OpenOffice.org API − new concepts in API design

Michael Hönnig
StarOffice GmbH, Webtop Group
mi@sun.com / Michael.Hoennig@Germany.Sun.COM

ABSTRACT

- **design paradigms and principles**
- **conflicts with remote transparency**
- **UNO object technology overview**
- **between Java and CORBA**
- **Examples with code**

After a short overview, in which areas our API can be used, I like to start with the design paradigm of the OpenOffice.org API: interfaces and support classes. I will continue with orthogonality as a design principle and its problem when it comes to remote transparency or the conflict of client optimized vs. server optimized APIs.

Next, I like to give a short overview of the used base technology UNO and how it relates to Java and CORBA. This includes some conflicts involved with these two worlds and how the OpenOffice.org API deals with it. OpenOffice.org its open source approach had a big influence to this aspect.

Finally, I will give you an overview about the topics covered by the API and how they interrelate. Two examples, one of it in detail with code, similar to Java, will show the power of the API and give some more tangibility.

IINTRODUCTION − APPLICATION AREAS

- **office suite automation macros**
- **use OpenOffice.org components**
- **modify StarOffice components**
- **integrate new components into OpenOffice.org environment**
- **adapt or exchange the UI (for example as in Sun ONE Webtop)**

The OpenOffice.org API is designed for multiple purposes, not only, like other office suite APIs for automation macros.

Through this API you can use the OpenOffice.org applications as components in your own application. For example there is a JavaBean wrapper available.

It is possible to modify the functionality of the OpenOffice.org components by wrapping into own components.

You can integrate your own components into OpenOffice.org to extend the functionality. Examples here are specialized linguistic modules, spreadsheet addins, import− and export− filters or specific chart diagram types.

And you can exchange the user interface of OpenOffice.org, like it is done in Sun ONE Webtop. What is currently not possible, is to modify the user interface. Here we expect a major shift in the implementation, for example to XUL. But this is not fixed yet.

OUR PARADIGM: INTERFACES AND SUPPORT CLASSES

- **Implementation Inheritance (no)**
  - partly implemented base classes
  - fat interfaces or deep hierarchy
  - components do depend on environment
  - high version dependence
- **Interfaces and Support−Classes (yes)**
  - communication only by interfaces
  - support classes for recurring implementations
  - components are independent from environment
  - low version dependence

Most object oriented systems are based on the implementation inheritance paradigm. This paradigm uses base classes with partial implementations and

implementation inheritance for specialization. When it comes to programming in the large, some disadvantages become obvious: you end up either in a deep inheritance hierarchy or with many methods in each class. Additionally, you will quickly realize how dependent specialized implementations become on their environment − the base class implementation, not only their interfaces. This results in a high version dependence, too. Another problem is mixed language programming, which is difficult with this paradigm.

Our approach uses only specifications for communications between components. The specifications are made up by several stereotypes, where interfaces play the major role. To reuse implementations, other components can be aggregated. Partial implementations are usually called support classes. This paradigm has certainly an overhead which is not appropriate for programming−in−the small, but for programming−in−the−large the advantages are show up: independence on concrete implementations of the environment and low version dependence. Additionally, this paradigm makes mixed language programming easy.

stereotypes

- **implementation classes**
- **services**
- **interfaces**
- **structs**
- **exceptions**
- **constants/constant groups/enums**

Stereotypes are categories of user defined types. Within a single stereotype the purpose is defined and different from the other stereotypes.

Implementation classes have a minor role in our paradigm. Their purpose is mostly to implement service specifications. Multiple implementation classes of the same service should be interchangeable.

Services specify the outer behavior of objects. They are pure specifications, with no implementation or data storage at all. Services consist of interfaces, properties and a description of how these elements interrelate. Additionally, services can inherit other interfaces in a way, by specifying that these other services have to be implemented as well. Some services are abstract specifications of a general behavior, like a TextContent. Others are concrete that they can be instanced. Instancing a service simply means to find an implementation class which supports this service and create an instance of it.

Interfaces are collections of methods which belong to a single aspect of functionality. They, too, do not have any implementation. Many interfaces are very general and their concrete behavior is specified in the services.

Structs consist of data only, they are very useful to transfer fixed collections of data. Exceptions are similar and used for error handling. Constant groups specify values.

stereotype: interface

On our design paradigm, interfaces are the only stereotype which contains methods.

One big advantage over having methods in classes which are partly or fully implemented, is compete programming language independence without a big loss in efficiency.

In a context of component technology, this is superior over APIs which defined methods at an implementation class level, as well. This is true, even if these are only used on helping objects because these classes would be either inefficient from other programming languages or would have to be implemented in all target programming language.

stereotype: service

In our design paradigm, services specify the behavior of objects without representing a certain implementation.

In many cases, multiple implementation of the same service make sense, even within a single environment like OpenOffice.org. An example can be a Text service, which exists in a very powerful implementation for a TextDocument and in a less powerful, but more efficient, implementation for texts in drawing objects.

Actually, there are two kinds of services

- services which can be instanced directly, for example a TextDocument: This kind of service specifies already enough details to form a useful component.
- and abstract base−services, for example a TextContent: This kind of service specifies only a general behavior.

stereotype: struct

In our design paradigm, structs are used as plain data containers of mixed types. They have no methods.

Though this usage seems to break basic OOP rules, it is a big advantage for efficiency in component environments.

A struct without methods can easily be transferred into other programming language environments. A struct with methods (a class) either would have to be implemented in all

programming languages or access would be very inefficient.

stereotype: implementation class

In our design paradigm, implementation classes play a minor role. They have to fulfill the service specification they predict to support, period.

Implementation classes can be instanced by

- either a factory (specified by a service name)
- implicitly by accessing sub−objects

Programs should be implemented against the service description of the components, not the actual implementation classes.

COMMON DESIGN PATTERNS

- **Factory global/doc.**
- **PropertySet/−Access etc.**
- **Collections/Containers**
- **Enumerators/Iterators**
- **X...Supplier**
- **Events**
- **Exceptions for Error Handling**

Many design patterns reoccur over and over again in our API. Here is a list of the most important design patterns in OpenOffice.org API:

New instances of objects are created by factories, mostly specified by their service name. Factories are simply interfaces with a method which creates new object of a specified type. We have a global factory and many components have their own factory for sub−components.

PropertySets are used to access non−structural member data of objects, for example formatting information. The interfaces resemble pretty much the PropertySet of Java.

Collections offer a generic access to multiple sub−objects of the same type or at least the same base type. Containers additionally offer methods to add, change and remove sub−objects.

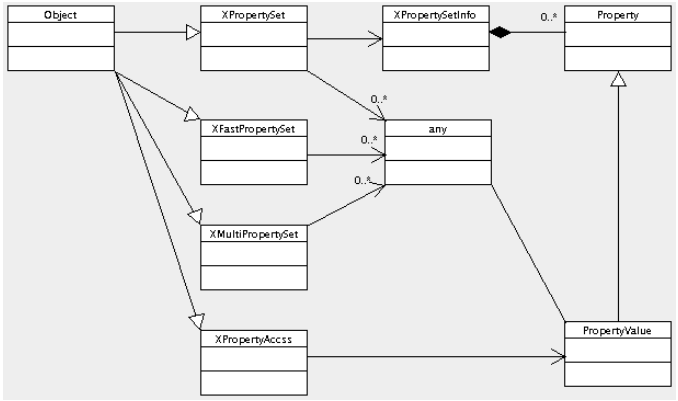
Enumerators or iterators, including cursors, make it possible to walk through sub−objects. They are always created by the container, this way efficient iterators can be made available.

X...Supplier interfaces are used to access structural data, this means distinct sub−objects, when these sub−members are optional and accessors don't belong in another interface.

Events are used to notify external components about changes. We use the same event concept like in Java. Exceptions are used for error handling.

EXAMPLES

example: PropertySet

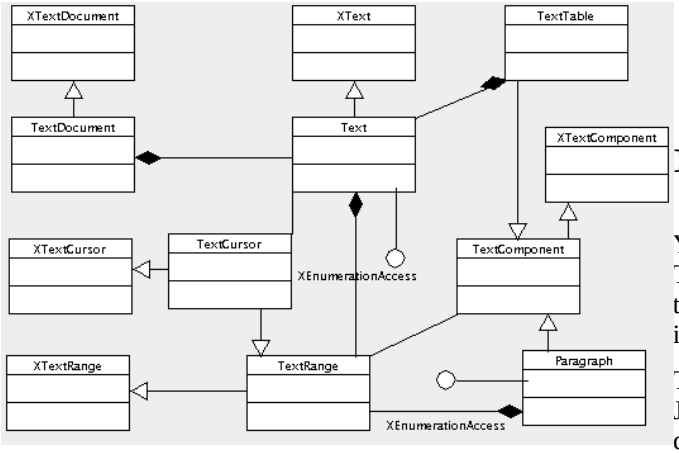


The PropertySet concept of OpenOffice.org is actually very similar to the one of Java. We have probably a stricter use of PropertySets for non−structural elements only (execpt for the generic XPropertySet when using introspection).

Here we have a nice example of breaking our orthogonality rule and why remote transparency does not exist. At least, it does not exist when it comes about efficiency. XMultiPropertyAccess and XPropertyAccess are mostly for remote access (or at least access from a different process); their purpose is to reduce the number of calls, for the getter methods even synchronous calls.

XPropertySet, on the other hand, is very convenient for the application programmer, but slow in distributed environments.

example: TextDocument / structure



A TextDocument consists of the main Text object an many other sub−objects, like style−sheets, meta information etc., which are not shown in this diagram.

A Text service is used not only as a main text object of a TextDocument, it is used as the text object in table cells or text frames as well. It is even used as a text object in drawing objects or spreadsheet cells. Of course the concrete Text in a TextDocument has more interfaces, thus offers more functionality.

The Text itself consists of a series of TextContents, mostly Paragraphs and TextTables or TextSections. These can be enumerated, by creating an enumerator object at its XEnumerationAccess interface.

A Paragraph consists of one or multiple special TextRanges, called text portions. Each portion spans an area with same attributes and/or TextContents bound to it. Bound TextContents comprise for example TextFrames, TextEmbeddedObjects and TextGraphics.

Example: TextDocument / code

```
XTextDocument xDoc = xEnv.loadComponentByURL(
      “file:///..., ...  );

XText xText = xDoc.getText();

XEnumerationAccess xNodes =
      xText.queryInterface(
          XEnumerationAccess );
XEnumeration xNodeIter =
      xNodes.createEnumeration();

while ( xNodeIter.hasMoreElements() ){
      XParagraph xPara = xNodeIter.
          nextElement().queryInterface(
          XParagraph );

      if ( xPara != null )      {

            ...

      }
}
```

You can obtain TextCursors from a Text object as well. These TextCursors are independent from the TextCursor in the views, although these are TextCursors too. A TextCursor is a TextRange too, just one which can be moved.

This piece of pseudo code, which is somewhat similar to Java, shows a simple iteration through a TextDocument. The differences to real Java are neglectable for our purpose, the API, real Java code would be a little bit longer, for example because type names cannot simply be used as arguments. But this belongs more in a UNO presentation, anyway.

The first line loads a new document from the desktop environment. The desktop environment itself is a singleton service which can be instanced by the global UNO service manager. XComponentLoader::loadComponentByUR()L expects the URL from which the document is to be loaded, and some other arguments which are omitted here.

The second line casts the loaded component to a text document and gets the main text from it. In a real live application, we would need some error handling code here, of course.

The next line gets the XEnumerationAccess interface from our main text object. In this context, the XEnumerationAccess enumerates the nodes (mostly paragraphs) of the text.

This iteration is shown here too. Actually, to simplify the code, we omitted the cast from the any, which is returned by XEnumeration::nextElement() to XTextContent.

The last few lines try to get an XParagraph interface from the node, which could be a TextTable too, for example. And if it is available, we execute some more code on the paragraph.

```
XTextDocument xPara = ...;

XEnumerationAccess xParaPortions =
      xPara.queryInterface(
          xEnumerationAccess );
      XEnumeration xPortionIter =
          xParaPortions.createEnumeration();

while ( xPortionIter.hasMoreElements() ){

      XPropertySet xPortionProps =
            xPortionIter.nextElement().
```

```
            queryInterface( XPropertySet );
        any aBoldVal =
        xPortionProps.getValue(CharWeight);
        if ( aBoldVal.getBoolean() )     {
            ...
        }
}
```

This examples continues the first one by working on the paragraph object we have found by iterating through the text document.

Again, we get an XEnumerationAccess from our object, this time from our paragraph. And we create an enumeration from it.

This enumeration enumerates text portions within the paragraph, A text portion is a special kind of TextRange which has the same attributes like font etc. If, for example, all attributes within a paragraph were the same, just one single word in the middle was in bold, we would have three text portions in this paragraph: one with non−bold, one with bold and another one with bold characters.

In this example, we get exactly this bold attribute from the text portion, and if it is set, we execute some special code.

REFERENCES

1. http://api.openoffice.org
2. http://udk.openoffice.org