MQSeries Everyplace - Administration tool

MetaModel XML Format Specification

Version 1.0

17 August 2000

Phill van Leersum
MQSeries Development
IBM United Kingdom Laboratories
Hursley Park
Hursley
Hampshire, SO21 2JN
UK

phillvl@uk.ibm.com

Take Note!

Before using this report be sure to read the general information under "Notices".

First Edition, August 2000

This edition applies to Version 1.0 of *MQSeries Everyplace - Administration tool* and to all subsequent releases and modifications unless otherwise indicated in new editions.

Table of Contents

Notices

Trademarks and service marks

Summary of Amendments

Date	Changes
17 August 2000	Initial release

Preface

This document describes the format of the XML used as a persistant representation of a UIA MetaModel.

Bibliography

List any supporting publications here otherwise delete this page. Use following format:

- *User Interface Architecture*
- *MQSeries Everyplace - Administration tool design*

Overview

Metamodel defines a lot of data that describes the appearance and behaviour of administered system. Very little is then required to complete a system - just the specific behaviour upon invocation of operations.

Metamodel contains data on the following broad areas:

The different types objects that are allowed

- Attributes of the objects, default values, Attribute Types (constraints)
- Operations of the objects, menu paths, standard prompts
- Permitted child types for objects, categories
- The class used to provide the functionality for the object type

The Attribute Types (constraints), enumerations, strings, integers

The class used to provide the functionality for the metamodel

The initial (root) object to create

These are described in detail later.

Caption, Name, Mapped Value

The terms Caption, Name and Mapped Value are used consistently throughout the metamodel. They have specific meanings in this context:

Name: All entities defined in the metamodel have name. This is the string by which the program recognises the entity. The name should never be visible to the user of a renderer, and probably has no meaning for the underlying system. If no name is specified, then the name is the same as, or derived from, the caption.

Caption: The caption is the text that the user sees when an entity is represented. The captionis never used by the program except to represent the underlying object. As a result the captions can all be translated without affecting the behaviour of the program. This make NLS support easy. If no caption is defined then it defaults to the name.

Mapped Value: Certain metamodel entities directly represent an entity in the underlying system (business logic). It is useful to store some business logic specific data in the metamodel. For exmple, an enumerated type might allow a value wth a caption of 'Default Queue Rules' which could have a mapped value of 'com.ibm.mqe.MQeQueueAttributeRules'. The user would see the simple english of the caption, but the mapped value can be supplied to the undelying business logic.

MetaModel

```
<metamodel className="com.ibm.uia.metamodel.MetaModel" rootType="Administered System"
rootName="Example System">

               <attributeType......../>

               <metaObject.........../>

</metamodel>
```

The metamodel tag wraps all the other data in the file. At the top level the tag has three xml attributes :

- *className*: the name of the class implementing the metamodel functionality. This defaults to 'com.ibm.uia.metamodel.MetaModel' (and so is unnecessary in the example above).
- *rootType*: the name of a MetaObject contained within the meta model. When the metamodel is loaded, an instance of this type will be created as the root of the administered system.
- *rootName*: this is the caption of the instance root type created initially by the renderer.

The metamodel tag contans zero or more tags of each of the following types:

- *attributeType*: a constraint on values allowed to an attribute.
- *metaObject*: a definition of a renderable object.

MetaObjects

```
<metaObject name="RootObject" className="com.ibm.uia.metamodel.ExampleObject" >

               <attributeDefinition...../>

               <operation....../>

               <childType...../>

</metaObject>
```

Meta Object Tags have several attributes:

- *name*: the internal name of the type/metaObject
- *className*: the name of the class that will be loaded to implement the behaviour of instances of the meta object.
- *image*: the name of a file (.gif or .bmp) that will be used to represent instances of the meta object.
- *category*: defines if the meta object instance have category-like behaviour (defaults to 'false')
- *autoLoad*: if the meta object is defined as a category, then this modifies its initial behaviour (defaults to 'true')

Meta Objects contian the following tags:

- AttributeDefinition
- operation
- childType

Child Types

MetaObjects can define other metaobjects as permissable child types. RootObject defines ChildType3 as a permissable child:

```
<metaObject name="RootObject" className="com.ibm.uia.metamodel.ExampleObject" >

        <attributeDefinition ............./>

        <operation............................/>

        <childType name="ChildType3"/>

</metaObject>
```

In the above xml fragment the childType tag indicates that instances of the RootObject type are allowed to have instances of ChildType3 as direct children. Attempts to add children of types not secificaly allowed will fail. The type checking allows for autmation of many tasks, including checking if a 'paste' operation is allowed.

ChildTypes must be valid child types that exist within the same metamodel as the parent Type.

Categories as child types

Categories act as containers for one or more meta objects types. This is best explained with an example:

```
<metaObject name="RootObject" className="com.ibm.uia.metamodel.ExampleObject" >

        <attributeDefinition ............./>

        <operation............................/>

        <childType name="Child1Category"/>

</metaObject>

<metaObject name="Child1Category" category="true" image="folder.gif" >

        <operation ............./>

        <childType name="ChildType1"/>

</metaObject>
```

The example metamodel defines the category Child1Category with child type of Child1Type - the category contans only instances of Child1Type (it could contain more, but in the example there is only one defined).

Because RootObject is defined as having Child1Category as a child, it can now contain (indirectly) Child1Type objects.

More importantly, when internal searches for children, the children of the category are considered to be children of the parent of the category. It is as if the child is of the category is actually a child of the category parent.

Whenever a child of type Child1Type is added to a RootObject, the child appears as a child of the category.

AutoAdd behaviour is only aparent when a parent of a category is created. If the category is marked as autoadd (or not specified, as autoload defaults to true) then an instance of the category is created when the parent instance is created.

Operations

```
<operation caption="Edit.Delete" name="Delete"/>

<operation caption="Edit.Cut" name="Cut"/>

<operation caption="Edit.Paste" name="Paste"/>
```

This operation tag defines the delete, cut and paste operations. The captions define what is visible to the user. The dot ('.') separators break the 'path' of the command. Paths are used to group operations, for example in menus. The above operations would require a menu item of 'Edit' with cascade menu items for 'Delete', "Cut", and "Paste". This nesting can continue to an arbtrary level.

Since the name is considered distinct from the caption, and the program only concerns itself with the name, the arrangement and wording of the menu items can be changed without affecting the behaviour (internationalisation).

If no name is specified then the name is assumed to be the last element of the caption (eg "Delete").

When an operation is selected, then the internal representation of the operation is passed to the instance of metaObject for performance:

public boolean performOperation(RenderableOperation op, String[] data, Renderer renderer)

The class com.ibm.uia.metamodel.UIObject handles default operations such as delete, copy, paste, cut, and edit attributes.

The class com.ibm.uia.metamodel.ExampleObject adds the functionality to creat a new child if the name of the operation is the same as the name of a valid child type.

Operation Prompts

Operations may have implied prompts. Currently only stringPrompt and filePrompt are supported.

String Prompts

String Prompts are defined as follows:

```
<operation caption="New.ChildType3" name="ChildType3">

       <stringPrompt caption="Enter childType3 Details">

              <element cation="name" default="childType3Name"/>

       </stringPrompt>

</operation>
```

The prompt appears as a dialogue after the user selects the operation (meu item, button...) and before the performAction() method is called. If the user selects 'ok' or otherwise accepts the values in the prompt, then the data collected is passed to the performOperation method. If the user selects cancel/abort or similar, then performOperation is never called.

For string prompts, the caption appears in the title bar of a dialogue. Each element appears as a label (the caption), and a text entry field initialised with the default value. When the user presses ok, the strings in the entry fields are presented as 'String[] data' in the perform operation method call.

FilePrompt

```
<operation caption="Edit.Import XML">

       <loadFilePrompt caption="Select file to import">

              <element directory="c:\test" filename="*.xml"/>

       </loadFilePrompt>

</operation>
```

Caption is title of file prompt.

directory is initial directory (defaults to current directory).

filename is initial filename (acts as filter if wildcarded)

Attribute Definitions

Attribute defintions declare the set of attributes posessed by each instance of a metaobject type.

```
<attributeDefinition name="Name" type="ASCII" mappedName="admname" edit="false" />
```

- *name*: the internal name of the attribute.
- *caption*: the visible representation of the attribute. If none is supplied defaults to the name (bad practice).
- *type*: the attributeType that constrains values allowed for this attribute (*vide infra*). This must be the name of a valid attribute type held within this metamodel. The fragment above uses the type called 'ASCII'.
- *mappedName*: a string used to represent the attribute in the underlying system.

- *edit*: indicates whether the user can edit the value. defaults to true.
- *default*: a default value to use for this attribute. The value must be a valid valuue for the constraning attribute type. If no default valu is supplied, then the default for the constraining attribute type is used.

Attribute Types

Attribute Types constrain the values allowed by an attribute

```
<attributeType name="ASCII" mode="string">

       <defaultValue caption="some string data" />

</attributeType>
```

The fragment above shows part of the definition of the type .ASCII'. It shows only thos parts common to all attributeTypes:

- *name*: the string used to identify uniquely the type internally. No two types can have the same name.
- *mode*: defines the basic mode of operation of the type. Currently this must be one of:
  - *String*: a simple string with no formatting requirements
  - *Integer*: a string representation of an integer value in decimal notation. non-digits are invalid
  - *Enumerated*: achice from a defined set of values.
- *defaultValue*: an allowed value is specified as the caption. This must be a valid value.

The modes are detailed below:

String

```
<attributeType name="ASCII" mode="string">

       <maxLength value="128" />

       <defaultValue caption="null" />

</attributeType>
```

String attribute types are the simlest constraint. Any string is allowed, up to an optional maximum length.

Integer

Integer types represent a constraint to a valid decimal representation of a number.

```
<attributeType name="BYTE" mode="integer">

        <defaultValue caption="0" />

</attributeType>
```

Enumerated

Enumerated types represent a constraint to one of a predefined set of values:

```
<attributeType mode="enumeration" name="BOOLEAN">

        <enumeration>

                <literal mappedValue="1" caption="true" />

                <literal mappedValue="0" caption="false" />

        </enumeration>

        <defaultValue caption="false" />

</attributeType >
```

In the above xml code the type called BOOLEAN is defined. This is an enumeration with two values. One value appears to the user as 'true' and the other as 'false'.

The values 'true' and 'false' are mapped to the values '1' and '0' in the underlying system.