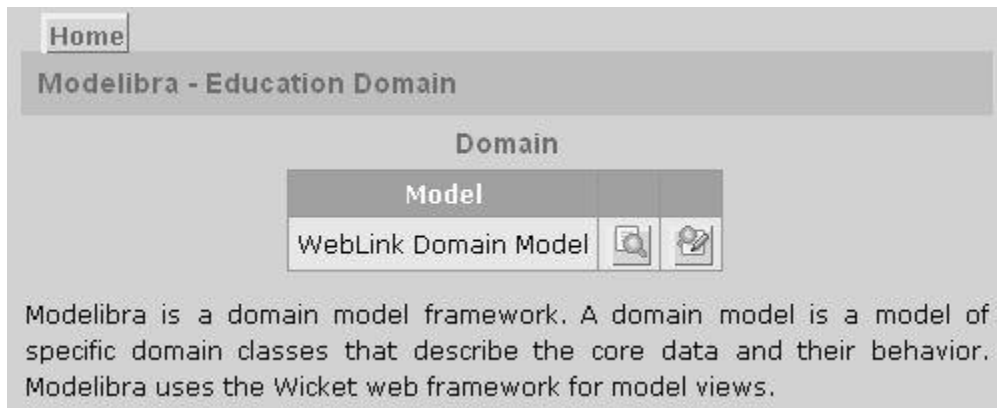


Figure 1.3. Domain page

The display model page presents a table of model entries (Figure 1.4). In order to present web pages of a simple model (without relationships) from one of early spirals, only two entry concepts are used: Comment and Url. For each concept there are display and select buttons. The display button presents the concept as a table of entities, with only required properties displayed. Each entity may be further displayed with all its details. The select button provides the keyword based selection.

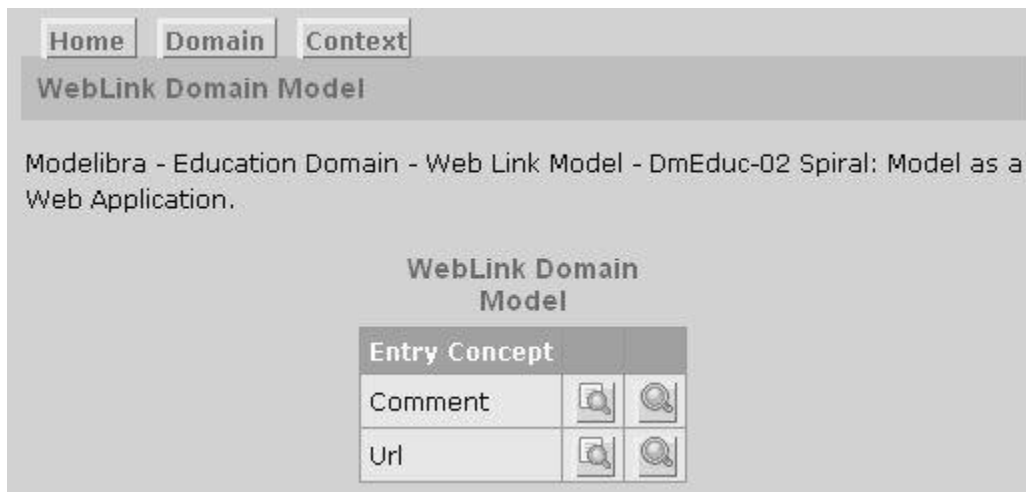


Figure 1.4. Entry concepts display

The update model page presents a table of model entries with update buttons (Figure 1.5).

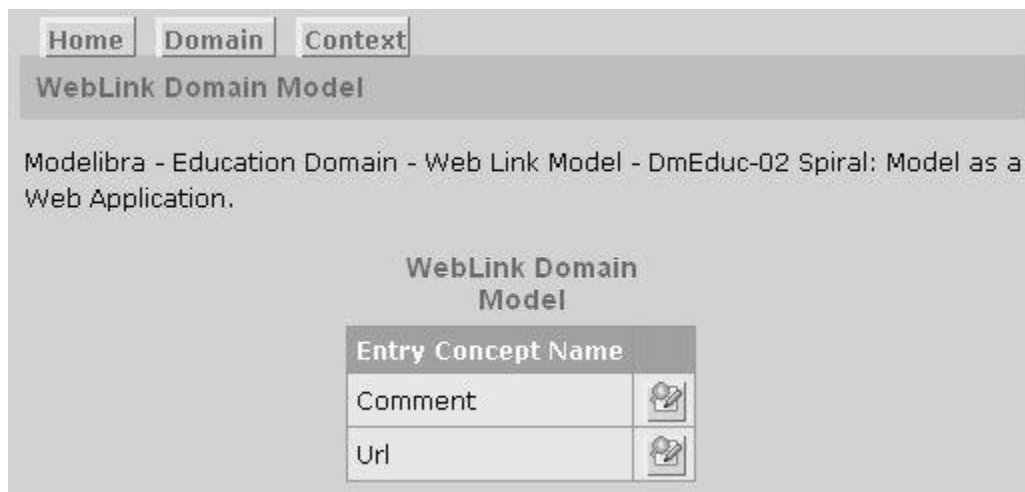


Figure 1.5. Entry concepts update

The entry concept has its own display or update page with a table of entities. The *Context* button in the upper left corner of a page plays a role of the back button. In Figure 1.6. comments are displayed with only two required properties.

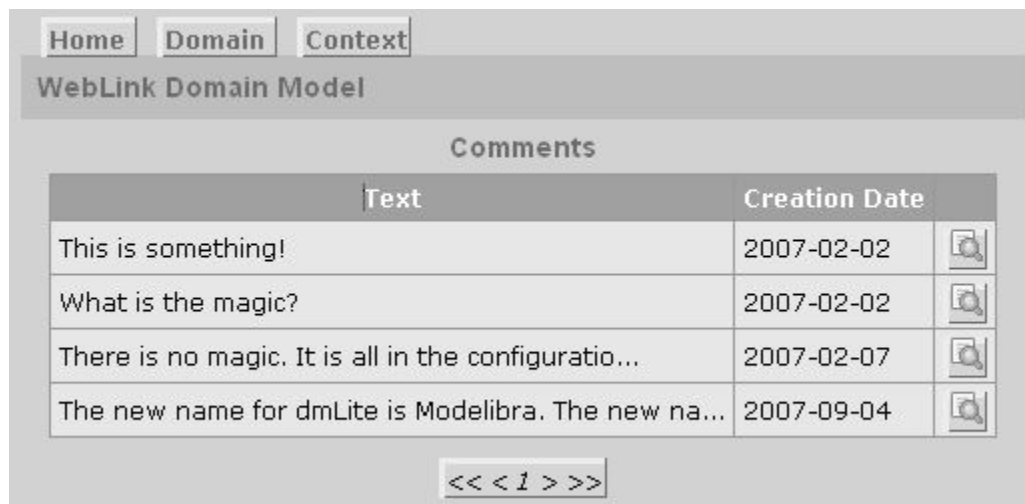


Figure 1.6. Display table of comments

An entity may be added, displayed, edited or removed. For example, a new comment can be added and an existing comment can be displayed with all its details, edited or removed (Figure 1.7). When an entity is added or edited, its properties are validated with respect to the XML configuration of the concept.














Home Domain Context		
WebLink Domain Model		
Comments		
Text	Creation Date	
This is something!	2007-02-02	  
What is the magic?	2007-02-02	  
There is no magic. It is all in the configuratio...	2007-02-07	  
The new name for dmLite is Modelibra. The new na...	2007-09-04	  
<< < 1 > >> 		

Figure 1.7. Update table of comments

Modelibra Software

Modelibra is the OSS family that is used to develop dynamic web applications based on domain models. The software family consists of a graphical design tool called ModelibraModeler, a domain model framework called also Modelibra since it is the core of the Modelibra software family, a web component framework called ModelibraWicket, a collection of CSS declarations, XML, database and Java code generators, and an Eclipse project skeleton called ModelibraWicketSkeleton to start a new project.

The Modelibra software is developed in Java 6. It is assumed in this book that you know Java [Java] and that you have installed the latest version of Java [JDK] on your computer. The Modelibra software is maintained in the Subversion [Subversion] software repository at Google coder [GoogleCode].

If you want to update the software on your computer by using only a difference between the version that you have locally and the version that is in the repository on a remote server, use the Subversion software. Subversion is a software version control system. It allows multiple programmers to work on the same software. Programmers update the software on which they work from the Subversion's repository. They make changes to the code, and then commit the changes back to the repository. Subversion keeps track of the changes and integrates them back into the code repository. If other modifications were made between their last update and commit, Subversion informs programmers about possible conflicts in the code changed by more than one programmer.

In order to make commits to the Subversion code repository, a programmer must be an internal developer of the Modelibra team. However, if you want to checkout the code and later make incremental updates of your local code so that you have the latest version of the software, you do not need to register as a developer.

In order to access the Subversion code repository, you should use the Eclipse IDE (Integrated Development Environment) [Eclipse]. If you have not done it already, download the Eclipse IDE for Java Developers. Subclipse is an Eclipse plug-in that provides the functionality to interact with a

Subversion repository, and to do checkouts, commits and updates. In order to use Subversion from Eclipse, you need to install the Subclipse plugin:

- Eclipse/Help/Software Updates/Find and Install...;
- Search for new features to install/Next;
- New Remote Site...;
- Name: Subclipse 1.z.x;
- URL: http://subclipse.tigris.org/update_1.z.x.

After the installation of the Eclipse plugin for Subversion, open two new perspectives in Eclipse:

- Window/Open Perspective/Other...;
- SVN Repository Exploring;
- Team Synchronizing.

In the SVN perspective connect to the Modelibra's repository:

- File/New/Other.../SVN/Checkout projects from SVN/Next;
- Create a new repository location/Next;
- URL: <http://modelibra.googlecode.com/svn/trunk/> .

If you want to use the latest version of Modelibra software, open the trunk remote directory, select one of the Eclipse projects and check it out by using the pop-up menu. However, if you want to use a specific version open the tags remote directory.

The following list introduces the basic Modelibra projects:

- ModelibraModeler
- Modelibra
- ModelibraWicket
- ModelibraWicketSkeleton
- ModelibraWicketCss
- Educ
- App

ModelibraModeler is a graphical application developed in Swing. It is used to design domain models and to generate their XML configurations. Modelibra is a domain model framework. ModelibraWicket is a web component framework based on Wicket. It is used to produce a web application based on the domain models designed in ModelibraModeler. ModelibraWicketSkeleton is a skeleton web application, which becomes a web application after the code generation. ModelibraWicketCss contains spirals to learn the basic of CSS used in ModelibraWicket. The DmEduc spirals from this book are in the Educ directory. Some other applications can be found in the App directory.

After awhile, in order to have the latest Modelibra software, in the Team Synchronizing perspective of Eclipse, click on the Synchronize SVN (Workspace) button to see if there are changes in the source code since your last visit. If there are changes click on the Incoming mode button to see the changes and use the Update menu item in the pop-up menu to update your local version. If you are not an internal developer you cannot make commits.

In summary:

Use Checkout... to get a project into Eclipse for the first time.

Use Synchronize SVN (Workspace) to see if there are differences between your local version and the repository.

Use Update in the Incoming Mode to update your local version by the last version in the repository.

Summary

Modelibra facilitates the definition and the use of domain models in Java. With Modelibra, a programmer does not need to learn a complex data framework in order to create and save domain models. Modelibra uses Wicket for application views of domain models. Modelibra interprets the domain model and creates the default web application. This application helps developers validate and consequently refine the domain model. In addition, Modelibra has a collection of generic web components that may be easily reused in professional web applications to display or update model entities.

In the next chapter a simple domain model with only one concept will be created. The Eclipse IDE will be used.

Questions

1. What are the main characteristics of an OSS?
2. Is Java an OSS?
3. What is a domain model?
4. What is the major difference between a dynamic web application and a web sites with static HTML pages?
5. In a web application based on a domain model, where is the domain model located?

Web Links

[Domain Model] DSM Forum
<http://www.dsmforum.org/>

[Domain Modeling] Domain Modeling
<http://www.aptprocess.com/whitepapers/DomainModelling.pdf>

[Eclipse] Eclipse
<http://www.eclipse.org/>

[EMF] Eclipse Modeling Framework
<http://www.eclipse.org/emf/>

[Frameworks] Frameworks
<http://st-www.cs.uiuc.edu/users/johnson/frameworks.html>
http://en.wikipedia.org/wiki/Software_framework

[GoogleCode] Google code
<http://modelibra.googlecode.com/svn/trunk/>

[Interface] Java Interface
<http://java.sun.com/docs/books/tutorial/java/concepts/interface.html>

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

[JDK] Java Development Kit
<http://java.sun.com/javase/downloads/index.jsp>

[Modelibra] Modelibra Domain Model Framework
<http://www.modelibra.org/>

[MVC] Building Graphical User Interfaces with the MVC Pattern
<http://csis.pace.edu/~bergin/mvc/mvcgui.html>

OSS
http://www.dwheeler.com/oss_fs_why.html

[OSS Web] Open Source Web Frameworks in Java
<http://java-source.net/open-source/web-frameworks>

[POJO] Plain Old Java Object
<http://en.wikipedia.org/wiki/POJO>

[Silverrun] Silverrun CASE Tools
<http://www.silverrun.com/>

[Spiral Education] Spiral Approach
http://en.wikipedia.org/wiki/Spiral_approach

[Spiral Model] Spiral model
http://en.wikipedia.org/wiki/Spiral_model
<http://www.stsc.hill.af.mil/crosstalk/1995/01/Comparis.asp>

[Struts] Struts Web Framework
<http://struts.apache.org/>

[Subversion] Subversion Repository

<http://subversion.tigris.org/>

<http://www-128.ibm.com/developerworks/opensource/library/os-ecl-subversion/>

<http://open.ncsu.edu/se/tutorials/subclipse/>

[URL] Uniform Resource Locator

<http://www.webopedia.com/TERM/U/URL.html>

[Wicket] Wicket Web Framework

<http://wicket.apache.org/>