OpenSHORE

OpenSHORE-XML and meta modelling

Version 0.3

www.openshore.org

History

Version	*State*	*Date*	*Author(s)*	*Description*
0.1	finished	29.11.2002	Tammo Schnieder	Created
0.2	finished	09.12.2002	Tammo Schnieder	Added review comments from Mark Kulke
0.3	in work	13.12.2002	Tammo Schnieder	Added review comments from Denis Kuniß

Content

1 Introduction

OpenSHORE is a repository for structured text documents. You can put these documents into the repository and navigate through them or let OpenSHORE answer queries.
Generally, OpenSHORE separates content, structure and layout. The **content** is the text of a document. The **structure** are logical sections of the text, that you can define. The **layout** is how the text is displayed.
In this document, we are mainly writing about the content and structures.
The internal document format of OpenSHORE is XML. The structures of documents that are imported into an instance have to be compliant with a prior imported meta model.
In this document, you will see, how XML has to be structured for OpenSHORE and how the XML is dependant from the meta model of an instance.
The layout is configured with cascading stylesheets (CSS). This is not part of this document.

1.1 Used conventions

In the text, we use the `Courier` font for sources and related things. In syntax definitions, if there is a variable text that you can change with a value of your choice, we use *`Courier italic`*.
Example:

```
Dokumenttyp AnyType
```

Here, the word “Dokumenttyp” is fixed, but you can change “AnyType” to another value.

In syntax definitions, we use the following notation:
Block brackets ([and]) are used to mark an optional part. If a part can occur zero or more times, it is put in round brackets and marked with an asterisk: (this can occur zero or more times)*.
If a part can occur one or more times, it is marked with round brackets and a plus sign: (this occurs one or more times)+.
If you can choose between options, these are marked in brackets and are divided with a vertical bar: choose (one|two|three).

1.2 German / English translations

There are some naming issues that are by now a little strange.
This is, because OpenSHORE was born of a pure german project and is by now learning other languages (mainly english).
As for all language beginners, not everything can be expressed in other than your mother language.
The OpenSHORE team is working to translate everything first in english. But it will take a little time until this is finished.
So you will find many german words in OpenSHORE. For your help, here is a table of translations:

German	***English***
Metamodell	meta model
Dokumenttyp	document type

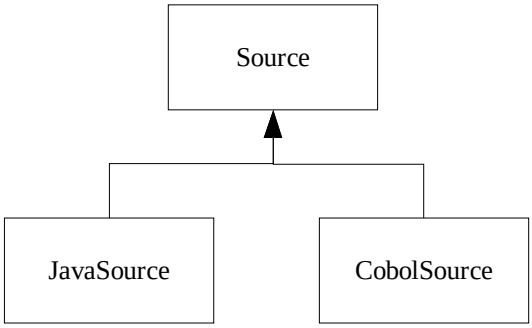
German	***English***
Objekttyp	object type
Beziehungstyp	relationship type
“ist ein”, “ist eine”	is a
“ist nicht volltextindizierbar”	Is not indexable for full text search
Elternname	Parent name
Elterntyp	Parent type
ist Teilmenge von	is subset of
Richtung	direction
Quelle	Source
von..nach	from..to
bis	until (for cardinality)
ist definiert in	is defined in
Quelltyp	type of source
Quellname	name of source
Zieltyp	type of target
Zielname	name of target

2 Meta modelling

2.1 Introduction

In OpenSHORE, you store structured data that is the content of the files you import. The definition of the structures you want to store is declared in the meta model. Here you write down, which document types, object types and relationship types you want to see in OpenSHORE.
Document types are files with content of the same kind e.g. programming sources, concepts, interface definitions, use cases and so on. A specific single file then is a document for OpenSHORE.
Inside of the documents, you find objects. Objects are parts of the text of a document or sometimes the whole text will be an object. You are free to decide, what you want to declare as an object. Sometimes an object can be a whole chapter, sometimes a specific paragraph and sometimes single words. Examples of object types are: use cases, methods, functions, test cases, test reports, change requests and so on.
Between objects and documents, you have some dependencies, which you define as relationships. A relationship type is a kind of a relationship between object types and document types – also mixed. So you may have a relationship between a java class and a java method (two objects), or you have a relationship between a C++-class and a header file (an object and a document). Examples of relationship types are: “use case involves actor”, “file includes file” or “glossary explains word”.

For all these types, you can use inheritance. For example, you have the document types “Source”, “JavaSource” and “CobolSource”. The types “JavaSource” and “CobolSource” are concrete types of the general type “Source”. Then you may have java and cobol documents, but you may not have a document of the general type “Source”. This type is called an “abstract” type.
If you define object and relationship types, it is common to use abstract types as explained in the next chapter. Concrete types are corresponding to the XML-Tags that are inserted by a parser in the documents that you want to import to OpenSHORE.



Picture 1: Example for a document type hierarchy

If you work with an OpenSHORE repository, you can request to see documents of an abstract type, and the result are documents of children types of the abstract type. Related to the example above, if you tell OpenSHORE to show you documents of “Source”, you get all “JavaSource” and “CobolSource” documents.

2.2 Strategies for designing a meta model

If you are designing a meta model, there are a few things that have been useful in the practice:
- define only few relationship types
- if possible, define relationships between parent types, so that they are valid for many concrete types
- for concrete types of documents and for programming languages define concrete object types (e.

g. for that programming language).

- If you are designing a meta model for a programming language, have a look at the OpenSHORE meta model for programming languages, which is installed with OpenSHORE in the folder .../OpenSHORE/parser/meta and look into the “OpenSHORE Parser Cookbook” (right now, only available in german).

3 OpenSHORE XML and its meta model definition

In OpenSHORE there are three types that are fixed: document type (Dokumenttyp) and object type (Objekttyp) – if we mean both, we use the term ressource type – and the third is the relationship type (Beziehungstyp). With these types, you define, how the structure of documents can appear and how information is linked. The definition of this structure is written down in the meta model.
The concrete structure of a document is identified by a parser (or scanner/pattern matcher). The parser then transforms the original document into XML.
Each structure is represented by a specific XML-tag. The names of the XML-tags are the types that you have previously defined in the meta model.
Lets look into a simple example that should give you an idea how this works:

meta model:

```
Dokumenttyp Glossary

Objekttyp Term
```

imported document:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<Glossary Name=”project_glossary.txt”>
This is the glossary of our project.

Term:      Description:
<Term Name=”abstract_type”><Hotspot>abstract type</Hotspot>   A  type  of  which
you may not import concrete entities.</Term>
<Term Name=”calculator_example”><Hotspot>calculator example</Hotspot>     The
very first example that we imported in the old SHORE for presenting it at a sd&m
conference in 1998.</Term>
...
</Glossary>
```

3.1 XML for a document type

All files that you import into OpenSHORE are documents. A document must be of a specific type that you have declared in your meta model.

3.1.1 Document type declaration in the meta model

In the meta model you declare a document as follows:

Syntax	Dokumenttyp *name_of_document_type* (ist (ein|eine) *name_of_parent_document_type*)* [ist nicht volltextindizierbar]
Examples	Dokumenttyp Source Dokumenttyp C-Source ist ein Source

You first begin with the keyword “Dokumenttyp”. After that a unique name follows.
You can use inheritance. That means, that you declare, that a document type is of the type of another document type by adding “ist ein” or “ist eine” after the name declaration.
This makes the parent document type an abstract type. That means, that you can not have files that are of the parent document type.

3.1.2 XML for Document types

OpenSHORE compliant files start with a XML-declaration. The next element is the beginning XML-tag of the document type of that file. Inside of the document type tag, there is the whole content of the document.

Syntax:

```
XML declaration
DocTypeDeclBegin
DocumentContent
DocTypeDeclEnd
```

Example:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<Glossary Name=”project_glossary.txt”>
...here is the content...
</Glossary>
```

XML declaration

The XML declaration is the very first line in each file and is a “normal” XML declaration like the one in the example.

Document type declaration

The document type declaration has the following syntax:

```
<DocTypeName Name="aName">...</DocTypeName>
```

There must be an opening and a closing XML-tag for a DocTypeName. The DocTypeName is the name of a document type that must be declared in the meta model. The name of the document must be unique for this document type. If you import a second document of the same type with the same name, this will replace an already existing document.

There are some additional attributes, that influence the processing of documents of the OpenSHORE server: “Elternname” and “Elterntyp” (in english: “parent name” and “parent type”).

These Attributes are always coming together and mean, that this document is a child document and is dependent from another document, the parent document. If the parent document is deleted or imported for a second time, then all child documents are deleted.
Child documents can be deleted manually and may point to not yet imported parent documents.
If the child document points to a non-existant parent, then you can see this document under the Menu “Administration > Zeige undefinierte Resourcen”. In that case, if than the parent document is imported, the parent “adopts” the childs that are referencing to it. But a second import of the parent document – or if you delete it – will also delete all its children.
If a document points to itself as a parent, this is ignored.
You may not reference to an abstract document type, because there may be several documents with the same name of a concrete type. Then the OpenSHORE server does not know when to delete the children.

Here is an example of an entry of a document with a parent:

```
<JavaQuelltext
   Name="de.sdm.net.Buffer"
   Elternname="de.sdm.net.CompoundPanel"
   Elterntyp="JavaQuelltext">
```

In this example, the document “`de.sdm.net.Buffer`” of the type `JavaQuelltext` is dependent from the (parent) document “`de.sdm.net.CompoundPanel` ” of the type `JavaQuelltext`.

3.2 XML for objects

Inside of documents are structured text passages or single words that are marked as objects.

3.2.1 Object type declaration in the meta model

In the meta model you declare the type of an object as follows:

Syntax	Objekttyp *name_of_object_type* (ist (ein|eine) *name_of_parent_object_type*)* (ist definiert in document_type)+
Examples	Objekttyp TestCase ist definiert in TestSpecification Objekttyp C-Function ist eine Function ist definiert in C-Source Objekttyp Method ist ein Member ist eine Function ist definiert in Source

You first begin with the keyword “Objekttyp”. After that a unique name follows.
You can use inheritance, that means, that the object type is of the type of another object type by adding “ist ein” or “ist eine” after the name declaration.
This makes the parent object type an abstract type. That means, that you can not have concrete objects that are of the parent object type.
For each object type declaration, you have to declare, in which document types it may appear by writing “ist definiert in” (is defined in) and then a document type – even if it would be clear by the parent object. You may also use abstract types as a parent.

3.2.2 XML for Object types

A documents content is text. Some text passages that you want to define as objects are surrounded by specific XML-tags. You may also define objects nested inside of objects.

Syntax:

```
<ObjectTypeName Name="Name_of_object">...</ObjectTypeName>
```

Example:
(the content of a file named “GUI42.txt”)

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<TestSpecification Name=”GUI42.txt”>
      <TestCase Name=”GUI-Tests-42”>GUI-Tests: <Hotspot>Case 42</Hotspot>
      <TestStep Name=” GUI-Tests-42.1”>Step <Hotspot>1</Hotspot>
      In the Menu “Help” click on “About...”.
      The Window “About this Application” opens.</TestStep>
      </TestCase >
</TestSpecification >
```

This example is a complete document of the type `TestSpecification`. Its content is an object of the type `TestCase` with one `TestStep` inside.

Hotspots

Inside of an object, there is the special XML-tag `Hotspot`. This tag marks, what you see as clickable area inside of the object. Each Hotspot belongs to the first embracing object (or relationship as you will see later). If a piece of text should be the Hotspot for more than one object, that is also possible.

Object Naming

If you look at the name of the `TestStep`, you see a common practice to produce unique names. The name is built by concatenating the embracing object with the name of the current object separated by a dot.

Multiple Hotspots

If one Hotspot shall be used by more than one object, you nest these objects inside and add to them the attribute `Hotspot="multiple"`. The most inner objects embraces the Hotspot element.

Example:

```
<Namensraum_definiert_Element
  Quelltyp="DTDElement"
  Quellname="KNLL_Orders"
  Zieltyp="DTDAttribut"
  Zielname="KNLL_Orders.description"
  Hotspot="multiple">
      <DTDAttribut
        Name="KNLL_Orders.description"
        Hotspot="multiple">
            <Hotspot>description</Hotspot>
      </DTDAttribut>
</Namensraum_definiert_Element>
```

Here you see, that both – a relationship and an object – are using the Hotspot `description`.

3.3 XML for relationships

3.3.1 Relationship type declaration in the meta model

In the meta model you declare a relationship as follows:

Syntax	Beziehungstyp *name_of_relationship_type* [alias *short_name*] von n1 bis (n2|*) *ressource_type* nach n3 bis (n4|*) *ressource_type* (ist Teilmenge von *name_of_parent_relationship_type*)* (ist definiert in *document_type*)+
Examples	Beziehungstyp Source_includes_Source alias includes von 0 bis * Source nach 0 bis * Source ist definiert in Source Beziehungstyp Block_calls_CallingTarget alias calls von 0 bis * Block nach 0 bis * CallingTarget ist Teilmenge von Block_uses_Element ist definiert in Source

You first begin with the keyword “Beziehungstyp”. Then a unique name follows, and optionally an alias, which is used for displaying.
The next definition is the source of the relationship: which type is the starting point. Then, you define, which type is the target of the relationship. `ressource_type` means, that you can use an object or a document type here – even mixed types are allowed: you can have relationships from document types to object types and vice versa. The cardinality (n1 -n4) is not used by OpenSHORE at this time.
After the definition of the involved ressources, you can define, if this relationship is inherited by another relationship by adding “ist Teilmenge von” followed by the relationship type. If you use inheritance, you must make sure, that the types of the source and sink are of the same types or of subtypes of the parent types.
Using inheritance, makes the parent relationship type an abstract type. That means, that you can not have concrete relationships of the parent relationship type.
As the last declaration, you must specify in which document types this relationship may occur (ist definiert in).

3.3.2 XML for relationship types

With relationships you define connections between objects and documents.

Syntax:

```
<RelationshipTypeName
  Quelltyp="RessourceType"
  Quellname="RessourceName"
  Zieltyp="RessourceType "
  Zielname="RessourceName ">
...
</RelationshipTypeName>
```

Example:
(the enhanced content of the file “GUI42.txt”)

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<TestSpecification Name=”GUI42.txt”>
      <TestCase Name=”GUI-Tests-42”>GUI-Tests: <Hotspot>Case 42</Hotspot>
      <TestStep Name=”GUI-Tests-42.1”>Step <Hotspot>1</Hotspot>
      In the Menu “Help” click on “<TestStep_tests_GuiObject
      Quelltyp="TestStep"  Quellname="GUI-Tests-42.1"
      Zieltyp="GuiMenuEntry" Zielname="Help.About"><Hotspot>About...</
      Hotspot></RelationshipTypeName>”.
      The Window “About this Application” opens.</TestStep>
      </TestCase>
</TestSpecification>
```

In this example, you see, that a relationship is added for the menu entry `About`. The source object is the `TestStep` with the name `GUI-Tests-42.1` (in fact the only test step in this example). It links to an object of the type `GuiMenuEntry` with the name `Help.About`. By the name of the relationship, you can guess, that the defined target type is a GuiObject and that the GuiMenuEntry is a subtype of it. This would be allowed – if you want to be sure, look into the meta model (you should do it in real life – here, I am to lazy to write it down...).

Relationship direction from target to source

Sometimes, the direction of the relationship must be just the other way round: in the situation that in your document you want to link from a target object to a source object you can add the attribute value pair `Richtung=”Quelle”` to the XML of your relationship (Richtung means direction, and Quelle means source). This means, that in the Hotspot, you have the source of the relationship.

Relationship resolution in OpenSHORE

A strength in OpenSHORE is the following: on the example above, we wrote as the target type `GuiMenuEntry`. We could also write a parent of `GuiMenuEntry` here e.g. `GuiObject`. If there is in another document a concrete object with the name `Help.About`, OpenSHORE would automatically resolve from the parent type to the concrete type with the matching name. That means, that you do not have to know which special type you are referencing, but an abstract type. If there is only one matching object, this is recognized as the right one. If there are more than one matching objects, OpenSHORE will offer you both objects for selection if you are navigating in your Browser on such a link.

Alphabetical Index