Appendices

Appendix A: Modelibra Configuration Validation

In the Modelibra XML configuration file you may reference the domain-config.dtd file for document type definition validations. The purpose of a DTD file is to define the valid building blocks of an XML document. It defines the document structure with a list of valid elements. This file should not be changed by a user of Modelibra.

Elements are the main building blocks of both XML and HTML documents. An element is surrounded by tags. Tags are used to markup elements. A starting tag such as <element-name> marks up the opening of an element, and a closing tag like </element-name> marks up the end of the element.
An element declaration has the following syntax:

```
<!ELEMENT element-name (element-content)>
```

Elements with data content are declared with the data type inside parentheses:

```
<!ELEMENT element-name (CDATA)>
```

CDATA means that an element that contains character data will not be parsed for additional elements.

Elements with children are defined with the name of the children elements inside the parentheses:

```
<!ELEMENT element-name (child-element-name, child-element-name, ...)>
```

or

```
<!ELEMENT element-name (child-element-name*)>
```

The * sign declares that the child element may occur zero or more times inside the parent element.

Attributes provide extra information about elements. In DTD, XML element attributes are defined with the ATTLIST declaration:

```
<!ATTLIST element-name attribute-name attribute-type default-attribute-value>
```

The valid Modelibra XML configuration, according to the domain-config.dtd file, starts, after the usual first line declaration, with the domains root element, followed by zero or more domain elements.

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

```
<!ELEMENT domains (domain*)>
```

The domain element has many child elements such as code and packageCode. The code is mandatory, while the packageCode is optional (the question mark sign). The last child element is models. The domain element has one attribute called oid. The attribute is identifier and it is required.

```
<!ELEMENT domain (code, type, packageCode?, abstraction?, defaultConstruct?, packagePrefix?,
referenceModel?, i18n?, signin?, signinConcept?, shortTextDefaultLength?, pageBlockDefaultSize?,
validateForm?, confirmRemove?, models?)>
<!ATTLIST domain
```

```
	oid ID #REQUIRED
>
```

The domain child elements are defined as regular elements. Only the last one is decomposable, i.e., it has its own XML element. The code element may contain any data, while the abstraction element may be either **true** or **false**. The non-decomposable domain child elements are divided into two blocks separated by an empty line. The first block configures the domain, while the second block defines how the domain will be viewed in the context of an application. Both blocks are used by ModelibraWicket. The second block is not used by Modelibra, since Modelibra does not know anything about views.

```
<!ELEMENT code (CDATA)>
<!ELEMENT type (CDATA)>
<!ELEMENT packageCode (CDATA)>
<!ELEMENT abstraction (true, false)>
<!ELEMENT defaultConstruct (true, false)>
<!ELEMENT packagePrefix (CDATA)>
<!ELEMENT referenceModel (CDATA)>
```

In Modelibra, the package names for classes are constructed by default from the domain, model and concept codes. For example, domain classes are in the package determined by the domain code in lower letters. However, in some cases a developer wants to use a different package name. The packageCode element in combination with the packagePrefix element provides a different package naming strategy. If an element is not present in the configuration, its value is **null**. If an element is present in the configuration, but its value is an empty space (one or more spaces), its value is **not null**.

```
	public String getPackageCode() {
		if (packageCode == null || packageCode.trim().equals("")) {
			return getDomainPackageCode();
		} else {
			if (packagePrefix == null || packagePrefix.trim().equals("")) {
				return packageCode;
			} else {
				return packagePrefix + "." + packageCode;
			}
		}
	}

	private String getDomainPackageCode() {
		if (packagePrefix == null || packagePrefix.trim().equals("")) {
			return getCodeInLowerLetters();
		} else {
			return packagePrefix + "." + getCodeInLowerLetters();
		}
	}
```

A domain may be declared as abstract for the purpose of using its configuration in the inheritance of configurations. As a consequence, the abstract domain object cannot be constructed. This is an advanced feature of Modelibra. Therefore, the default value for the abstraction element is **false**. There is an option in Modelibra where there is no need for specific classes for a domain and a model. This option can only be used if the defaultConstruct element is **true**. The default value is **false**. A domain may have more than one model. One of those models may be a reference model. A reference model contains those concepts that frequently appear in different domains and different models, such as natural languages with their international codes. In that case, the referenceModel element carries the reference model code.

In Modelibra, a domain may have a default application. With the use of ModelibraWicket, a default application is a web application. This application is used by developers and their clients as a mechanism to validate in an easy way a domain model. If a model changes frequently, the productivity of the model view development slows down. This is the reason to use a default application even with the first version of a model. Several features of this application are configurable. The application may have support for various natural languages (i18n is a shorthand for internationalization). The application may provide a sign-in section in the home page. The sign-in concept, in a single model or in the reference model for multiple domain models, is used to validate if a user is a legitimate one. When long text properties are used in sections where multiple entities are displayed, the short text default length may be used to display only the beginning of the text. If there are many entities, they are usually displayed in page blocks of a default size. A form is used to enter data for a new entity or to modify the existing entity. The validation of data including data types may be delegated to Modelibra only if the validateForm element is **false**. If the validateForm element is **true**, in the case of ModelibraWicket, it is Wicket that will do validations. If an entity is removed at the view level, the confirmation for the removal may be requested.

```
<!ELEMENT i18n (true, false)>
<!ELEMENT signin (true, false)>
<!ELEMENT signinConcept (CDATA)>
<!ELEMENT shortTextDefaultLength (CDATA)>
<!ELEMENT pageBlockDefaultSize (CDATA)>
<!ELEMENT validateForm (true, false)>
<!ELEMENT confirmRemove (true, false)>
```

Since there are many elements in the domain configuration, most of them have a default value:

```
false  abstraction
false  defaultConstruct

false  i18n
false  signin
48     shortTextDefaultLength
16     pageBlockDefaultSize
true   validateForm
true   confirmRemove
```

A domain may have several models.

```
<!ELEMENT models (model*)>
```

The model element has many child elements such as code and abstraction. The last child element is concepts. The model element has one attribute called oid. The attribute is identifier and it is required.

```
<!ELEMENT model (code, abstraction?, extension?, extensionDomain?, extensionDomainType?, extensionModel?,
author?, packageCode?, persistent?, persistenceType?, persistenceRelativePath?, persistenceConfig?,
defaultLoadSave?, datePattern?, session?, concepts?)>
<!ATTLIST model
      oid ID #REQUIRED
>
```

The model child elements are defined as regular elements. Only the last one is decomposable.

A model may be declared as abstract. As a consequence, the abstract model object cannot be constructed. A model may inherit a configuration from another model in the same or a different domain. Overriding is possible by having a specific value for an XML element. A model may have one or several authors, but as a single value. The model package code serves the same purpose as the domain package code. A model is persistent by default. Its persistence type is then xml. The persistence type may also be jdbc for relational databases or db4o for the db4o object database [db4o]. The persistence relative path is a path of directories within the project where data will be saved. By default, Modelibra will save data using a path constructed in the following way: data/xml/domaincode/modelcode. Both domain and model codes are all in small letters. In the case of a DBMS that requires a configuration, the persistence configuration file name is indicated. By default, a model is loaded automatically and all data are saved automatically without an intervention of a programmer. The use of dates is quite a tricky subject in Java. In order to simplify it, the default pattern of yyyy-MM-dd is used. Modelibra has sessions, actions that can be undone and transactions. Both actions and transactions must be used within a session. However, by default, Modelibra starts without creating a session.

```
<!ELEMENT code (CDATA)>
<!ELEMENT abstraction (true, false)>
<!ELEMENT extension (true, false)>
<!ELEMENT extensionDomain (CDATA)>
<!ELEMENT extensionDomainType (CDATA)>
<!ELEMENT extensionModel (CDATA)>
<!ELEMENT author (CDATA)>
<!ELEMENT packageCode (CDATA)>
<!ELEMENT persistent (true, false)>
<!ELEMENT persistenceType (xml, jdbc, db4o)>
<!ELEMENT persistenceRelativePath (CDATA)>
<!ELEMENT persistenceConfig (CDATA)>
<!ELEMENT defaultLoadSave (true, false)>
<!ELEMENT datePattern (CDATA)>
<!ELEMENT session (true, false)>
```

Default values for the model configuration:

```
false         abstraction
false         extension
true          persistent
xml           persistenceType
true          defaultLoadSave
yyyy-MM-dd    datePattern
false         session
```

In general, a model may have many concepts.

```
<!ELEMENT concepts (concept*)>
```

The concept element has many child elements such as code and abstraction. The last two child elements are properties and neighbors. They are both decomposable. The concept element has one attribute called oid. The attribute is identifier and is required.

```
<!ELEMENT concept (code, abstraction?, extension?, extensionDomain?, extensionDomainType?,
extensionModel?, extensionConcept?, extensionWithNeighbors?, entitiesCode?, packageCode?, min?, max?,
entry?, fileName?, index?, display?, displayType?, add?, remove?, update?, properties?, neighbors?)>
<!ATTLIST concept
     oid ID #REQUIRED
>
```

The concept child elements are defined as regular elements. Only the last two are decomposable.

A concept may be declared as abstract. As a consequence, the abstract concept object cannot be constructed.

A concept may inherit a configuration from another concept in the same or a different model, in the same or different domain. Overriding is possible by having a specific value for an XML element. If a concept inherits its configuration from another concept, by default it inherits only properties and not neighbors (relationships). In Modelibra, a concept has two Java classes, one of the IEntity type and another of the IEntities type. The concept code is used to name the concept's Java class of the IEntity type. By default, the entities code is derived from the concept code with the s character for plural at the end. However, a specific name can be given to that class. The concept package code is what we have seen in both domain and model configurations. The number of entities in the same collection may be restricted by the minimal and maximal cardinalities. By default, the concept is not entry into the model. In the case of the xml persistence, a concept is saved within a data file that has the same name as the concept code but with all lower letters. The extension of a data file is xml. A concept may be indexed. By default, it is not.

```
<!ELEMENT code (CDATA)>
<!ELEMENT abstraction (true, false)>
<!ELEMENT extension (true, false)>
<!ELEMENT extensionDomain (CDATA)>
<!ELEMENT extensionDomainType (CDATA)>
<!ELEMENT extensionModel (CDATA)>
<!ELEMENT extensionConcept (CDATA)>
<!ELEMENT extensionWithNeighbors (true, false)>
<!ELEMENT entitiesCode (CDATA)>
<!ELEMENT packageCode (CDATA)>
<!ELEMENT min (CDATA)>
<!ELEMENT max (CDATA)>
<!ELEMENT entry (true, false)>
<!ELEMENT fileName (CDATA)>
<!ELEMENT index (true, false)>
```

At the view level, a concept may be displayed, and if so, the display type for a collection of entities may be a table, a list or a slide. Entities may be added, removed and updated. By default, all view boolean configurations are **true**, and the table like display of entities a preferred format.

```
<!ELEMENT display (true, false)>
<!ELEMENT displayType (table, list, slide)>
<!ELEMENT add (true, false)>
<!ELEMENT remove (true, false)>
<!ELEMENT update (true, false)>
```

In summary, default values for the concept configuration are:

false	abstraction
false	extension
false	extensionWithNeighbors
code + “s”	entitiesCode
true	defaultLoadSave
0	min
N	max
false	entry
code.xml	fileName
false	index
true	display
table	displayType
true	add
true	remove

```
true        update
```

A concept has usually several properties.

```
<!ELEMENT properties (property*)>
```

The property element has many child elements such as code and extension. However, there are no further decomposable child elements. The property element has one attribute called oid. The attribute is identifier and is required.

```
<!ELEMENT property (code, extension?, extensionProperty?, propertyClass?, derived?,  validateType?,
validationType?, maxLength?, required?, sensitive?, defaultValue?, autoIncrement?, unique?, index?,
reference?, referenceNeighbor?, display?, update?, displayLength?, essential?, scramble?,
whitespaceAllowed?, referenceDropDownLookup)>
<!ATTLIST property
     oid ID #REQUIRED
>
```

The property child elements are defined as regular elements.

A property may inherit declarations from a property in the inherited concept. Overriding is possible by having a specific value for an XML element. A property class is a Java class with the full name. In Modelibra, a property class cannot be a base type. A derived property cannot be saved nor loaded. It has only the get method, in which any derivation algorithm can be introduced. For a property of the String type that is not derived, an additional validation may be done, for example, if a value is a well-formed email. A String value may have a maximal length. A value may be required and in that case **null** is not accepted. If a property is sensitive it cannot be exported from a concept or from a model. A sensitive property may be a salary of an employee. A property may have a default value. If the property class is Integer, its value can be automatically incremented. The first value will be 1. A property may be a part of the unique combination of properties and/or parent neighbors. At most one unique combination for the concept is possible in Modelibra. A property may be a part of the index combination of properties and/or parent neighbors. At most one index combination for the concept is possible. A property may reference an external neighbor.

```
<!ELEMENT code (CDATA)>
<!ELEMENT extension (true, false)>
<!ELEMENT extensionProperty (CDATA)>
<!ELEMENT propertyClass (CDATA)>
<!ELEMENT derived (true, false)>
<!ELEMENT validateType (true, false)>
<!ELEMENT validationType (CDATA)>
<!ELEMENT maxLength (CDATA)>
<!ELEMENT required (true, false)>
<!ELEMENT sensitive (true, false)>
<!ELEMENT defaultValue (CDATA)>
<!ELEMENT autoIncrement (true, false)>
<!ELEMENT unique (true, false)>
<!ELEMENT index (true, false)>
<!ELEMENT reference (true, false)>
<!ELEMENT referenceNeighbor (CDATA)>
```

A property may be displayed or not. If displayed it may be updated or not. Its display length can be shorten. If a property is essential for the concept, it will be displayed in a table of entities If a property is scrambled, its value will be hidden from a user. A password property is usually configured to be scrambled. If a whitespace is allowed, a distinction between the **null** and whitespace is made at the view level. If nothing is entered in a form field representing a property, the property will become **null** if not required. However, if one or more white spaces is entered and no other characters, the property will have those white spaces as its value. This is true only for properties of the String class. If a whitespace is not allowed, the **null** will stand even for a whitespace. If a property is reference, a lookup of its reference neighbor is done by default using a drop down choice. A lookup can also be done using a table of neighbor entities in a different page.

```
<!ELEMENT display (true, false)>
<!ELEMENT update (true, false)>
<!ELEMENT displayLength (CDATA)>
<!ELEMENT essential (true, false)>
<!ELEMENT scramble (true, false)>
<!ELEMENT whitespaceAllowed (true, false)>
<!ELEMENT referenceDropDownLookup (true, false)>
```

In summary, default values for the property configuration are:

false	extension
false	derived
false	validateType
false	required
false	sensitive
false	autoIncrement
false	unique
false	index
false	reference
true	display
true	update
false	essential
false	scramble
false	whitespaceAllowed
true	referenceDropDownLookup

The concept element has the neighbors child element that is followed by one or more neighbor child elements.

```
<!ELEMENT neighbors (neighbor*)>
```

The neighbor element has many child elements such as code and extension. However, there are no further decomposable child elements. The neighbor element has one attribute called oid. The attribute is identifier and is required.

```
<!ELEMENT neighbor (code, extension?, extensionNeighbor?, destinationConcept, inverseNeighbor?,
internal?, partOfManyToMany?, type, min?, max?, unique?, index?, addRule?, removeRule?, updateRule?,
display?, update?, absorb?)>
<!ATTLIST neighbor
     oid ID #REQUIRED
>
```

The neighbor child element are defined as regular elements.

A neighbor may inherit declarations from a neighbor in the inherited concept. Overriding is possible by having a specific value for an XML element. A neighbor for its concept is a relationship direction from the (source) concept to its destination concept. The inverse neighbor is the opposite direction of the same relationship. A neighbor is either internal or external. A neighbor may be a part of a many-to-many relationship between the participating concepts. A neighbor type is either parent or child. A parent neighbor must have the maximal cardinality of 1. A child neighbor usually has the maximal cardinality of N. For the parent neighbor the minimal cardinality is often 1. For the child neighbor the minimal cardinality is usually 0. A neighbor may be a part of the unique combination of properties and/or parent neighbors for its concept. A parent neighbor may be a part of the index combination of properties and/or parent neighbors. The add, remove and update rules may be used for referential integrity constraints when the persistence type is jdbc.

```
<!ELEMENT code (CDATA)>
<!ELEMENT extension (true, false)>
<!ELEMENT extensionNeighbor (CDATA)>
<!ELEMENT destinationConcept (CDATA)>
<!ELEMENT inverseNeighbor (CDATA)>
<!ELEMENT internal (true, false)>
<!ELEMENT partOfManyToMany (true, false)>
<!ELEMENT type (parent, child)>
<!ELEMENT min (CDATA)>
<!ELEMENT max (CDATA)>
<!ELEMENT unique (true, false)>
<!ELEMENT index (true, false)>
<!ELEMENT addRule (NONE, RESTRICT, DETACH)>
<!ELEMENT removeRule (NONE, RESTRICT, CASCADE)>
<!ELEMENT updateRule (NONE, RESTRICT, DETACH)>
```

A neighbor may be displayed or not. If displayed, it may be updated or not. It may be absorbed but only if it is of the parent type. A neighbor is absorbed by including the essential properties of the destination concept in the display of the concept of the neighbor.

```
<!ELEMENT display (true, false)>
<!ELEMENT update (true, false)>
<!ELEMENT absorb (true, false)>
```

In summary, default values for the property configuration are:

false	extension
true	internal
false	partOfManyToMany
false	unique
false	index

true	display
true	update
true	absorb