WIRTSCHAFTS UNIVERSITÄT WIEN VIENNA UNIVERSITY OF ECONOMICS AND BUSINESS

IDL-XML-Converter

A Package For Transforming IDL into XML

Seminar Paper

Lukas Schreier
0525061

0675 IS-Projektseminar SS 2010
Ao. Univ.-Prof. Dr. Rony G. Flatscher
Institute for Management Information Systems
Vienna University of Economics and Business Administration

Table of Contents

1 About.... 4
1.1Requirements.... 4
2 An introduction to IDL.... 5
3 IDL for OpenOffice.org.... 7
3.1 IDL-Types in UNO-IDL.... 8
Modules.... 8
Enumerations.... 8
Constants.... 9
Structures.... 9
Typedef.... 9
Exceptions.... 10
Singleton.... 10
Interfaces.... 10
Services.... 12
4 The Compilation Process of UNO.... 14
5 The OpenOffice-Registry.... 16
5.1 Structure of the Registry.... 16
5.2 Structure of the Binary-Array.... 18
HEAD.... 18
Constants-Pool (CP).... 21
Field.... 22
Methods.... 24
Parameters.... 25
Reference-block.... 26
5.3 Overview of the registry.... 27
6 Writing UNO-IDL-Files in XML.... 28
6.1 XML-Basics.... 28
6.2 General rules about writing correct XML files [14].... 28
6.3 Skeletal Structure of the XML-UNO-IDL-File.... 29
6.4 XML-UNO-IDL-Types.... 31
Enumerations.... 31
Constants.... 32
Structures.... 33
Typedef.... 33
Exception.... 34
Singleton.... 35
Interfaces.... 35
Services.... 39
6.5 Service Initializations / Register components in XML.... 41
7 IDL-XML-Converter-Package.... 42
7.1 Run the Programs.... 42
Setting up CLASSPATHS for Windows.... 42
Setting up CLASSPATH on UNIX variants and MacOS.... 42
7.2 Starting a program.... 43

idl2xml..... 45
xml2reg..... 45
reg2xml..... 46
regmerge..... 46
Examples..... 47
8 Round-up and Outlook..... 48
9 Appendix..... 50
10 List of Codings..... 54
11 List of Illustrations..... 55
12 List of Tables..... 55
13 Literature..... 56

1 About

The IDL-XML-Converter-Package is used to convert IDL to XML files. Those XML files can then be used to write an appropriate OpenOffice-Registry.

The package includes the following functions:

- Converting IDL-files into XML files
- Converting XML files into an OpenOffice-Registry format
- Converting OpenOffice-Registries into XML files
- Merging two OpenOffice-Registries

The content of the package consists of six Java programs, which are published under the LGPLv3-license.

- idl2xml.jar
  This program converts an existing OpenOffice.org IDL file into an XML file.
- XMLReg.jar
  Writes an XML file which contains OpenOffice IDL-types into an OpenOffice-Registry file.
- RegXML.jar
  Extracts a given OpenOffice-Registry key into an XML file.
- RegMerge.jar
  Merges two specified OpenOffice-Registries into one.

1.1 Requirements

To use this package it is necessary to have at least JAVA 1.5 and the OpenOffice 3.2. installed. For development the OpenOffice SDK 3.2 is needed as well as to make changes to the idl2xml-Interpreter JAVACC is needed.

2 An introduction to IDL

Imagine the following case: Mr. C++ wrote a method which is called dWord. This method generates a String-value and returns some value to the method-caller. Ms. Java wants to call this method to get that value. Since she is programming in Java and not in C++, she has no idea about C++ characteristics and the specific implementation of the function. As a result she cannot call this function with Java.

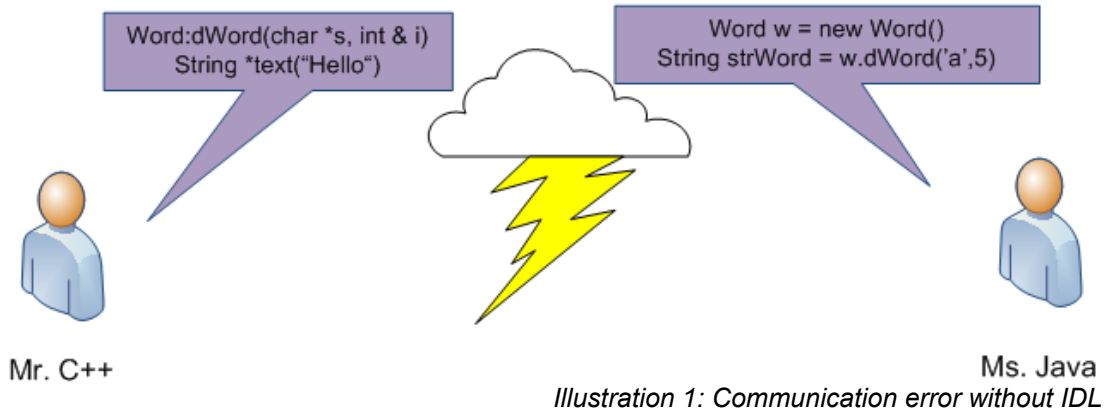


Illustration 1: Communication error without IDL

Now, a so called Interface Description Language (IDL), comes into scope. With an Interface Description Language it is possible to write language-independent sets of definitions, so that a language-depending system can understand the implementation. The Objects Management Group (OMG) defines those sets of definitions as *"...interfaces that both client and server object understand and can use easily, regardless of their platform, operating system, programming language, network connection, or other characteristics*" [1].

Let's say Mr. C++ is now using an Interface Description Language to enable Ms. Java to call his method: To do so, Mr. C++ needs to write an IDL-File which holds the method-signature of dWord in a language-independent way. After that the file needs to be compiled by an IDL-Compiler. The IDL-Compiler produces "stub"-files for Ms. Java and "skeleton" files for Mr. C++ which are used for communication with each other. [2]

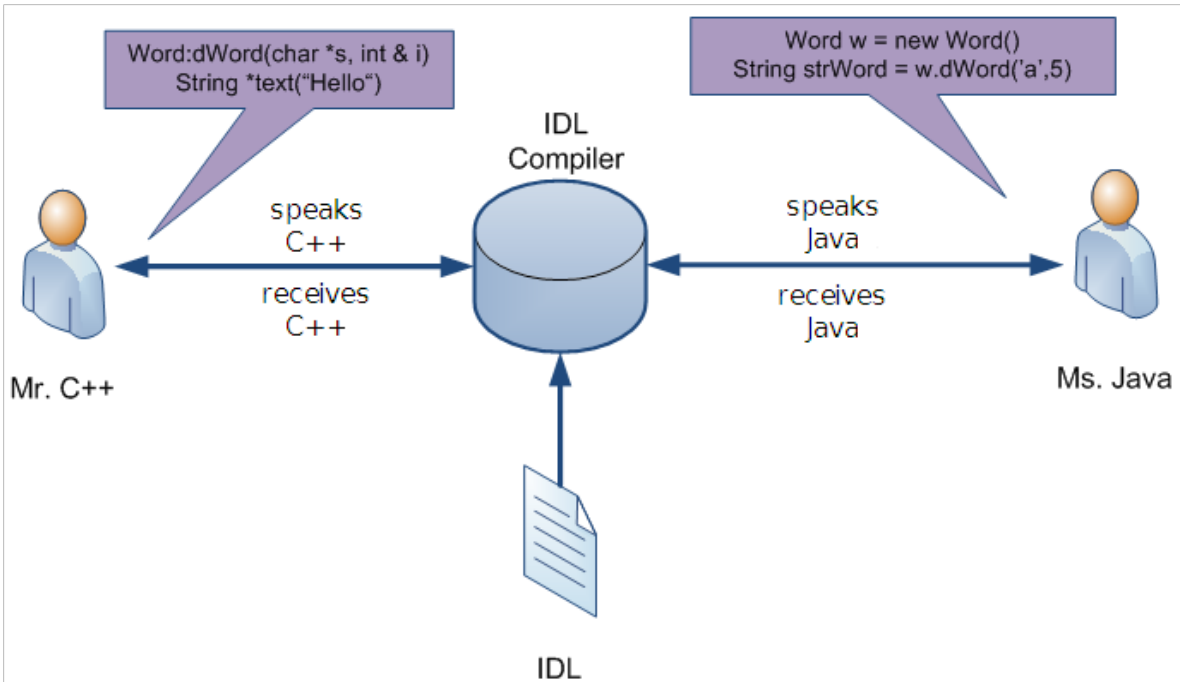


Illustration 2: Communication with IDL

A possible implementation of that method can look something like that:

```
interface myWord {
     const string text = "Hello";
     Word dWord ([in] char c, [in] int i);
};
```

Coding 1: Hello IDL

This language-independent definition of the method dWord can now be compiled by the IDL-Compiler for a language-depending method-call.

3 IDL for OpenOffice.org

Since components for OpenOffice can be written in different languages, OpenOffice needs a concept which collects all the different components and makes them available for other programming languages. The framework which makes this possible is called UNO (**U**niversal **N**etwork **O**bjects).

The UNO framework draws upon such an Interface Description Language which describes UNO components in a language-independent way. [4]

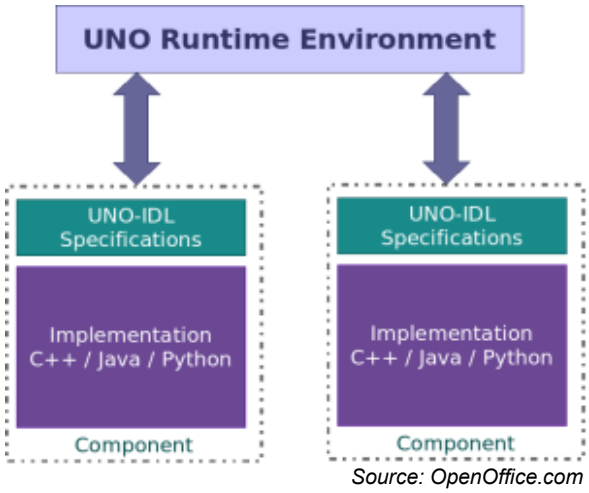
The Interface Description Language of OpenOffice is call UNO-IDL. UNOI-DL stands for Universal Network Objects Interface Description Language and is quite similar to CORBA IDL and MIDL [5]. However, there are some differences between UNO-IDL and CORBA IDL:

UNO-IDL does not support Unions and Arrays but it implements an inheritance for exceptions and structures, it is possible to set values for enumeration types and a completely new type was introduced which is called 'service' [16].

Currently the UNO framework supports the following programming languages: Java, C++ and Python. Additionally, UNO allows the control through some scripting-languages and StarBasic can access the whole UNO-API [7].

Components in UNO therefore need an implementation of the component itself and also a UNO-IDL-specification. The implementation is the language-depending code in which the component was programmed. This part of a component is never accessed by any component of the UNO-framework [6]. While the UNO-IDL-specification is the abstraction through the UNO-IDL of a UNO component which was written in a specific programming language.

Not until the implementation and specification is provided to the UNO-framework the component cannot be called by any other one in the framework.



Source: OpenOffice.com

Illustration 3: UNO-Components

3.1 IDL-Types in UNO-IDL

The following types are currently defined in UNOIDL:

- Modules
- Enumerations
- Constants
- Services
- Interfaces
- Structures
- Typedef
- Exceptions
- Singleton

Preferable each UNO-IDL type should get its own definition-file.

Modules

Modules serve as a kind of path to the UNO components. Here the location is set with "*module"* at the beginning of a UNO component definition.

	UNO Example	BNF notation

1
2
3
4
5
6
	module com {
	  module sun {
	  …
	  };
	};
	
		module-decl ::= "module" identifier "{" declaration* "}" ";"

Coding 2: Modules in UNOIDL

In the file system 'com/sun' is a directory-structure. Modules are closed with "*};*". The concept is quite similar to Java-packages or C++-namespaces. OpenOffice recommends using module-structures for every IDL-Type created [10].

Enumerations

An enumeration defines a finite amount of elements at once. The elements in an enumeration which can be accessed with their names defined.

	UNO Example	BNF notation

1
2
3
4
5
6
	enum myEnum{
	  elementOne,
	  elementTwo,
	  elementThree
	};
		enum-decl ::= "enum" identifier "{"
		  enum-member-decl ("," enum-member-decl)* "}" ";"
		
		enum-member-decl ::= identifier ["=" expr]

Coding 3: Enumerations in UNOIDL

An enumeration starts with the keyword `enum` and an *id* (here: `myEnum`), followed by the names of the elements in the enumeration separated with a comma ('`,`'). Additionally, it is possible to add a value to the elements. If no value is defined, the value of each element will be raised incrementally starting with zero.

Constants

Constants are fixed values. There a two different kinds of definitions. First, single constants are defined in a module and are initialized with the key word '`const`'. Second, a set of constants as an own UNO-IDL type. A set of constants as a type starts with the keyword '`constants`'.

	UNO Example	BNF notation
1	`module com {`	`constants-decl ::= "constants" identifier`
2	`  module sun {`	`                    "{" const-decl* "}" ";"`
3	`    const long myConst = 100;`	
4	`    const float myFloat = 100.4;`	`const-decl ::= "const" type identifier "=" expr ";"`
5		
6	`    module star {`	
7	`      constants aSetOfConstants {`	
8	`        const long ONE = 1;`	
9	`        const long TWO = 2;`	
10	`        const float THREE = 3.1;`	
11	`};`	
12	`};};};`	
13		
14		
15		

Coding 4: Constants in UNOIDL

Structures

With structures it is possible to store a specific amount of variables in one construction. Therefore a structure is a kind of data container. The name of a variable in a specific structure must be unique. A structure is initialized with the key word '`struct`' followed by its members:

	UNO Example	BNF notation
1	`struct myStruct {`	`struct-decl ::= "struct" identifier [single-inheritance]`
2	`        long varOne;`	`                 "{" struct-member-decl+ "}" ";"`
3	`        float varTwo;`	
4	`        string varThree;`	`struct-member-decl ::= type identifier ";"`
5	`};`	
6		`struct-params ::= "<" identifier ("," identifier)* ">"`
7		

Coding 5: Structures in UNOIDL

In a structure a variable has no value. Furthermore every declaration ends with a semicolon '`;`'. Structure definitions with just one member are wrong defined.

Typedef

A Typedef sets an alternative name to an already existing name. OpenOffice.org recommends not to use Typedefs or, if necessary, carefully [17].

	UNO Example	BNF notation
1	`typedef byte myByte;`	`typedef-decl ::= "typedef" type identifier ";"`
2		

Coding 6: Typedef in UNOIDL

Exceptions

An exception describes a situation in a program which can throw an error. A defined exception interrupts the program in an error situation for a special handling. In OpenOffice.org exceptions are the general error handling concept [18].

	UNO Example	BNF notation

1
2
3
4
	exception ErrorOccoured {
	  long errorNum;
	};
		exception-decl ::= "exception" identifier [single-inheritance]
		"{" struct-member-decl* "}" ";"

Coding 7: Exceptions in UNOIDL

Singleton

A Singleton can be seen as an alias for UNO-services. When a singleton is defined the specific component can be instantiated only once.

	UNO Example	BNF notation

1
2
3
4
	singleton myAlias{
	  service myService;
	};
		singleton ::= "singleton" identifier "{" "service" name ";" "}"
		";"

Coding 8: Singleton in UNOIDL

In this example the UNO-component '`myService`' gets the alias '`myAlias`'.

Interfaces

With interfaces in UNO-IDL it is possible to describe attributes and methods of an UNO-component in a language-independent way. Every interface in OpenOffice needs to inherit from another interface. On writing a new interface and no inheritance is possible, the interface must be derived from the base-interface `com/sun/star/uno/XInterface`. In general the UNO-IDL allows a single-inheritance only. The inheritances in the UNO-IDL are always the fully qualified name of the derived interface. The fully qualified name holds the whole package-structure to the interface separated with '`::`'.

According to the convention by OpenOffice interfaces should always start with an 'X' in the interface-name.

The general form of an interface-declaration in UNO-IDL:

	UNO Example	BNF notation for an interface header

1
2
3
4
5
	interface myInterface :
	  com::sun::star::lang::XInterface {
	  …….
	};
		interface-decl ::= "interface" identifier [single-inheritance]
		“{" interface-member-decl* "}" ";

Coding 9: Interfaces in UNOIDL

Attributes in interfaces

The attributes in interfaces store values in a specific data type. This type can be a primitive data type like '`char`' or '`byte`', but as well as complex type. When the type is a complex type the fully qualified name must be declared, separated with the scope-operator '`::`'.

	UNO Example	BNF notation for attributes in interfaces
1	`interface myInterface :`	`attribute-decl ::= attribute-flags type identifier";"`
2	`  com::sun::star::lang::XInterface {`	
3		`attribute-flags ::=`
4	`  [readonly,attribute] long attr1`	`"[" (attr-flag ",")* "attribute" "," attr-flag)* "]"`
5	`  [attribute] com::sun::complex attr2`	
6		`attr-flag ::= "bound" | "readonly"`
7	`};`	
8		

Coding 10: Interface with Attributes in UNOIDL

This example shows two attributes '`attr1`' and '`attr2`'. The first one is an attribute which can only be read and not written. This is marked by an additional attribute flag in the squared parenthesis. For all possible attribute flags see the appendix. The second attribute '`attr2`' is defined through the complex type '`com/sun/complex`' and is separated with the scope-operator.

Methods in interfaces

Interfaces can also contain methods. If a method is defined in an interface, the signature of the method is added into the interface-declaration. This includes the name of the method, the return type, the parameter-list and, if necessary, an exception-handling.

The parameter-list of a method is quite similar to the definition of attributes in interfaces. However, the attribute flag will be changed with a parameter mode option. Every parameter in the list needs to get set with one of the three parameter mode options:

- `in` – defines the parameter as input-parameter
- `out` – defines the parameter as an output-parameter
- `inout` – specifies the parameter as both input- and output-parameter.

An exception is added to the method with the keyword '`raises`' and the fully qualified name of the specific exception. On adding more than one exception, they get separated with a comma ('`,`').

	UNO Example	BNF notation for attributes in interfaces
1	`interface myInterface :`	`method-decl ::= ["[" "oneway" "]"] type identifier`
2	`  com::sun::star::lang::XInterface {`	`"(" [method-param ("," method-param)*] ")"`
3		`[exception-spec]`
4	`  long myMethod(`	
5	`  [in] byte param1,`	`method-param ::= "[" direction "]" type identifier`
6	`  [out] long param2`	
7	`  )`	`direction ::= "in" | "out" | "inout"`
8	`  raises(com::sun::star:lang:IllegalArg)`	
9		`exception-spec ::= "raises" "(" name ("," name)* ")"`
10	`};`	
11		

Coding 11: Interface with Method in UNOIDL

Services

"*The specification of an interface or service is abstract, that is, it does not define how objects supporting a certain functionality do this internally. Through the abstract specification of the OpenOffice.org API, it is possible to pull the implementation out from under the API and install a different implementation if required.*" [11]. If a service is defined it will get implemented into the object`s service manager under the name specified in the UNO-IDL-specification. Every time an implementation changes its class names or implementations, the service name is always the same and the service manager decides which implementation should be started.

There are two different kinds of services in OpenOffice. First, "old-styled" service and second, the "new-styled" service which was introduced with OpenOffice2.0. The new-styled service inherits one single interface while the old-styled service can contain further services, more than one interface-inheritances and properties which are attribute-definitions similar to interface-attributes.

New-styled service / interface service

	UNO Example	BNF notation for a new-styled service
1		`interface-service-decl ::= "service" identifier ":" name`
2	`service UnoUrlResolver :`	`["{" constructor-decl* "}"] ";"`
3	`XunoUrlResolver;`	
4		
5		`constructor-decl ::= identifier "(" [constructor-params] ")"`
6		`[exception-spec]`
7		
8		
9		`constructor-params ::= rest-param`
10		`| ctor-param ("," ctor-param)*`
11		
12		
13		`rest-param ::= "[" "in" "]" "any" "..." identifier`
14		
15		
16		`ctor-param ::= "[" "in" "]" type identifier`
17		

Coding 12: Interface Service in UNOIDL

Old-styled service / accumulated service

	UNO Example	BNF notation for a old-styled service
1	`module com {`	`accumulated-service-decl ::= "service" identifier ":" name`
2	`module sun {`	`"{" service-member-decl+ "}"`
3	`module star {`	`";"`
4		
5	`module xsd {`	`service-member-decl ::= service-inheritance-decl`
6	`service Year {`	`| interface-inheritance-decl`
7		`| property-decl`
8	`interface XdataType;`	
9		`service-inheritance-decl ::= ["[" "optional" "]"] "service"`
10	`[property, maybevoid]`	`name ";"`
11	`short MaxInclusiveInt`	
12	`};`	`property-decl ::= property-flags type identifier ";"`
13		
14	`};`	`property-flags ::= "[" (property-flag ",")* "property" ","`
15	`};`	`property-flag)* "]"`
16	`};`	
17	`}`	`property-flag ::= "bound" | "constrained" | "maybeambiguous"`
18		`| "maybedefault" | "maybevoid" | "optional" | "readonly" |`
19		`"removable" | "transient"`
20		

Coding 13: Accumulated Service in UNOIDL

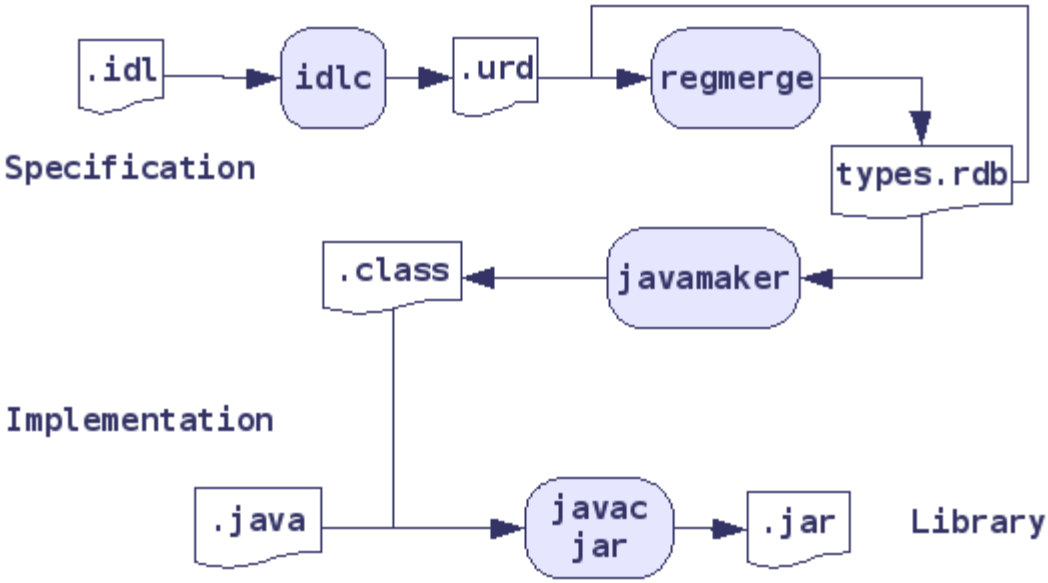
This examples shows, that attributes in a service are called properties and marked with the flag '`property`' in squared parentheses.

Further readings

For further information about writing IDL files see the official OpenOffice documentation about UNOIDL [19]. There is also a good tutorial about IDL by IBM. This tutorial is actually about CORBA-IDL but since CORBA-IDL and UNO-IDL is quite similar it can be quite useful [20].

4 The Compilation Process of UNO

After both implementation and specification was written, the component needs to be registered in UNO. Fortunately, there are some tools which register components in OpenOffice. The next illustrations shows a typical compilation process for registering UNO components in Java.



Source: OpenOffice.org

Illustration 4: Compilation Chain

First the UNO-IDL-specification file is loaded into idlc which compiles the specification-file into a registry file. After a compilation with idlc an `*.urd` registry-file is established. This `*.urd` file can be seen as a temporary type-registry file with all value-pairs of an IDL type. If more then one component is compiled with idlc the `*.urd` files needs to get merged (`regmerge`) into a registry-database. This registry-databases ends with the suffix `*.rdb`.

In the next step the IDL type has to be translated, so that other languages can call the members of the component. For the translation, which is actually called language-binding, all types and references of a component which are used in the implementation need to be in this `*.rdb` registry-database. OpenOffice has a registry-database for all components used at runtime. This file is called '`types.rdb`'. [8]

The above shown illustration binds a component written in Java with `javamaker`. If the component is in C++, then the `cppumaker` is taken[3]. `Javamaker`/`cppumaker` produces some additional files which are used for the language binding. In case `javamaker` was used in the

compilation process a `*.class` file will be created. This `*.class` file needs to be packed together with the implementation files into a `*.jar`-archive. This `*.jar`-archive is then copied to `<OpenOfficePath>/programs/classes`.
The language-binding process ends with that step.

To make the component available in the OpenOffice Runtime Environment some more steps are necessary. The newly generated `*.rdb` file must now be loaded against `regcomp`, which makes the registration of the components in the `*.rdb` file to OpenOffice. Finally the registration file needs to be set in a file called `uno.ini` on Windows or `unorc` on Linux/Unix-Systems. [8] For more information about the registration process of an OpenOffice component see the Developer's Guide, and here in particular section 4.2.2 Generating Source Code from UNO-IDL Definitions.

5 The OpenOffice-Registry

The OpenOffice-Registry, or in the following short registry, is an ordered binary-array which holds the compiled `*.idl` files of OpenOffice [3]. The registry is used to invoke the registered components from OOoBasic or other supported programming languages of OpenOffice. By using this kind of registry it's possible that any bridge to a programming language can use this kind of information [4].

One binary-array in a registry holds all the values of one single key. Therefore each key in the registry has its own binary-array. Here the key-name can be seen as the name of the compiled IDL type.

5.1 Structure of the Registry

The keys are ordered hierarchically in the registry [4]. While the hierarchy depends on the 'module'-structure set in the `*.idl`-file.

An example:

***.idl file**	***.urd/*.rdb file**

`module com {`
`  module sun {`
`  module star {`
`  module test {`
`  struct myStruct {`
`  long varOne;`
`  float varTwo;`
`  string varThree;`
`  };`
`};};};};`
	`/`
	`  /com`
	`  /sun`
	`  /star`
	`  /test`
	`  /myStruct`
	`  field 0: varOne`
	`  field 1: varTwo`
	`  field 2: varThree`

Coding 14: Registry Structure

The next example extends the previous one through adding a new interface '`myInterface`' with the same module-structure:

***.idl file**	***.urd/*.rdb file**

`interface myInterface :`
`  com::sun::star::lang::XInterface {`
 	
`  long myMethod(`
`  [in] byte param1,`
`  [out] long param2`
`  )`
`  raises(com::sun::star:lang:IllegalArg)`
 	
`};`
	`/`
	`  /com`
	`  /sun`
	`  /star`
	`  /test`
	`  /myStruct`
	`  field 0: varOne`
	`  field 1: varTwo`
	`  field 2: varThree`
	`  /myInterface`
	`  method 0: myMethod`
	`  param 0: param1`
	`  param 1: param2`
	`  raises: IllegalArg`

Coding 15: Registry Structure after a Second Key

As this example shows the registry hierarchy remains the same as adding a new IDL type when the IDL type has the same module-structure. Otherwise a new hierarchy will be appended to the registry.

Furthermore, every key itself holds value-pairs which describe the properties of a specific key, or in more detail, the value-pairs are the fragmented definition-sets of an `*.idl`-file. In general the value-pairs are in the form of '`value-name : value`'.

A key can be divided into two main blocks: a head- and a body-block, while the body can contain three different types of registry entries:

- Fields
- Methods
- References

Head

The head of a registry-key holds general information about the whole key.

Fields

Fields contain the attributes of the following IDL-Types: structures, constants, exceptions, enumerations, attributes of services and interfaces.

Methods

A method is written when the IDL type is an interface and the functions of this certain interface can be called.

References

A reference is set in the registry when a service imports an interface.

5.2 Structure of the Binary-Array

Since the OpenOffice-Registry is stored in a binary format and therefore not human-readable, the bytes in the array must have a special order to extract the values with an algorithm which presents the data in a human-readable form. Depending on the IDL type the registry array can be extended with blocks for fields, methods and references (see Structure of the Registry).
Each block in the array is subdivided into fields which represents a single information and those fields have a special size measured in bytes. However, there are only two different sizes for the fields in general – 2 bytes or 4 bytes. That means for example, if a field has 2 bytes, the field holds two values between -127 and 127. When the field has 4 bytes it holds four values between -127 and 127.
An exception to this rule is the constants-pool, which has a variable length, however, this will be described later in this chapter.

Because the fields store the information in bytes, the value needs to be calculated into the decimal format to get the necessary information from a specific field. An example: Let's say a field has the size of 2 bytes with the values `[1, 23]`. The first byte has the value `1` and the second byte is set to `23`. Converted to the decimal format the field has the value '`279`'. Depending on the field, the value can have two different meanings. First, it is a 'direct'-value, which means, that the value is directly stored in the field. This could be, for example, the amount of fields in the registry for a specific key. Or, second, the value points to a particular position in the binary of a key. Then the value is an index. In general the index-values point into the constants-pool where all the names and values of the fields, methods and references are stored.

The binary-array in its basic form always has the same structure. It starts with the HEAD and is then extended with field-, method- and reference-blocks, depending on the IDL type.
In the following all blocks and its fields will be explained in detail.

HEAD

Above is a list of all fields of the HEAD-block. For a graphical view see section Overview of the registry.

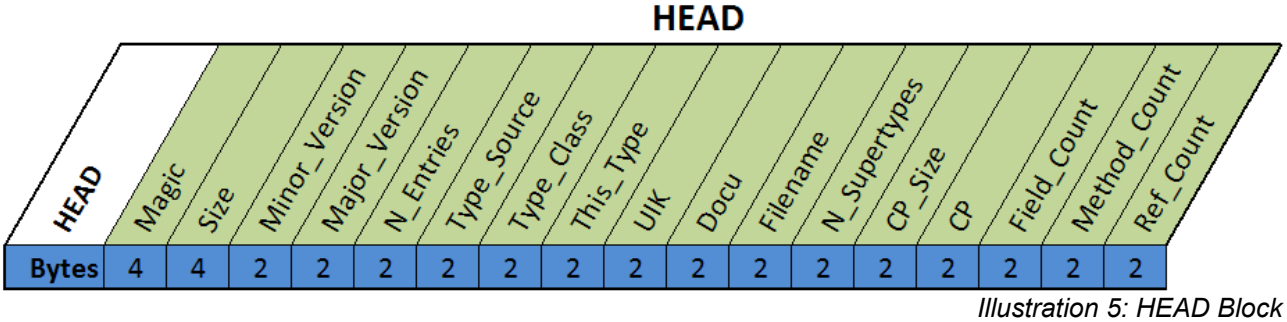


Illustration 5: HEAD Block

Field Name	Direct/Index	Description
Magic	Direct	The ’magic’ marks the start of an array of values in a registry for a specific key. The ‘magic’ has always the hex-value of ‘12345678’. This value is for internal use only.
Size	Direct	The size describes the length of a key's array. This includes the whole head, the constants-pool and, if any, the fields, the methods and references. The size is the sum of all bytes written into the registry.
Minor_ Version	Direct	This field should stand for a version control mechanism of the IDL-files. But currently OpenOffice doesn't version control [9]. Therefore the value of Minor_Version is set to ‘0’ (zero) by default.
Major_ Version	Direct	This field should stand for a version control mechanism of the IDL-files. But currently OpenOffice doesn't version control [9]. Therefore the value of Major_Version is set to ‘1’ by default.
N_Entries	Direct	This field stands for the amount of entries in the head of a key. OpenOffice writes by default the value ‘6’.
Type_ Source	Direct	This field holds the name of the language name in which the component was written. This could be 'UNO-IDL', 'CORBA-IDL' or 'Java'. Because 'UNO-IDL' is only supported the current value is by default an empty value.
Type_ Class	Direct	This entry describes the IDL type. For example number '2' stands for 'module' or '7' marks the IDL type as a service. For the enumeration list of the IDL types see the Appendix.
This_Type	Index	The entry 'This_Type' holds the fully qualified name of the IDL type. This means, that the package path is listed in front of the name. For example 'com/sun/star/test/myService'. While 'com/sun/star/test' is the path of the package and 'myService' is the name of the IDL type.
UIK	Direct	UIK stands for a unique identifier. Formally the UIK was used to identify UNO interfaces but this field is deprecated. This field now has the value ‘0’ (zero) by default.
Docu	Index	This field holds a general documentation about the IDL type. The documentation should be written in a JavaDoc-style.
Filename	Index	The filename entry holds the name of the IDL file, typically ending with a *.idl-suffix.
N_ Supertypes	Direct	This field holds the amount of supertypes of a specific key. This field is only set when the IDL-Type is an exception, an interface, a

		service or a structure because this IDL-Types can inherit from other IDL-Types. In short, the supertypes holds the inheritances of an IDL-Type.
`Supertype`	Index	Holds the name of the supertype. For each supertype, a new 2 byte field is appended. This means, that the standard structure of a registry array is extended by 2 bytes for every supertype. If N_Supertypes equals zero, no supertype field is added.
`CP_Size`	Direct	CP_Size is a 2 byte field which holds the overall amount of strings and values used in a specific key.
`CP`	Direct	If CP_Size is greater than zero, then CP-blocks are added for each string or value in a key. For further information see section CP-block in this chapter.
`Field_ Count`	Direct	The Field_Count holds the amount of fields stored in the array.
`Method_ Count`	Direct	Holds the amount of methods in the array.
`Reference_ Count`	Direct	This field stores the amount of reference when the IDL-Type is a service.

Table 1: HEAD Block of a Registry's Binary Array

Supertypes

As mentioned if `N_Supertypes` is greater than zero Supertype fields will be added to the HEAD which extends the HEAD with N_Supertypes-fields. In the next example two supertypes should be added:

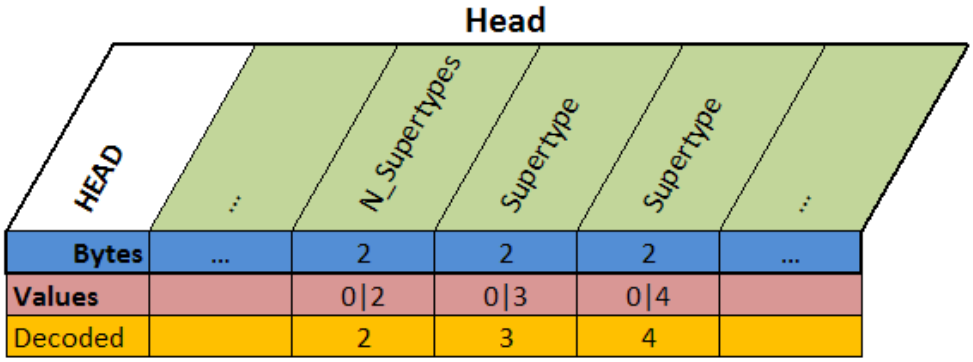


Illustration 6: Two Supertypes Added to the HEAD

The field `N_Supertypes` has now the value [0,2] which means decoded '2' and two `Supertype` fields were added. The first `Supertype` field is pointing to the third CP-block which contains the name of the supertype. The second `Supertype` field holds the index-value 4 which means that the name of the supertype is in CP-block 4.

Constants-Pool (CP)

The CP-block has this general form:

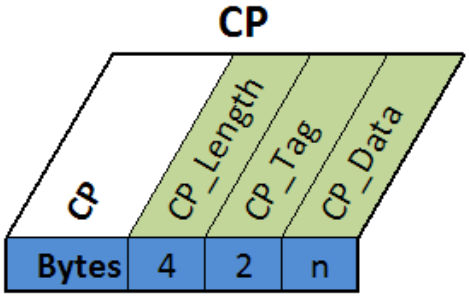
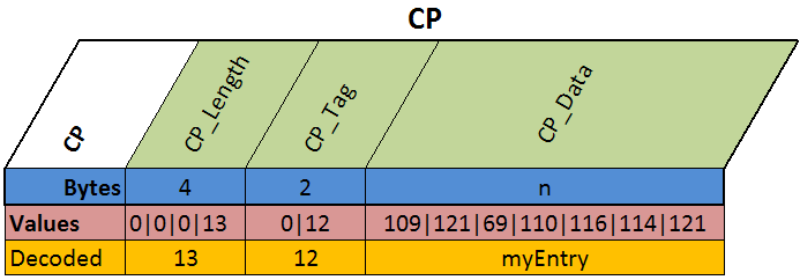


Illustration 7: Constants-Pool Block

Field Name	Direct/Index	Description
CP_Length	Direct	Length is the size measured in bytes of a single CP-block. This includes the 4-bytes of the CP_Length, 2-bytes for the CP_Type, and n-bytes for the content. The CP_Length is stored directly in the field.
CP_Tag	Direct	Currently there are three CP-tags only: String values: '12' Long value: '5' Single value: '3' This also means, that other types like float- or hyper-types are not supported yet. This issue is based on the architecture of the registry itself. A hyper value is bigger than a long value and a float value cannot be mapped to long-types. Therefore OpenOffice writes instead of the float-value just 'float' into the value field of the registry. The CP_Type is stored directly in this field.
CP_Data	Direct	Depending on the CP_Type the byte-values in the CP_Data fields have different meanings. When the CP_Type is '12' – that means a String value – each byte value stands for a letter in the ASCII code. When the CP_Type is a number, then the byte values represents the number stored in the registry. The CP_Data is stored directly as well.

Table 2: CP Block of a Registry's Binary Array

This example shows a CP-block which holds the String '`myEntry`'.



This example shows the `CP_Length` which is a 4 byte-binary-array with the value `[0,0,0,13]`, the `CP_Tag` which is a 2 byte-binary-array with the value `[0,12]` and followed by the `CP_Data` itself which holds the encoded ASCII-values. This abstract description of a CP-block means that the CP-block has a length of 13 bytes, the `CP_Data` is of type String and the entry holds in this block is '`myEntry`' (without the quotation marks).

Field

If the `field_count` is greater than zero then this block is added to the array. Constants, enumerations, exceptions, interfaces, services and structures could extend the array with the field-block. After the field_count entry an offset mark follows, which is set once and has currently the value '6'.

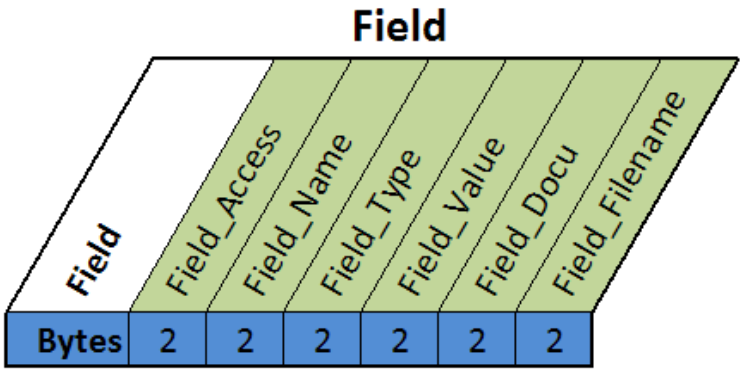


Illustration 9: Field Block

Field Name	Direct/Index	Description
Field_ Access	Direct	Sets the kind of accessibility for the field. For example: `readonly` or `bound`.
Field_Name	Index	This field holds the pointer to the name in the constants-pool.
Field_ Type	Direct	Specifies the type of the field which is stored as string in the constant-pool. The type could be a primitive type like 'char' or 'long' or a complex type which is hold with its fully qualified name.
Field_ Value	Index	If a value was specified, the field holds the pointer to the constant-pool where the value is stored. If no value was specified the field holds a zero.
Field_Docu	Index	Holds the documentation of the field as index for the constant-pool.
Field_ Filename	Index	Sets the filename where the value comes from.

Table 3: Field Block of a Registry's Binary Array

Methods

Only when an interface has methods, then the method-block is added for providing information about a specific interface-method. Since methods can hold parameter, it is possible, that an additional block about parameters is added to the method-block. Furthermore, if a method throws an exception, an exception-block will be added as well. The internal offset for the method-block is by default '5' and the offset for parameters is '3'.

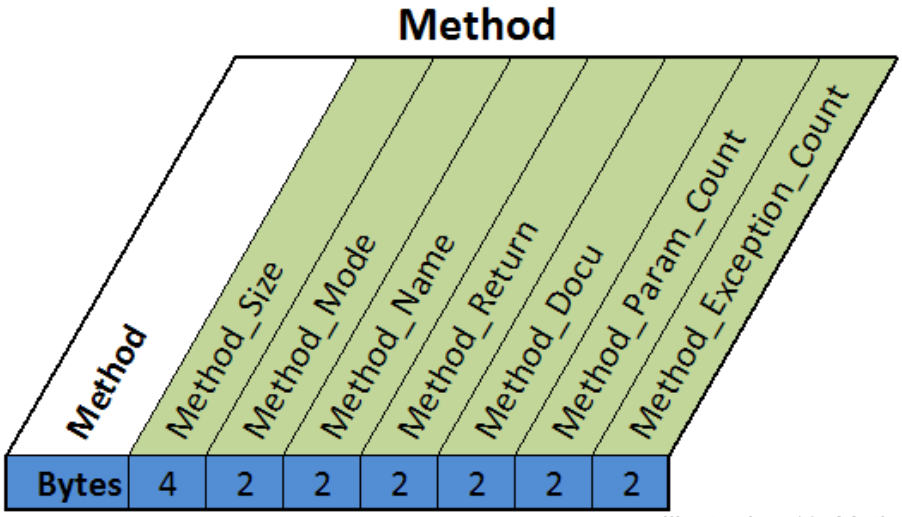


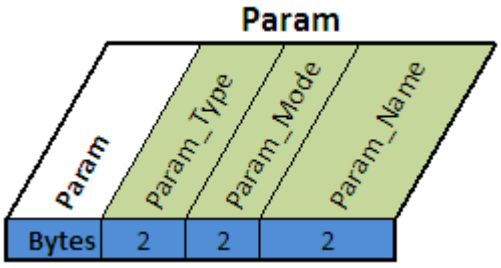
Illustration 10: Method Block

Field Name	Direct/Index	Description
Method_ Size	Direct	The method_size holds the size of the method measured in bytes. This includes the whole method-block, the Param-block and Exception-block.
Method_ Mode	Direct	Specifies the access mode of the method. This can be for example `oneway` or `synchronous`.
Method_ Name	Index	Holds the index-value for the constant-pool. This string in the constants-pool holds the name of the current method.
Method_ Return	Index	Specifies the index in the constants-pool which holds the fully qualified name of the type which is return by the method.
Method_ Docu	Index	This field holds the documentation for the current method. The documentation in this field is an index-value which points to the constant-pool-block where the content is stored.
Method_ Param_Count	Direct	Holds the amount of parameters defined for this method.
Method_ Exception_ Count	Direct	The field Method_Exception_Count stores the amount of exceptions which could be thrown be the current method
Method_ Exception_ Name	Index	If Method_Exception_Count is greater zero, then the same amount of Method_Exception_Name-fields as the Method_Exception_Count-field holds will be added. The name of the exception is an index-value which points to the specific constant-pool-block where the name is stored.

Table 4: Method Block of a Registry's Binary Array

Parameters

The Param-block is added when the current method has parameters defined.



Field Name	Direct/Index	Description
Param_Type	Index	This field specifies the type of the current parameter. This could be whether a primitive data type like 'char' or 'byte' or a complex type. If the param_type field holds a complex type the fully qualified name is stored in the constants-pool.
Param_Name	Index	Param_Name holds an index-value where the name of the current parameter is stored in the constants-pool.
Param_Mode	Direct	The param_mode specifies the direction from which the parameter is passed. This could be `in`, `out`, `inout`.

Table 5: Parameter Block of a Registry's Binary Array

Reference-block

The reference-block will be added when the IDL-Type is a service and this service holds interfaces or other services. Again, if `reference_count` is greater than zero, an additional offset entry is added. This offset has by default the value '4'.

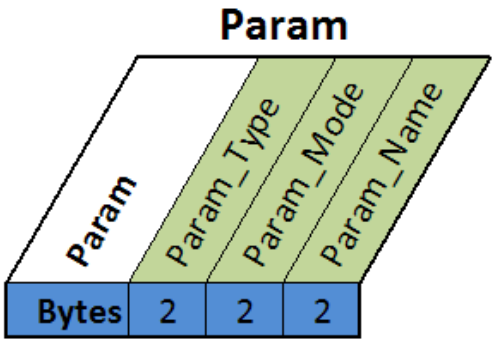


Illustration 12: Reference Block

Field Name	Direct/Index	Description
Ref_Flag	Direct	The Ref_Flag specifies if the reference is optional or invalid.
Ref_Name	Index	Specifies the fully qualified name as an index in the constants-pool.
Ref_Docu	Index	This field holds the index-value where the documentation of the this reference is stored.
Ref_Type	Direct	Holds the value about the accessibility of the reference. This could be `exports`, `needs` or `supports`.

Table 6: Reference Block of a Registry's Binary Array

5.3 Overview of the registry

Overview about the whole Structure of the binary-array for a single key in the OpenOffice-Registry

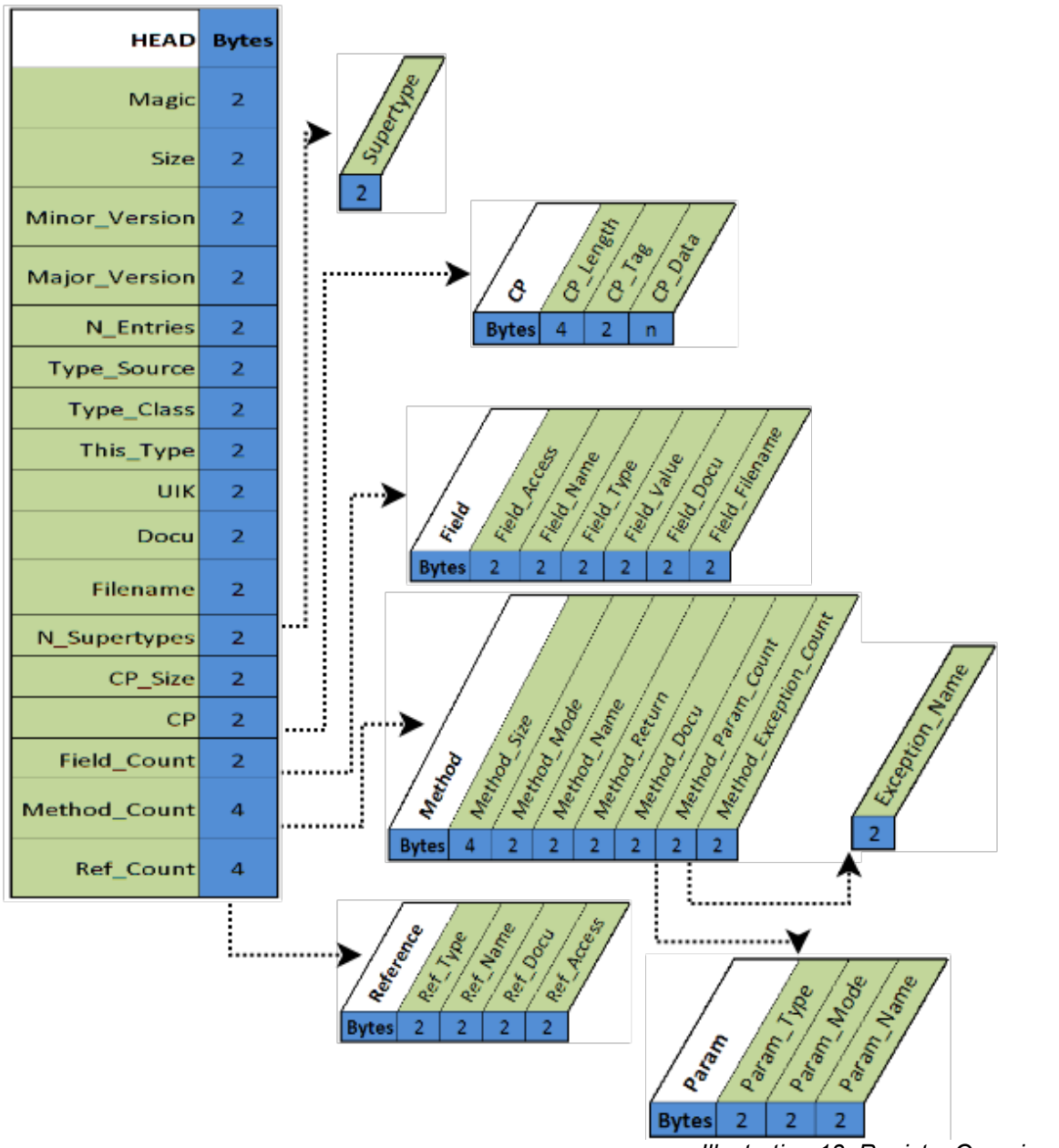


Illustration 13: Registry Overview

6 Writing UNO-IDL-Files in XML

This section will be an introduction about how to write a UNO-IDL-File within an XML-format. Basically, the structure and keywords/tags of an XML-UNO-IDL-file is derived from the same abstract syntax specification as OpenOffice defines the UNO-IDL in the Developer's Guide [12].
First, this chapter gives an short introduction about how to write XML files in general. Then follows a section about the skeletal structure of the XML-UNO-IDL-file and lastly some examples of each UNO-IDL type will be mentioned.

6.1 XML-Basics

XML stands for eXtensible Markup Language and was designed to describe data as structured information, whereby each information is tagged with a markup in the XML-document. [13]
Let's say there is contact information about Chuck Norris:

Chuck Norris, Actor
Milkyway 23
Ryan, Oklahoma
chuck@noris.com

The contact can be translated into an XML-format to structure the information like this:

```
<person>
  <name>Chuck Norris</name>
  <email>chuck@norris.com</email>
  <profession>actor</profession>
  <address>Milkyway 23</address>
  <city>Ryan</city>
  <state>Oklahoma</state>
</person>
```

Coding 16: Hello XML

This examples shows that each information of the business-card was tagged with markups to specify the “meaning” of certain information.

6.2 General rules about writing correct XML files [14]

- Each starting tag must have a closing tag

  That means that every information in the XML file must be surrounded with a start-tag `<myTag>` and an end-tag like `</myTag>`. A closing tag always has a '`/`' between the less-symbol and the tag name itself
- The tags are case sensitive

The tag <myTag> and <MyTag> are two different tags

- Nesting of the XML-Tag is possible
- Every XML document has a root-element

  As shown in the previous example the root-element of the business-card was <person>. All XML files need such a root element.

There are more rules about writing correct XML files and since this is just a small introduction its recommended to read through the XML-tutorial at w3schools.com [14].

6.3 Skeletal Structure of the XML-UNO-IDL-File

Before every XML-UNO-IDL-file starts a header needs to be set. This header includes a link to a DTD-schema which checks the correctness of the XML-UNO-IDL-file.

The skeletal structure of the XML-UNO-IDL-file starts with the root-element called <xmlidl>. After the root element one or more UNO-IDL-types can follow. The markup, that an IDL-type starts is <idl_object>, while the name of the type is set in the tag as an attribute: <idl_object name='idl_object_name'>. If a package-structure should be defined, then this can be done with the tag <module>. The tag module has the same meaning in the XML-UNO-IDL as in UNO-IDL – which is the keyword module. The module-tag nests the whole content of the idl_object.

If an UNO-IDL-Type refers to other UNO-IDL-types, the specific referred UNO-IDL-type needs to get imported with the <import>-tag. The import-tag is the equivalent of the preprocessor-command #include in the UNO-IDL specification.

Then the content of the UNO-IDL-type follows. Each UNO-IDL-type has its own tag-name and its own content-definition. Both will be discussed in the next section.

The next example shows the skeleton of an XML-UNO-IDL-file:

	UNO IDL - skeleton	XML-UNO-IDL
1	#ifndef __com_test_IOException_idl__	<?xml version="1.0" encoding="UTF-8"?>
2	#include <com/test/IOException.idl>	<!DOCTYPE xmlidl SYSTEM "idlToXML_schema.dtd">
3	#endif	
4		<xmlidl>
5	module com {	<idl_object name="UNO-XML-skeleton">
6	module test {	<import>com/test/IOException.xml</import>
7		<module name="com">
8	//An IDLType	<module name="test">
9		//An IDLType
10	};	</module>
11	};	</module>
12		</idl_object>
13		</xmlidl>
14		

Coding 17: XML-UNO-IDL-Skeleton

The XML file starts with the root-element <xmldil> and is followed by the start of an UNO-IDL-type with <idl_object name='UNO-XML-skeleton'>. This name is usually the name of the

UNO-IDL-file itself. After the idl_object-tag an import tag will follow, if necessary. In this example, the IDL-type has one import. The path to the referred IDL-type is mentioned with the `#include`-commend in UNO-IDL while XML-UNO-IDL marks it with an `<import>`. This IDL-type has the package-structure com/sun/star/test which are stated with the 'module'-keyword in UNO-IDL and marked up in XML-UNO-IDL with `<module>`. The module-tag in XML-UNO-IDL is as nested as in UNO-IDL. The header is marked with a `<!DOCTYPE>`–tag and holds the name of the root-element `xmlidl` and the link to the DTD-schema `idlToXML_schema.dtd`.
After the modules were defined, the content of a certain IDL-type follows. All UNO-IDL-types are discussed in the next section.

6.4 XML-UNO-IDL-Types

All the currently defined UNO-IDL-types are available in the XML-UNO-IDL as well:

- Enumerations
- Constants
- Services
- Interfaces
- Structures
- Typedef
- Exceptions
- Singleton

The modules are not mentioned here because those are part of the skeletal structure of the XML-UNO-IDL-file. For an explanation of the UNO-IDL-types see section IDL for OpenOffice. The following examples show UNO-IDL-specifications and these examples will be converted into an XML-format to see the similarities and differences between UNO-IDL and XML-UNO-IDL. The outcome of this section is that there are more similarities than differences and those differences are due to a better legibility of the XML file.

Enumerations

	UNO IDL	XML-UNO-IDL
1		<?xml version="1.0" encoding="UTF-8"?>
2		<!DOCTYPE xmlidl SYSTEM "idlToXML_schema.dtd">
3		
4	module com {	<xmlidl>
5	module sun {	<idl_object name="Timing">
6	module star {	<module name="com">
7	module animations {	<module name="sun">
8		<module name="star">
9	enum Timing {	<module name="animations">
10		<enumeration isPublished="false">
11	INDEFINITE,	<id>Timing</id>
12	MEDIA	<enum_content>
13		<enum>
14	};	<id>INDEFINITE</id>
15	};	<value></value>
16	};	</enum>
16	};	<enum>
17	};	<id>MEDIA</id>
18		<value></value>
19		</enum>
20		</enum_content>
21		</enumeration>
22		</module>
23		</module>
24		</module>
25		</module>
26		</idl_object>
27		</xmlidl>
28		

Coding 18: Enumerations in XML-UNO-IDL

The enumeration example shows the UNO-IDL-file of `com/sun/star/animations/Timing.idl` which was converted into an XML-format. After the

standard skeleton of the XML-UNO-IDL an enumeration type is defined with `<enumeration>`. The next child-element of an enumeration-tag is an `<id>` which specifies the name of the enumeration. In this case the name is 'Timing'. After that the content of the enumeration is set with `<enum_content>` which is the equivalent to the '`{`' in line 6. Each element in the enumeration starts with the markup `<enum>`-tag which has the child-elements `<id>` for the name of the enumeration-element and a `<value>`-tag which sets the value of the element. In this example for both no value was specified. That means the `<value>`-tag is empty.
Due to the rules of correct XML all opened tags also need closing-tags - this extends the XML file.

Constants

	UNO IDL	XML-UNO-IDL
1		`<?xml version="1.0" encoding="UTF-8"?>`
2		`<!DOCTYPE xmlidl SYSTEM "idlToXML_schema.dtd">`
3		
4	`module com {`	`<xmlidl>`
5	`  module sun {`	`  <idl_object name="RelOrientation">`
6	`  module star {`	`  <module name="com">`
7	`  module text {`	`  <module name="sun">`
8		`  <module name="star">`
9	`  published constants RelOrientation{`	`  <module name="text">`
10	`  const short FRAME = 0;`	`  <constants isPublished="true">`
11	`  const short PRINT_AREA = 1;`	`  <id>RelOrientation</id>`
12	`  };`	`  <constants_content>`
13	`  };`	`  <const>`
14	`  };`	`  <type>short</type>`
15	`  };`	`  <id>FRAME</id>`
16	`  };`	`  <value>0</value>`
17		`  </const>`
18		`  <const>`
19		`  <type>short</type>`
20		`  <id>PRINT_AREA</id>`
21		`  <value>1</value>`
22		`  </const>`
23		`  </constants_content>`
24		`  </constants>`
25		`  </module>`
26		`  </module>`
27		`  </module>`
28		`  </module>`
29		`  </idl_object>`
30		`</xmlidl>`

Coding 19: Constants in XML-UNO-IDL

This shortened example shows the UNO-IDL-type `com/sun/star/text/RelOrientation`, which is of type constants. Therefore the type is marked with the tag `<constants>`. In addition, this type is published, so the `isPublished` attribute is set to '`true`'. The name of the constants-field is `RelOrientation` which was marked with an `<id>` tag right under the `<constants>`-tag. The content of the constants-field starts with `<constants_content>`, where every single constant can be defined. In the UNO-IDL-file each constant is specified with the keyword const. In XML-UNO-IDL the definition of an constant start with the tag `<const>` as well. The `<const>`-tag holds three compulsory child-elements. The first one is the data type of the constant, followed by the name and the value. Compared with UNO-IDL the sequence of the definition of a constant is almost the same.

Structures

	UNO IDL	XML-UNO-IDL
1		<?xml version="1.0" encoding="UTF-8"?>
2		<!DOCTYPE xmlidl SYSTEM "idlToXML_schema.dtd">
3	module com {	
4	module sun {	<xmlidl>
5	module star {	<idl_object name="RealPoint2D">
6	module geometry {	<module name="com">
7	struct RealPoint2D {	<module name="sun">
8	double X;	<module name="star">
9	double Y;	<module name="geometry">
10	};	<structure isPublished="false">
11	};	<id>RealPoint2D</id>
12	};	<struct_content>
13	};	<member>
14	};	<type>double</type>
15		<id>X</id>
16		</member>
16		<member>
17		<type>double</type>
18		<id>Y</id>
19		</member>
20		</struct_content>
21		</structure>
22		</module>
23		</module>
24		</module>
25		</module>
26		</idl_object>
27		</xmlidl>
28		

Coding 20: Structures in XML-UNO-IDL

The XML-UNO-IDL structure type is quite similar to the XML-UNO-IDL constants type. A structure begins with a <structure>-tag followed by its name in an <id>-tag. Attributes are defined between the <struct_content>-tag. While a constants-type has <const>-tags for each element in the content, the structure type has <member>-tags. Those tags contain child-elements which specify the type and the name of a certain element within a structure. Unlike the constants-type a structure-type does not define a value of an element.

Typedef

	UNO IDL	XML-UNO-IDL
1	module com {	<?xml version="1.0" encoding="UTF-8"?>
2	module sun {	<!DOCTYPE xmlidl SYSTEM "idlToXML_schema.dtd">
3	module star {	
4	module util {	<xmlidl>
5		<idl_object name="Color">
6	published typedef long Color;	<module name="com">
7		<module name="sun">
8	};	<module name="star">
9	};	<module name="util">
10	};	<typedef isPublished="true">
11	};	<type>long</type>
12	};	<id>Color</id>
13		</typedef>
14		</module>
15		</module>
16		</module>
16		</module>
17		</idl_object>
18		</xmlidl>
19		

Coding 21: Typedef in XML-UNO-IDL

This example shows a type definition of `com/sun/star/util/Color.idl`. Here, the primitive data type long gets a new name which is '`Color`'. In the XML-UNO-IDL the typedef-type is marked with a `<typedef>`-tag. This tag has two compulsory tags called `<type>` and `<id>`. While `<type>` specifies the data type which gets an alias name defined in `<id>`.

Exception

	UNO-IDL	XML-UNO-IDL
1	`module com {`	`<?xml version="1.0" encoding="UTF-8"?>`
2	`  module sun {`	`<!DOCTYPE xmlidl SYSTEM "idlToXML_schema.dtd">`
3	`   module star {`	
4	`    module embed {`	`<xmlidl>`
5		`  <idl_object name="UseBackupException">`
6	`     exception UseBackupException :`	`  <import>com/sun/star/io/IOException.idl</import>`
7	`           com::sun::star::io::IOException {`	`     <module name="com">`
8	`               string TemporaryFileURL;`	`        <module name="sun">`
9	`            }`	`           <module name="star">`
10	`    };`	`              <module name="embed">`
11	`   };`	`                 <exception isPublished="false">`
12	`  };`	`                    <id>UseBackupException</id>`
13	`};`	`                    <inheritance>`
14		`                       <complex_type>`
15		`                          <id>com</id>`
16		`                          <id>sun</id>`
16		`                          <id>star</id>`
17		`                          <id>io</id>`
18		`                          <id>IOException</id>`
19		`                       </complex_type>`
20		`                    </inheritance>`
21		`                    <exception_content>`
22		`                       <member>`
23		`                         <type>string</type>`
24		`                         <id>TemporaryFileURL</id>`
25		`                       </member>`
26		`                    </exception_content>`
27		`                 </exception>`
28		`              </module>`
29		`           </module>`
30		`        </module>`
31		`     </module>`
32		`  </idl_object>`
33		`</xmlidl>`
34		

Coding 22: Exception in XML-UNO-IDL

The above shown example about a conversion of the `com/sun/star/embed/UseBackupException.idl` is an exception with an additional attribute. A UNO-IDL exception type starts with the tag-name `<exception>` which holds the `isPublished` attribute. The name of the exception is specified in the `<id>` tag, under the `<exception>`-tag. This exception inherits from an exception called `IOException`. The start-tag of a single-inheritance is <inheritance> which is the equivalent to the ':'-operator in line 6 of the UNO-IDL. The inheritance is another UNO-IDL-type and it needs to get imported, which was done with the `<import>`-tag in line 6 of the XML-UNO-IDL-file. Since `IOException` is a type defined in another UNO-IDL-file it therefore gets a special markup which is called `<complex_type>`. The `<complex_type>`-tag has at least one `<id>`-child-element which specifies the package-structure of the imported type while each `<id>`-tag stands for one single package-directory.

If an exception also has additional attributes, an `<exception_content>`-tag must be set with all its child-tags. In this example the exception holds an exception content with one attribute. This attribute is of data type string and its name is '`TemporaryFileUrl`'. Nevertheless, if an exception does not hold an exception-attribute, an empty `<exception_content>` -tag must be set tough. This empty tag is of the form `<exception_content></exception_content>`.

Singleton

	UNO-IDL	XML-UNO-IDL
1	`module com {`	`<?xml version="1.0" encoding="UTF-8"?>`
2	`module sun {`	`<!DOCTYPE xmlidl SYSTEM "idlToXML_schema.dtd">`
3	`module star {`	
4	`module test {`	`<xmlidl>`
5		`<idl_object name="thePackageManagerFactory">`
6	`singleton theManagerFactory :`	`<module name="com">`
7	`com::sun::star::XPackageManager;`	`<module name="sun">`
8		`<module name="star">`
9	`};`	`<module name="test">`
10	`};`	`<singleton isPublished="false">`
11	`};`	`<interface_singleton>`
12	`};`	`<id>theManagerFactory</id>`
13		`<complex_type>`
14		`<id>com</id>`
15		`<id>sun</id>`
16		`<id>star</id>`
17		`<id>XPackageManager</id>`
18		`<complex_type>`
19		`</interface_singleton>`
20		`</singleton>`
21		`</module>`
22		`</module>`
23		`</module>`
24		`</module>`
25		`</idl_object>`
26		`</xmlidl>`
27		

Coding 23: Singleton in XML-UNO-IDL

This example implements a singleton over the interface `com/sun/star/XPackageManager`. A singleton is marked with a `<singleton>`-tag. Followed by an `<interface_singleton>`–tag or, if the singleton implements a service then a `<service_singleton>`-tag follows. Both the `<interface_singleton>`-tag and `<service_singleton>` -tag have the same child-elements: First, an `<id>`-tag for the alias/singleton name of the implemented interface or service. In this case, the alias/singleton name is `theManagerFactory`. Second a `<complex_type>` which holds the path and the name of the implemented interface/service. This example implements the interface `XPackageManager` in the package `com/sun/star`.

Interfaces

To keep the samples as easy as possible, there are two examples with interfaces. The first example shows an interface with attributes and the second example is an interface with methods only. Of course one interface can hold both attributes and methods.

Interface with attributes

	UNO IDL	XML-UNO-IDL
1	`#include`	`<xmlidl>`
2	`<com/sun/star/uno/XInterface.idl>`	`<idl_object name="XCore">`
3	`module com {`	`<import>com/sun/star/uno/XInterface.xml</import>`
4	`module sun {`	`<module name="com">`
5	`module star {`	`<module name="sun">`
6	`module oooimprovement {`	`<module name="star">`
7		`<module name="oooimprovement">`
8	`interface XCore {`	`<interface isPublished="false">`
9	`[readonly, attribute]`	`<id>XCore</id>`
10	`long SessionLogEventCount;`	`<interface_content>`
11		`<attribute>`
12	`[readonly, attribute]`	`<attr_flag>readonly</attr_flag>`
13	`boolean UiEventsLoggerEnabled;`	`<type>long</type>`
14	`};`	`<id>SessionLogEventCount</id>`
15	`};`	`</attribute>`
16	`};`	`<attribute>`
17	`};`	`<attr_flag>readonly</attr_flag>`
18	`};`	`<type>boolean</type>`
19		`<id>UiEventsLoggerEnabled</id>`
20		`</attribute>`
21		`</interface_content>`
22		`</interface>`
23		`</module>`
24		`</module>`
25		`</module>`
26		`</module>`
26		`</idl_object>`
27		`</xmlidl>`
28		

Coding 24: Interface with Attributes in XML-UNO-IDL

As this sample file of `com/sun/star/oooimprovment/XCore` shows, the interface only holds attributes in the content. An interface-definition in an XML-UNO-IDL-file starts with the tag `<interface>` and the attribute `isPublished`. The name of the interface is XCore and is specified in the next `<id>`-tag. Attributes are defined inside the `<interface_content>` with `<attribute>`. Each attribute tag consists of some tags which describe tag attribute. The first tag `<attr_flag>` sets flags like '`readonly`' or '`bound`' to the attribute. As the UNO-IDL-file shows, these flags are normally surrounded by squared parentheses and also holds the flag 'attribute'. In XML-UNO-IDL this is not necessary any more, because the tag `<attribute>` represents this flag already. If no `<attr_flag>` is set in the XML-UNO-IDL-specification of an interface-attribute, the attribute is '`readwrite`' by default. For all possible attribute flags see the appendix.

Next in the sequence is the compulsory attribute `<type>` which defines the data type of the attribute. Followed by the name of the attribute. Lines 16 till 20 are analogical.

Since the interface needs to be derived from another interface, an additional `<import>`-tag was set to import the base-interface '`XInterface`' in line 3 of the XML-UNO-IDL-file.

Interface with methods

	UNO IDL	XML-UNO-IDL
1	/**	<xmlidl>
2	* #include-section	<idl_object name="XAnimation">
3	*/	<module name="com">
4		<module name="sun">
5	module com { module sun { module	<module name="star">
6	star { module rendering {	<module name="rendering">
7		<interface isPublished="false">
8	interface XAnimation :	<id>XAnimation</id>
9	::com::sun::star::	<inheritance>
10	uno::XInterface {	<scope>
11		<id>com</id>
12	[oneway] void render (	<id>sun</id>
13	[in] XCanvas canvas,	<id>star</id>
14	[in] ViewState viewState,	<id>uno</id>
15	[in] double t	<id>XInterface</id>
16	)	</scope>
17	raises(com::sun::star::lang::	</inheritance>
18	IllegalArgument);	<interface_content>
19	};	<method>
20		<method_attr>oneway<oneway>
21	}; }; }; };	<type>void</type>
22		<id>render</id>
23		<param_content>
24		<param>
25		<param_attr>in</param_attr>
26		<scope>
27		<id>XCanvas</id>
28		</scope>
29		<id>canvas</id>
30		</param>
31		<param>
32		<param_attr>out</param_attr>
33		<scope>
34		<id>ViewState</id>
35		</scope>
36		<id>viewState</id>
37		</param>
38		<param>
39		<param_attr>in</param_attr>
40		<type>double</type>
41		<id>t</id>
42		</param>
43		</param_content>
44		<raises>
45		<scope>
46		<id>com</id>
47		<id>sun</id>
48		<id>star</id>
49		<id>lang</id>
50		<id>IllegalArgument</id>
51		</scope>
52		</raises>
53		</method>
54		</interface_content>
55		</interface>
56		</module>
57		</module>
58		</module>
59		</module>
60		</idl_object>
61		</xmlidl>
62		

Coding 25: Interface with Method in XML-UNO-IDL

This example about the interface `XAnimation` has been shortened a bit so that the example is not too detailed. The interface `XAnimation` inherits form the base-interface `XInterface`, this was specified within the <inheritance>-tag. The interface holds one method which is called render (`<id>`-tag) and has no return-value (`<type>void<type>`). The first element in the `<method>`-tag is a method-flag. In this case the method is a `oneway`-method. UNO-IDL marks the flags of a method in squared parentheses – see line 12 in the UNO-IDL-file. In general this tag is not compulsory. For all possible method_flags see the appendix.

A parameter-list of a method is defined within a <param_content>-tag. If a method doesn't hold any parameter then the <param_content> is left empty like <param_content> </param_content>.

Otherwise the tag will be extended with the definition of all parameters in the list. For each parameter a parameter flag must be specified in the <param_flag>-tag, followed by the data type and the name of the parameter. The method render holds three parameters.

Sometimes a method throws an error or an so called exception for special treatments. If this is the case as it is in the example, then the tag <raises> is appended to the <method>-tag. Within the <raises>-tag the fully qualified name of the exception must be defined. The method 'render' of the XAnimation interface throws the exception com/sun/star/lang/IllegalArgument.

Services

Since OpenOffice differs between two Service-types. It is necessary to show two examples of the different Service-types in XML-UNO-IDL.

The 'new-styled'-service / interface-service

	UNO IDL	XML-UNO-IDL
1	module com {	<?xml version="1.0" encoding="UTF-8"?>
2	module sun {	<!DOCTYPE xmlidl SYSTEM "idlToXML_schema.dtd">
3	module star {	
4	module bridge {	<xmlidl>
5	service UnoUrlResolver :	<idl_object name="UnoUrlResolver">
6	XunoUrlResolver;	<module name="com">
7	};	<module name="sun">
8	};	<module name="star">
9	};	<module name="bridge">
10	};	<service isPublished="false">
11		<id>UnoUrlResolver</id>
12		<inheritance>
13		<complex_type>
14		<id>XUnoUrlResolver</id>
15		</complex_type>
16		</inheritance>
17		<interface_service />
18		</service>
19		</module>
20		</module>
21		</module>
21		</module>
22		</idl_object>
23		</xmlidl>
24		

Coding 26: Interface Service in XML-UNO-IDL

This example sets a service to the global service manager of OpenOffice over the interface `XUnoUrlResolver`. A 'new-styled'-service starts with the tag `<service>` and the attribute `isPublished` followed by the <id>-tag which specifies the name of the service. As the service is just a derivative of an implemented interface, the service needs to inherit from this interface. In XML-UNO-IDL this is specified with an `<inheritance>`-tag like in line 12 till 16. The next tag in this example is an empty `<interface_service>`-tag. This tag normally stands for constructor-methods. For the specification of constructors see the syntax specification of the IDLtoXML-schema in the appendix.

The “old-styled” service / accumulated service

	UNO IDL	XML-UNO-IDL
1	`module com {`	`<xmlidl>`
2	` module sun {`	` <idl_object name="Year">`
3	`  module star {`	`  <module name="com">`
4	`   module xsd {`	`     <module name="sun">`
5	`    service Year {`	`        <module name="star">`
6		`           <module name="xsd">`
7		`              <service isPublished="false">`
8	`    interface XdataType;`	`                 <id>Year</id>`
9		`                 <accumulated_service>`
10	`    [property, maybevoid]`	`                    <service_members>`
11	`    short MaxInclusiveInt`	`                       <interface_inheritance isOptional="false">`
12		`                          <complex_type>`
13	`    [property, maybevoid]`	`                             <id>XDataType</id>`
14	`    short MaxExclusiveInt;`	`                          </complex_type>`
15		`                       </interface_inheritance>`
16	`    [property, maybevoid]`	`                       <service_property>`
17	`    short MinInclusiveInt;`	`                         <property_flag>maybevoid</property_flag>`
18		`                         <type>short</type>`
19	`    [property, maybevoid]`	`                          <id>MaxInclusiveInt</id>`
20	`    short MinExclusiveInt;`	`                       </service_property>`
21	`      };`	`                       <service_property>`
22	`   };`	`                         <property_flag>maybevoid</property_flag>`
23	`  };`	`                         <type>short</type>`
24	` };`	`                         <id>MaxExclusiveInt</id>`
25	`};`	`                       </service_property>`
26		`                       <service_property>`
27		`                         <property_flag>maybevoid</property_flag>`
28		`                          <type>short</type>`
29		`                          <id>MinInclusiveInt</id>`
30		`                       </service_property>`
31		`                       <service_property>`
32		`                         <property_flag>maybevoid</property_flag>`
33		`                          <type>short</type>`
34		`                          <id>MinExclusiveInt</id>`
35		`                       </service_property>`
36		`                    </service_members>`
37		`                 </accumulated_service>`
38		`              </service>`
39		`           </module>`
40		`        </module>`
41		`     </module>`
42		`  </module>`
43		` </idl_object>`
44		`</xmlidl>`
45		

Coding 27: Accumulated Service in XML-UNO-IDL

This example of an “old-styled”-service is much more complex than the “new-styled”-service. The accumulated service-type can contain interface-inheritances, service-inheritances and service-properties which are similar to attributes of an interface.

Service-inheritances are defined within a `<service_inheritance>`-tag followed by the fully qualified name of the derived service. Interface-inheritances are analogical to service-inheritances. Service-properties are like interface-attributes and follow almost the same sequence of definition. Unlike the `<property_flag>`, this is a specific flag for service-properties which are listed in the appendix.

Since the accumulated services are an old concept in OpenOffice it is recommended not to use them anymore [15].

6.5 Service Initializations / Register components in XML

In order to register a component in OpenOffice the location, the name and the activator of the component needs to be specified. For more details about that see the Developer's Guide in section 4.9 Deployment Options for Components [22]. For a registration of components using XML a specific XML structure is needed. This structure holds the implementation keys and the key values as followed:

	Component to register	XML-File
1	Service-name:	<?xml version="1.0" encoding="UTF-8"?>
2	com.sun.star.stoc.OServiceManager	<!DOCTYPE service_ini SYSTEM "ServicetoReg_schema.dtd">
3		
4		<service_ini>
5	Activator:	<key name="IMPLEMENTATIONS">
6	com.sun.star.loader.SharedLibrary	<key name="com.sun.star.stoc.OServiceManager">
7		<key name="UNO">
8		<key name="ACTIVATOR">
9	Location:	<data_item>com.sun.star.loader.SharedLibrary</data_item>
10	vnd.sun.star.expand:	</key>
11	$URE_INTERNAL_LIB_DIR/bootstrap.	<key name="SERVICES">
12	uno.dylib	<key name="com.sun.star.MultiServiceFactory"></key>
13		<key name="com.sun.star.lang.ServiceManager"></key>
14		</key>
15	Supported Services:	<key name="LOCATION">
16	com.sun.star.MultiServiceFactory,	<data_item>vnd.sun.star.expand:
17	com.sun.star.lang.ServiceManager	$URE_INTERNAL_LIB_DIR/bootstrap.uno.dylib
18		</data_item>
19		</key>
20		</key>
21		</key>
22		</service_ini>

Coding 28: Register a Component with XML

This service is called `com.sun.star.stoc.OServiceManager` which is located in `vnd.sun.star.expand:$URE_INTERNAL_LIB_DIR/bootstrap.uno.dylib`. The service gets activated by `SharedLibrary` which fully qualified name is `com.sun.star.loader.SharedLibrary`. The service `OServiceManager` consists of other services which are specified under the `<key>`-tag `SERVICES`. The content of every key is defined with a `<data_item>`-tag, which always holds the fully qualified package name of the component. As the example shows the `<key>`-tag can hold more then one `<key>`. This is necessary because a component can include more then one service.

When registering a component with XML keep in mind, that the header of the XML changes a bit: The underlying DTD-schema is called `ServiceToReg_schema.dtd` and not `idlToXML_schama.dtd`!

7 IDL-XML-Converter-Package

7.1 Run the Programs

Before the programs of the package IDL-Convert can be run, the `CLASSPATH` must be set to the file system. The `CLASSPATH` is an environment-variable which acts globally on the system. Java looks into this `CLASSPATH` variable to get the paths where it can find executable Java files. That means that Java looks for path-references where Java programs are stored.

Setting up CLASSPATHS for Windows

One possibility of setting a reference path to the programs of the IDL-Converter is by using the Windows console.

First open a new console-window and use the command 'set':

```
set CLASSPATH = Path\to\IDL_Converter_jars\package_name.jar
```

or if the new `CLASSPATH` should be appended to the existing one then:

```
set CLASSPATH = %CLASSPATH%;Path\to\IDL_Converter_jars\package_name.jar
```

Here the old values of `CLASSPATH` are considered when using `%CLASSPATH%` and `'Path\to\IDL_Converter_jars\package_name.jar'` is appended. Every path is separated with a semicolon '`;`'.

Example:

```
$ set CLASSPATH = %CLASSPATH%;C:\XMLReg.jar;C:\idl2xml.jar
```

In this example only the `*.jar`-packages `XMLReg.jar` and `idl2xml.jar` are added to the `CLASSPATH`.

The more convenient way is to use the 'System Properties'. Go to *Start* > *Settings* > *Control Panel* > *System Properties.* Then click '*Environment*' and choose the entry 'CLASSPATH' for editing. In the next dialog window the `CLASSPATH` can be set.

Setting up CLASSPATH on UNIX variants and MacOS

For UNIX variants and MacOS the procedure of setting up environment variables is almost the

same as on Windows. Note that the following command '`export`' is only available at 'bash'-consoles. On using a c-Shell like csh, the commend for setting variables is '`set`'.

For using bash

First open a new console-window and use the command '`export`':

```
export CLASSPATH=/path/to/IDL_Converter_jars/package_name.jar
```

or if the new CLASSPATH should be appended to the existing one:

```
$export   \
CLASSPATH=$CLASSPATH:/path/to/the/IDL_Converter_jars/package_name.jar
```

Here the old values of `CLASSPATH` are considered when using `$CLASSPATH` and '`path/to/the/IDL_Converter_jars/package_name.jar`' is appended. Every path is separated with a colon '`:`'. Note that there are no spaces between the equal sign.

Example:

```
$ export CLASSPATH=$CLASSPATH:/IDL_Converter/RegXML.jar:   \
                     /IDL_Converter/XMLReg.jar:/IDL_Converter/idl2xml.jar
```

7.2 Starting a program

After setting the CLASSPATH the program can be run from any directory on the file-system just use the command java and the package name and if necessary some additional options - those options will be described later. The general form of using those programs are:

```
$ java program_name filename [filename2]
```

First enter the command `java` followed by the program which should be run and lastly enter the input file for the program.

Example:

```
$ java reg2xml XControl.urd
```

Here the converter tool reg2xml is called for reading a registry file and converts it into an XML file

Specials for MacOS

It seems that OpenOffice has a problem with finding the correct path to the OpenOffice installation. This is actually a well known problem [21]. In this case every time a program of the IDLXML-Converter package is executed, a special parameter in the program call needs to be set:

```
$java
-Dcom.sun.star.lib.loader.unopath="/Applications/OpenOffice.org.app/Contents/MacOS" reg2xml XControl.urd
```

idl2xml

This program converts an existing UNO-IDL-file into an XML file

Syntax

```
java idl2xml filename [-c | --comment] [--output] [-i | --import]
                      [--nxDTD]
```

filename	Specifies the name of the file which should be converted to XML. In general this should be one single file ending with '.idl'. If a '.' is set in `filename` the interpreter includes all files ending with an '.idl' and converts them to XML.
-c | --comment	If this parameter is set all JavaDoc-styled comments will be taken into account. This parameter fills the `<docu>`-tag in an XML file.
-i | --import	This parameter is needed when the *.idl file contains an `#include`-command or a complex type which refers to another *.idl file.
--nxDTD	When using this parameter the DTD for checking the syntactical correctness of the XML file will not be exported. However, other programs may need this file.
--output	This parameter specifies the output directory of the XML file as fully qualified path.

xml2reg

This program parses an XML-file, which contains UNOIDL types and - if syntactically correct – converts it into an *.urd or *.rdb file. Second, the program can write service initializations as well. The program creates a new registry file if the registry doesn't exist or it writes into an existing registry. When writing in an existing registry the equal keys will be overwritten.

Syntax

```
java xml2reg filename [--key][--service][--output][--path][--merge]
                      [--suffix][--xnDTD]
```

filename	Specifies the name of the file which needs to be converted into a registry. In general this should be one single file ending with '.xml'. If a '.' is set in `filename` the program includes all files ending with '.xml' and converts them into an Open Office registry. Each XML file then gets its own registry file.
--key	This parameter specifies under which key the XML data should be written. The key always starts with a “/“followed by the key name. If `key` is not set, the root-node will be taken by default.
--service	This parameter tells xml2reg that the underlying data is service initialization data.

--path	If the XML file holds a `<complex_type>`-tag or an `<import>`-tag the path to the IDL-type must be set with this parameter. Otherwise xml2reg returns an error that it cannot find the complex type or the imported IDL-type.
--merge	If a '.' is specified in `filename` this option can be used to merge all *.xml-files into one registry.
--suffix	This parameter specifies if the output registry should be '*.urd' or '*.rdb'
--xnDTD	This prevents xml2reg from extracting DTD-schemas
--output	This parameter specifies the output directory of the XML file as absolute path.

reg2xml

This program extracts an Open Office registry into an XML-format.

Syntax

```
java reg2xml regfilename [--output] [--key] [--xnDTD]
```

regfilename	The name of the registry which needs to be extracted into an XML-format. This can be *.urd or *.rdb.
--output	This parameter specifies the output directory of the XML file as absolute path.
--key	Specifies a certain key which should be extracted from the registry. The key always starts with a “/“. If the key is not specified the root will be taken and all keys are then extracted.
--xnDTD	This prevents reg2xml from extracting DTD-schemas

regmerge

Regmerge merges two registries into one. This program can be used to extract specific keys into a new registry as well.

Syntax

```
java regmerge regfile1 regfile2 [--key1] [--key2]
```

regfile1	Name of the registry which needs to be merged into another one. This can be an *.urd or a *.rdb file
regfile2	Name of the registry which needs to get data from another registry. This can be an *.urd or a *.rdb file
--key1 | --regkey	The key name of regfile1 which needs to get merged into regfile2. This key always starts with a “/“. If this parameter isn't set all keys will be transferred.

--key2 | --mergekey	The key into which the data needs to get merged in regfile2. If no key is specified the default key is the root-node of the registry.

Examples

For examples see the provided example-folder. This folder contains for each program some examples for testing. There are README files which help you through the programs.

8 Round-up and Outlook

With an Interface Description Language it is possible to write a language-independent definition of a language-depending software module. The specification can then be used to enable other language-depending clients to use this module even it was written in a completely different programming language. Since Open Office allows different programming languages for the development of components such an approach is absolutely necessary. The Interface Description Language of Open Office is called UNO-IDL. This paper gives a short introduction about the different UNO-IDL types which can be specified in UNO-IDL.

Furthermore, Open Office uses a registry which collects all the UNO-IDL specifications to enable the usage of components for language-depending clients. The registry is a binary file and the content is organized hierarchically with keys and value-pairs. Unfortunately, Open Office hasn't provided a satisfactory documentation regarding the fields and structure of the registry. To go further into the development process of the IDLXML-Converter it was first necessary to establish a documentation of the structure and all the fields of the Open Office registry. The basic form and all its extensions are described in the paper.

The primary objectives were to set up an approach for writing UNO-IDL-components in XML, transforming registry data into XML, converting UNO-IDL files into an XML format and to convert XML-based UNO-IDL specifications into a registry format. With XML-UNO-IDL it becomes possible to get a structured form of IDL types. XML-UNO-IDL specifications can be published quite easily on web pages and through XML parsing technologies a lot of programming languages can handle XML quite well, especially Java. Therefore it is realistic to imagine using an XML-based registry instead of a binary-based registry. This would solve the problems with float- and hyper-values which cannot be stored in the binary-based registry yet.

To make all this possible a package has been written which is called IDLXML-Converter. The IDLXML-Converter consists of four programs:

- idl2xml – for converting UNO-IDL files into an XML-document
- xml2reg – writes UNO-IDL types specified in XML into a registry
- reg2xml – reads keys from the registry and writes them into an XML-document
- regmerge – merges two binary-based registries into one single registry

Nevertheless, there is still some further work to do. First, even if the above-mentioned XML-UNO-IDL examples look quite structured and meaningful, the description of particular interfaces can turn

out to be quite big. The result is that the XML file looks overloaded, barely human-readable and hard to edit. To solve this issue a graphical interface is needed which presents the XML-UNO-IDL specifications in a more fancy way, as well as generating these XML files. A graphical interface also brings the advantage that the user doesn't need to learn the UNO-IDL specification nor the XML-UNO-IDL specification and all its tags because the graphical interface will generate the code.

Second, since some of the IDLXML-Converter programs use Open Office commands every time on executing, the programs have to set up a connection to an Open Office instance. As a result this slows down the performance of the programs. Therefore there is a need to become independent of these Open Office commands and to increase the performance of the package.

9 Appendix

DTD-Schemata which are used by the IDL-XML-Package

IDL to XML Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT xmlidl (idl_object+)>
<!ELEMENT idl_object (import*,module+)>
<!ATTLIST idl_object name CDATA #REQUIRED>
<!ELEMENT import (#PCDATA)>

<!-- Module -->
<!ELEMENT module (docu?,(module | enumeration | single_const | structure | exception |
interface_forward | interface | typedef | constants | service | singleton)+)>
<!ATTLIST module name CDATA #REQUIRED>

<!-- Enumeration -->
<!ELEMENT enumeration (id, enum_content)>
<!ATTLIST enumeration isPublished (true | false) "false">
<!ELEMENT enum_content (enum+)>
<!ELEMENT enum (id,value?, docu?)>

<!ELEMENT single_const (docu?,(type | complex_type | sequence),id,value)+>

<!-- Constant -->
<!ELEMENT const (docu?,(type | complex_type | sequence),id,value)+>

<!-- Constants -->
<!ELEMENT constants (docu?,id , constants_content )+>
<!ATTLIST constants isPublished (true | false) "false">
<!ELEMENT constants_content (const+)>

<!-- Struct -->
<!ELEMENT structure (docu?,id, (inheritance? | struct_param*), struct_content)>
<!ATTLIST structure isPublished (true | false) "false">
<!-- inheritance -->
<!ELEMENT struct_param (#PCDATA)>
<!ELEMENT struct_content (member)+>
<!ELEMENT member ((type | complex_type | sequence), id, docu?)>

<!-- Exception -->
<!ELEMENT exception (docu?,id, (inheritance)?, exception_content+)>
<!ATTLIST exception isPublished (true | false) "false">
<!ELEMENT exception_content (member*)>

<!--Interface_Forward -->
<!ELEMENT interface_forward (docu?,id)>
<!ATTLIST interface_forward isPublished (true | false) "false">

<!--Interface -->
<!ELEMENT interface (docu?,id, (inheritance)?, interface_content)>
<!ATTLIST interface isPublished (true | false) "false">
<!ELEMENT interface_content (attribute| interface_inheritance | method)*>
<!ELEMENT attribute (attr_flag*, (type | complex_type | sequence), id, (get_attr | set_attr)*)>
<!ELEMENT attr_flag (#PCDATA)> <!-- bound or readonly -->
<!ELEMENT get_attr (raises+)>
<!ELEMENT set_attr (raises+)>
<!ELEMENT interface_inheritance (docu?, complex_type)>
<!ATTLIST interface_inheritance
       isOptional (true|false) "false">


<!ELEMENT method ((method_attr)?, (type | complex_type | sequence), id, param_content, raises*,
docu? )>
<!ELEMENT method_attr (#PCDATA)>
<!ELEMENT param_content (param*)>
<!ELEMENT param (param_attr, (type | complex_type | sequence), id)>
<!ELEMENT param_attr (#PCDATA)>

<!-- TypeDef-->
<!ELEMENT typedef (docu?, (type | complex_type | sequence), id)>
<!ATTLIST typedef isPublished (true | false) "false">

<!-- Service -->
<!ELEMENT service (docu?,id, inheritance?, (accumulated_service | interface_service))>
<!ATTLIST service isPublished (true | false) "false">
<!ELEMENT accumulated_service (service_members)>
<!ELEMENT service_members (service_inheritance | interface_inheritance | service_property)+>
<!ELEMENT service_inheritance (docu?, complex_type)>
<!ATTLIST service_inheritance
        isOptional (true|false) "false">
<!ELEMENT service_property (property_flag*, (type | complex_type | sequence), id, docu?)>
<!ELEMENT property_flag (#PCDATA)>
```

```
<!ELEMENT interface_service (constructor)*>
<!ELEMENT constructor (id, param_content,raises*, docu?)>

<!-- Singleton -->
<!ELEMENT singleton (docu?,(interface_singleton | service_singleton))>
<!ATTLIST singleton isPublished (true | false) "false">
<!ELEMENT interface_singleton (id, complex_type, docu?)>
<!ELEMENT service_singleton (id, complex_type, docu?)>



<!-- General elements-->
<!ELEMENT id (#PCDATA)>
<!ELEMENT value (#PCDATA)>
<!ELEMENT docu (#PCDATA)>
<!ELEMENT type (#PCDATA)>

<!ELEMENT raises (complex_type)+>

<!ELEMENT complex_type (id)+>
<!ELEMENT sequence (type | complex_type | sequence)>
<!ELEMENT inheritance (complex_type)+>
```

Service To XML Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT service_ini (key+)>
<!ELEMENT key (key | data_item )*>
<!ATTLIST key name CDATA #REQUIRED>
<!ELEMENT data_item (#PCDATA)>
```

Registry to XML Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT reg (key*)>
<!ATTLIST reg
    regname CDATA #REQUIRED>
<!ELEMENT key (key | values | data_item )*>
<!ATTLIST key
    name CDATA #REQUIRED>

<!ELEMENT data_item (#PCDATA)>

<!ELEMENT values (value_type, value_size, data)>
<!ELEMENT version (#PCDATA)>
<!ELEMENT value_type (#PCDATA)>
<!ELEMENT value_size (#PCDATA)>
<!ELEMENT data (version?, docu, file_name, type_class, type_name, supertype,fields, methods, refs)>
<!ELEMENT docu (#PCDATA)>
<!ELEMENT type_class (#PCDATA)>
<!ATTLIST type_class isPublished (true | false) "false">

<!ELEMENT supertype (supertype_count, supertype_name*)>
<!ELEMENT supertype_count (#PCDATA)>
<!ELEMENT supertype_name (#PCDATA)>

<!ELEMENT fields (field_count, field*)>
<!ELEMENT field_count (#PCDATA)>
<!ELEMENT field (docu, file_name,flags+, name, type_name, value)>
<!ELEMENT file_name (#PCDATA)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT type_name (#PCDATA)>
<!ELEMENT value (#PCDATA)>

<!ELEMENT methods (method_count, method* )>
<!ELEMENT method_count (#PCDATA)>
<!ELEMENT method (docu, flags, name, return_type, params, exceptions)>
<!ELEMENT flags (#PCDATA)>
<!ELEMENT return_type (#PCDATA)>
<!ELEMENT params (param_count, param*)>
<!ELEMENT param_count (#PCDATA)>
<!ELEMENT param (flags, name, type_name)>



<!ELEMENT exceptions (exception_count, exception*)>
<!ELEMENT exception_count (#PCDATA)>
<!ELEMENT exception (#PCDATA)>

<!ELEMENT refs (ref_count, ref*)>
<!ELEMENT ref_count (#PCDATA)>
<!ELEMENT ref (docu, flags, sort, type_name)>
<!ELEMENT sort (#PCDATA)>
```

Tables of possible 'direct-values' in the registry

Primitive Data Types

Name	Internal name	Internal number
none	RT_TYPE_NONE	0
boolean	RT_TYPE_BOOL	1
byte	RT_TYPE_BYTE	2
short	RT_TYPE_INT16	3
unsigned short	RT_TYPE_UINT16	4
long	RT_TYPE_INT32	5
unsigned long	RT_TYPE_UINT32	6
hyper	RT_TYPE_INT64	7
unsingned hyper	RT_TYPE_UINT64	8
float	RT_TYPE_FLOAT	9
double	RT_TYPE_DOUBLE	10
string	RT_TYPE_STRING	11

IDL-Types

Name	Internal name	Internal number
Invalid	RT_TYPE_INVALID	0
interface	RT_TYPE_INTERFACE	1
module	RT_TYPE_MODULE	2
struct	RT_TYPE_STRUCT	3
enum	RT_TYPE_ENUM	4
exception	RT_TYPE_EXCEPTION	5
typedef	RT_TYPE_TYPEDEF	6
service	RT_TYPE_SERVICE	7
singleton	RT_TYPE_SINGLETON	8
constants	RT_TYPE_CONSTANTS	10

Attributes/Property-Flags

Name	Internal name	Internal number
invalid	RT_ACCESS_INVALID	0
readonly	RT_ACCESS_READONLY	1
optional	RT_ACCESS_OPTIONAL	2
maybevoid	RT_ACCESS_MAYBEVOID	4
bound	RT_ACCESS_BOUND	8
constrained	RT_ACCESS_CONSTRAINED	16
transient	RT_ACCESS_TRANSIENT	32
maybeambiguous	RT_ACCESS_MAYBEAMBIGUOUS	64

maybedefault	RT_ACCESS_MAYBEDEFAULT	128
removable	RT_ACCESS_REMOVEABLE	256
attribute	RT_ACCESS_ATTRIBUTE	512
property	RT_ACCESS_PROPERTY	1024
const	RT_ACCESS_CONST	2048
readwrite	RT_ACCESS_READWRITE	4096
parameterized	RT_ACCESS_PARAMETERIZED_TYPE	16384
published	RT_ACCESS_PUBLISHED	32768

Parameter-Flags

Name	**Internal name**	**Internal number**
invalid	RT_PARAM_INVALID	0
in	RT_PARAM_IN	1
out	RT_PARAM_OUT	2
inout	RT_PARAM_INOUT	3
rest	RT_PARAM_REST	4

Method-Flags

Name	**Internal name**	**Internal number**
invalid	RT_MODE_INVALID	1
oneway	RT_MODE_ONEWAY	2
attribute get	RT_MODE_ATTRIBUTE_GET	5
attribute set	RT_MODE_ATTRIBUTE_SET	6

Reference-Flags

Name	**Internal name**	**Internal number**
invalid	RT_REF_INVALID	0
support	RT_REF_SUPPORTS	1
exports	RT_REF_EXPORTS	2
needs	RT_REF_NEEDS	4
parameter	RT_REF_TYPE_PARAMETER	5

10 List of Codings

Coding 1: Hello IDL..... 6
Coding 2: Modules in UNOIDL..... 8
Coding 3: Enumerations in UNOIDL..... 8
Coding 4: Constants in UNOIDL..... 9
Coding 5: Structures in UNOIDL..... 9
Coding 6: Typedef in UNOIDL..... 9
Coding 7: Exceptions in UNOIDL..... 10
Coding 8: Singleton in UNOIDL..... 10
Coding 9: Interfaces in UNOIDL..... 10
Coding 10: Interface with Attributes in UNOIDL..... 11
Coding 11: Interface with Method in UNOIDL..... 11
Coding 12: Interface Service in UNOIDL..... 12
Coding 13: Accumulated Service in UNOIDL..... 12
Coding 14: Registry Structure..... 16
Coding 15: Registry Structure after a Second Key..... 16
Coding 16: Hello XML..... 28
Coding 17: XML-UNO-IDL-Skeleton..... 29
Coding 18: Enumerations in XML-UNO-IDL..... 31
Coding 19: Constants in XML-UNO-IDL..... 32
Coding 20: Structures in XML-UNO-IDL..... 33
Coding 21: Typedef in XML-UNO-IDL..... 33
Coding 22: Exception in XML-UNO-IDL..... 34
Coding 23: Singleton in XML-UNO-IDL..... 35
Coding 24: Interface with Attributes in XML-UNO-IDL..... 36
Coding 25: Interface with Method in XML-UNO-IDL..... 37
Coding 26: Interface Service in XML-UNO-IDL..... 39
Coding 27: Accumulated Service in XML-UNO-IDL..... 40
Coding 28: Register a Component with XML..... 41

11 List of Illustrations

Illustration 1: Communication error without IDL..... 5
Illustration 2: Communication with IDL..... 6
Illustration 3: UNO-Components..... 7
Illustration 4: Compilation Chain..... 14
Illustration 5: HEAD Block..... 18
Illustration 6: Two Supertypes Added to the HEAD..... 20
Illustration 7: Constants-Pool Block..... 21
Illustration 8: CP Block Example: myEntry..... 22
Illustration 9: Field block..... 22
Illustration 10: Method Block..... 24
Illustration 11: Parameter Block..... 25
Illustration 12: Reference Block..... 26
Illustration 13: Registry Overview..... 27

12 List of Tables

Table 1: HEAD Block of a Registry's Binary Array..... 20
Table 2: CP Block of a Registry's Binary Array..... 21
Table 3: Field Block of a Registry's Binary Array..... 23
Table 4: Method Block of a Registry's Binary Array..... 25
Table 5: Parameter Block of a Registry's Binary Array..... 26
Table 6: Reference Block of a Registry's Binary Array..... 26

13 Literature

[1] OMG-IDL Details, OMG.org
http://www.omg.org/gettingstarted/omg_idl.htm. Last visited on 15 July 2010

[2] CORBA-FAQ – How about a high-level overview , OMG.org
http://www.omg.org/gettingstarted/corbafaq.htm#HowWork. Last visited on 14 July 2010

[3] Understanding UNO, OpenOffice.org
http://wiki.services.openoffice.org/wiki/Uno/Article/Understanding_Uno.
Last visited on 2 July 2010

[4] Registries in OpenOffice, OpenOffice.org
http://udk.openoffice.org/common/man/tutorial/uno_registries.html.
Last visited on 15 July 2010

[5] Introduction to Professional Usage of UNO, OpenOffice
http://wiki.services.openoffice.org/wiki/Documentation/DevGuide/ProUNO/Introduction.
Last visited on 21 June 2010

[6] Java Eclipse Tutorial, OpenOffice.org
http://wiki.services.openoffice.org/wiki/JavaEclipseTuto. Last visited on 26 June 2010

[7] FAQ about the OpenOffice-API, OpenOffice.org
http://de.openoffice.org/doc/faq/api/index.html. Last visited on 21 June 2010

[8] Generating Source Code from UNOIDL Definitions, OpenOffice.org
http://api.openoffice.org/docs/DevelopersGuide/Components/Components.xhtml#1_2_2_Generating_Source_Code_from_UNOIDL_Definitions.
Last visited on 2 July 2010

[9] The Registry Type Reader, OpenOffice.org
http://api.openoffice.org/docs/cpp/ref/names/RegistryTypeReader.
Last visited on 15 July 2010

[10] Grouping Definitions in Modules, OpenOffice.org
http://wiki.services.openoffice.org/wiki/Documentation/DevGuide/WritingUNO/Grouping_Definitions_in_Modules.
Last visited on 10 July 2010

[11] Using Services, OpenOffice.org
http://wiki.services.openoffice.org/wiki/Documentation/DevGuide/FirstSteps/Using_Services.
Last visited on 18 July 2010

[12] The UNOIDL Syntax Specification, OpenOffice.org
http://api.openoffice.org/docs/DevelopersGuide/Appendix/IDLSyntax/IDLSyntax.xhtml#1_UNOIDL_Syntax_Specification.
Last visited on 20 July 2010

[13] A Technical Introduction to XML, xml.com
http://www.xml.com/pub/a/98/10/guide0.html?page=2#AEN58. Last visited on 13 July 2010

[14] XML Syntax of XML, w3schools.com
http://www.w3schools.com/xml/xml_syntax.asp. Last visited on 13 July 2010

[15] Objects, Interfaces, and Services, OpenOffice.org
http://wiki.services.openoffice.org/wiki/Documentation/DevGuide/FirstSteps/Objects,_Interfaces,_and_Services.
Last visited on 20 July 2010

[16] Comparison of the UNO – CORBA Object Models, OpenOffice.org
http://udk.openoffice.org/common/man/comparison_uno_corba.html.
Last visited on 21 June 2010

[17] Typedefs, OpenOffice.org
http://wiki.services.openoffice.org/wiki/Documentation/DevGuide/AppendixA/Typedefs.
Last visited on 15 June 2010

[18] Exceptions, OpenOffice.org
http://wiki.services.openoffice.org/wiki/Documentation/DevGuide/AppendixA/Exceptions.
Last visited on 15 June 2010

[19] Using UNOIDL to Specify New Components, OpenOffice.org
http://wiki.services.openoffice.org/wiki/Documentation/DevGuide/WritingUNO/Using_UNOIDL_to_Specify_New_Components.
Last visited on 21 July 2010

[20] Interface Definition Language, IBM.com
http://publib.boulder.ibm.com/infocenter/wasinfo/v4r0/index.jsptopic=/com.ibm.websphere.v4.doc/wasee_content/corbaio/ref/rcidlop1.htm.
Last visited on 21 July 2010

[21] Forum entry about MacOS problem, dev@api.openoffice.org
http://comments.gmane.org/gmane.comp.openoffice.devel.api/20877
Last visited on 22 July 2010

[22] Deployment Options for Components, OpenOffice.org
http://api.openoffice.org/docs/DevelopersGuide/Components/Components.xhtml#1_9_Deployment_Options_for_Components
Last visited on 22 July 2010