OpenOffice.org:

Selected Pitonyak' Nutshells in ooRexx

Michael Gmeiner
Vienna University of Economics and Business Administration
Reg. No. 0150323
E-Mail: h0150323@wu-wien.ac.at

Seminar Paper
Department of Business Informatics
Prof. Dr. Rony G. Flatscher
Department Chair "Information Systems and Operations"

Table of Contents

List of Figures

1 Introduction

1.1 Abstract

This paper tries to find out how Open Object Rexx (ooRexx) can be used to automate OpenOffice.org using various concepts like BSF and UNO. It explains which software components are needed and what benefits can be expected from writing OpenOffice.org macros. The theoretical aspects are backed up by various nutshell examples which demonstrate how to apply OpenOffice.org macro knowledge.

1.2 Problem Discussion

The automation of Microsoft Office via VBA is well documented and approved. Is it also possible to automate OpenOffice.org applications to benefit from this free, platform-independent office suite? Is ooRexx a suitable macro language to achieve this goal? Can existing OOo Basic macros be used as a guideline?

The result should also follow OpenOffice.org's cross-platform approach, i.e. no platform-specific concepts should be used, additionally no proprietary software tools must be used.

1.3 Approach

The necessary software components are listed and briefly described in the first step. An introduction to the script language ooRexx is provided in order to understand the nutshell examples in later chapters. These examples try to illustrate the capabilities of OpenOffice.org in conjunction with a powerful macro language like ooRexx. The basic concepts of the UNO framework are explained at first theoretically and later on practically, in the provided examples. The emphasis of this paper lies on the examples which can be used as a guideline to create own OpenOffice.org macros. Most macros are based on Andrew Pitonyak's OOo Basic nutshells ([Pito04] and [Pito08])

2 Technical Requirements

2.1 Open Object Rexx

Open Object Rexx ("ooRexx" or simply "Rexx") is a scripting language originally developed by IBM, roughly comparable to Tcl or Python. The main goal was to create a powerful yet easy to learn macro language for any system. It is a high level programming language which is very close to human language, in fact it resembles pseudo code regarding its simplicity. IBM open-sourced ooRexx in December 2004, today it is freely available on virtually any platform, including Windows, Linux, MacOS, Amiga OS as well as on portable systems like Palm OS, EPOC and Windows CE.

2.1.1 History

The first specification for the language (originally called REX) was dated 29 March 1979. IBM programmer Mike Cowlishaw wanted to create a new macro language for IBM mainframes in order to replace EXEC 2, a powerful macro language which was, however, lacking readability because of its complex syntax. Cowlishaw aimed to develop a similarly powerful language with a more classical syntax of high level languages like Pascal.

The first implementation was available in late 1979 and was spread via IBM's world-wide internal network. It quickly gained popularity because of its simplicity, and benefited from numerous contributions over the network. Because of ample feedback it quickly evolved to meet the user's requirements. Not only internal IBM programmers were avid users of the language, it also became very popular among IBM's customers, which also led to a strong commitment of IBM towards Rexx.

In 1985 the first non-IBM implementation of Rexx was released for PC-DOS, a commercial software known as Personal Rexx. 1992 was the birth year of the first Open Source ports: Ian Collier's REXX/imc for Unix and Anders Christensen's Regina (later adopted by Mark Hessling) for Windows and Linux. 1996 was another landmark year, as the ANSI standard for Rexx has been completed and published.

In 2004, IBM announced to release the object oriented version of Rexx, ooRexx, under the Common Public License (an Open Source license published by IBM). In 2005, the Rexx Language Association (RexxLA) took over the development.

IBM has also released an other Rexx-spinoff called “NetRexx”, which was designed to run Rexx programs on the Java Virtual Machine. The syntax as well as the object model differ considerably from Classic/Open Object Rexx, so it will not be discussed any further in this paper.

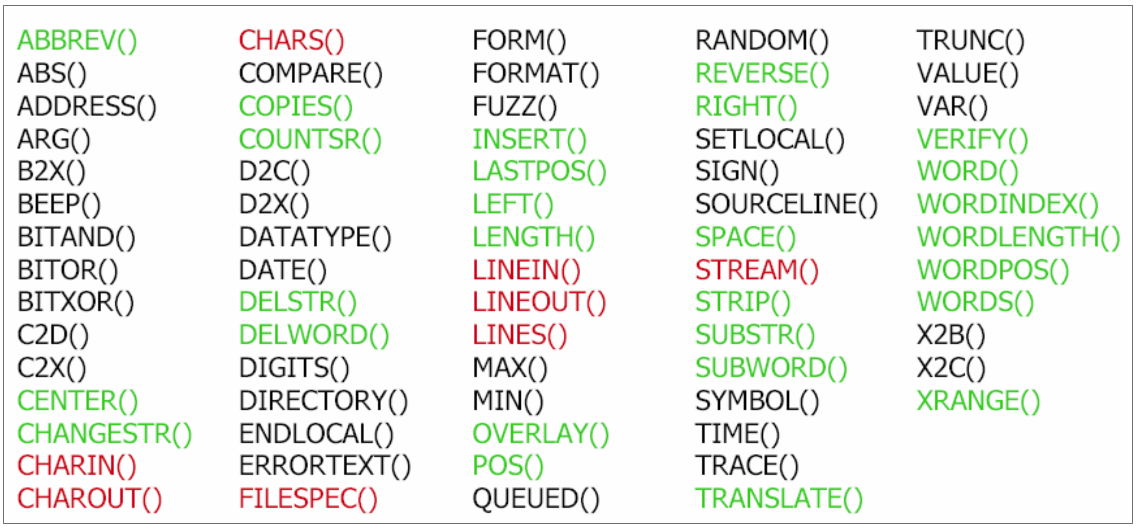
The following chapter will outline basic programming concepts of ooRexx. It is not meant to be a comprehensive programming guide, instead it should provide general knowledge and fundamentals of the language in order to understand the OpenOffice.org nutshell examples in later chapters.

2.1.2 Language aspects

As mentioned before, the Rexx language facilitates English-like statements, which highly eases the understanding of the code. Examples would be for instance the SAY statement, which prints text to the default output device (usually the screen). Other statements, which are similar to other high level programming languages include the `IF .... THEN ... ELSE` statement, or the `DO .... END` block/loop statement. All statements and instructions are case insensitive, strings (delimited via quotation marks) of course are processed as provided by the user (i.e., case sensitive).

Rexx has only twenty three instruction (like `SAY`, `CALL` and `PARSE`) which results in a fast learning experience; furthermore Rexx has no explicit data types (like Integers, Strings, floating-point numbers etc.), which means that no variable declarations have to be performed, and variables are processed automatically in the right way.

Beside of the aforementioned instructions, Rexx offers a limited number of built-in functions:



ABBREV()	CHARS()	FORM()	RANDOM()	TRUNC()
ABS()	COMPARE()	FORMAT()	REVERSE()	VALUE()
ADDRESS()	COPIES()	FUZZ()	RIGHT()	VAR()
ARG()	COUNTSR()	INSERT()	SETLOCAL()	VERIFY()
B2X()	D2C()	LASTPOS()	SIGN()	WORD()
BEEP()	D2X()	LEFT()	SOURCELINE()	WORDINDEX()
BITAND()	DATATYPE()	LENGTH()	SPACE()	WORDLENGTH()
BITOR()	DATE()	LINEIN()	STREAM()	WORDPOS()
BITXOR()	DELSTR()	LINEOUT()	STRIP()	WORDS()
C2D()	DELWORD()	LINES()	SUBSTR()	X2B()
C2X()	DIGITS()	MAX()	SUBWORD()	X2C()
CENTER()	DIRECTORY()	MIN()	SYMBOL()	XRANGE()
CHANGESTR()	ENDLOCAL()	OVERLAY()	TIME()	
CHARIN()	ERRORTEXT()	POS()	TRACE()	
CHAROUT()	FILESPEC()	QUEUED()	TRANSLATE()	

Figure 1: Built in Rexx functions [Flat01]

These Functions are available to the programmer to perform different tasks like String operations (`legth()` or `substr()`) or file operations (`filespec()`, `lineout()`). As these functions are not used in the examples in the following chapters, the listing is provided for the sake of completeness. Additional functions can be added easily by including libraries in order to extend the Rexx functionality. An example for such a library will be discussed in later chapters.

Comments can be placed at any position within the code. Rexx offers two ways to mark text as a comment: Two hyphens (`--`) signal that the rest of the following line is a comment. Using the comment pattern known from C/C++/Java (`/* ... */`) more lines of code can be marked as comments, and even nested comments are allowed.

2.1.3 Syntax

Block

A block of code starts with `DO` and ends with `END`:

```
DO
       <instructions>
END
```

Flow control

The well known `IF - THEN - ELSE` concept is also available in Rexx:

`IF` <condition> `THEN` <instruction> `ELSE` <instruction>

Use the block concept in order to execute more instructions:

```
IF <condition> THEN
DO
      <instructions>
END
```

Loops

Loops can be realized in several ways, all of them are based on the `DO` block instruction. The well known BASIC/C etc. FOR-loop looks like this for instance:

```
DO i = 1 TO 99
      SAY i
END
```

This loop prints the value of `i` every time it iterates through the loop.

Procedures

Rexx offers the concept of routines to program procedures:

```
::routine foo
      use arg x, y
      <instructions>
      return z
```

This procedure “foo” has two arguments (`x` and `y`) and returns the value of `z`. It can be called from within the code like this:

```
a = foo(x, y)
```

If a procedure has no return value, it can be called in an easier manner:

```
CALL foo x y
```

Requires – directive

In order to access functionality from other Rexx programs, the `::REQUIRES` directive has to be used. For programming in OOo, the class UNO has to be declared in every program. It acts as the binding link between Rexx and OOo, and offers additional functionality. [Flat03]

```
::requires UNO.CLS (to get OOo support)
```

Interaction with objects

The object oriented paradigm is based on objects communicating with each other. The communication is realized via messages. The “Twiddle” (~) symbol after an object signals such a message:

```
object~message
```

A message could be a function the object provides, so an example for this would be a Person (object `aPerson`) which says something (function/message `saySomething()`):

```
aPerson~saySomething(“Hello!”)
```

For programmers familiar with C++ or Java, the Twiddle is equals the . in Java and returns the outcome of the message.

Arrays

Classic Rexx does not offer real arrays (as known from Java for example). ooRexx however offers array functionality via the array object. Note the message (`new`) after the Twiddle!

```
anArray = .array~new
```

2.2 Java and the Bean Scripting Framework (BSF)

2.2.1 Java

The huge success of the programming language Java led to a boom Java programs for both the desktop market (Java standalone programs) as well as for web applications (Java applets). Thanks to the platform independent concept programmers can easily develop software for virtually any available platform, with no or only minor modifications needed.

Java however has certain drawbacks depending on the scope of the software project. (Refer to [King01]) It is not only a rather difficult language (heavily influenced by C/C++), but it also tends to create rather large code because of its strict syntax and object oriented concepts. Even for a small 'Hello World' program a class has to be created, the main function has to be implemented, etc.

Additionally, Java programs have to be compiled first in order to execute the program.[1]

Example: HelloWorld in Java vs. Rexx:

```
<HelloWorld.java>
Class HelloWorld
{
        public static void main (String[] args)
        {
                System.out.println ("Hello, world! This is Java speaking.");
        }
}
```

Rexx:

```
<HelloWorld.rex>
SAY "Hello, world! This is Rexx speaking."
```

As shown in the example above, script languages like Rexx offer a much easier and intuitive way of coding, which in turn leads to a dramatically increased coding speed. These languages lack however the huge class library offered by Java. Java features by default a vast class library which can be used to perform various tasks, ranging from a GUI development, 2D/3D graphics development to client/server communication etc. Additional Java libraries created by individual programmers can be found in various Java web communities.

In order to combine the advantages of Java and script languages ("best of both worlds", so to speak), the Bean Scripting Framework has been created.

2.2.2 Bean Scripting Framework (BSF)

The BSF is a class library for Java which enables script languages to access Java functionality (classes), and vice versa, Java to run script language code within Java. Using the BSF, script language programmers can utilize the huge array of Java classes in their own scripts, and Java programmers can use script languages to automate Java applications and benefit from the convenient programming style these languages offer.

The BSF was born in 1999 as a project of IBM and later handed over to the Apache Software Foundation. It is now part of Apache's Jakarta project which fosters and supports Java Open Source Software. The latest release is version 2.4.0, however a beta version of 3.0 is already available

[1] The javac compiler will compile the source code first, and the java interpreter will run the bytecode created by the compiler.

> “The two primary components of BSF are the `BSFManager` and the `BSFEngine`.
>
> The `BSFManager` handles all scripting execution engines running under its control, and maintains the object registry that permits scripts access to Java objects. By creating an instance of the `BSF-Manager` class, a Java application can gain access to scripting services.
>
> The `BSFEngine` provides an interface that must be implemented for a language to be used by BSF. This interface provides an abstraction of the scripting language's capabilities that permits generic handling of script execution and object registration within the execution context of the scripting language engine.
>
> An application can instantiate a single `BSFManager`, and execute several different scripting languages identically via the `BSFEngine` interface. Furthermore, all of the scripting languages handled by the `BSFManager` are aware of the objects registered with that `BSFMan-ager`, and the execution state of those scripting languages is maintained for the lifetime of the `BSFManager`.” [ApJa01]

Currently (version 2.4.0) there are many languages directly supported by BSF, for example Python, Tcl, JavaScript and NetRexx. Some other languages are also supported, they however require a separate engine in order to gain BSF functionality. Among these languages is ooRexx, which has to rely on the BSF4Rexx engine.

2.2.3 BSF4Rexx

BSF4Rexx is the implementation of the BSF engine for Rexx. It is written in C++ and acts like a bridge between Java, the BSF and Rexx:

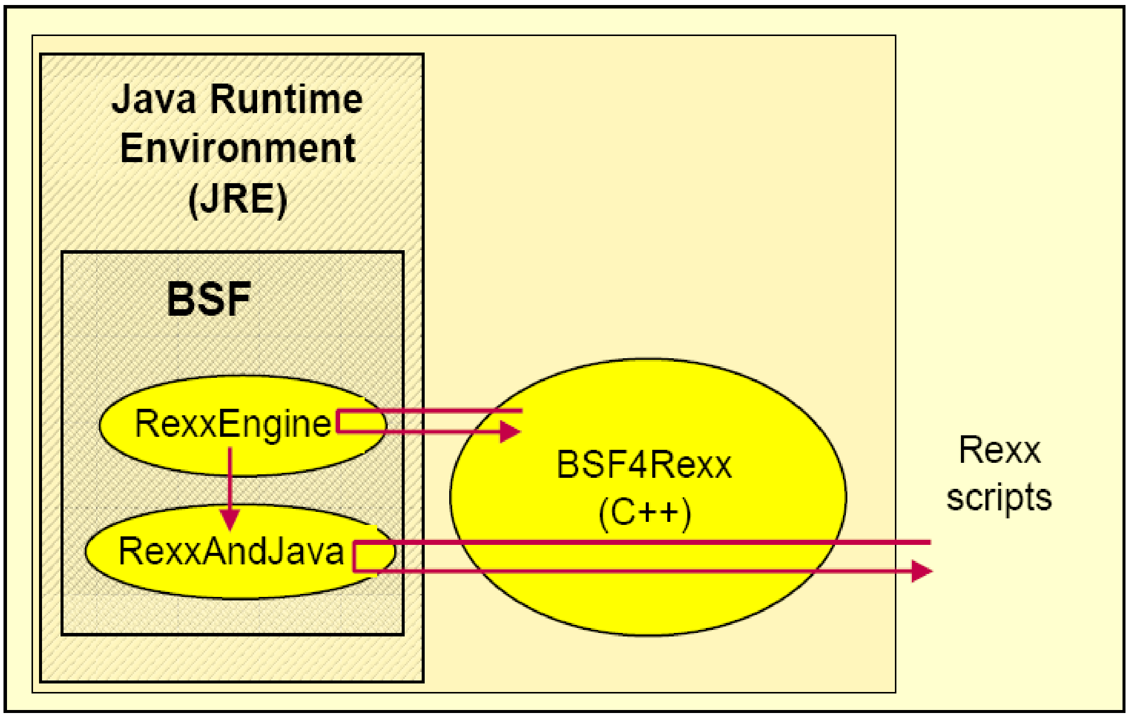


Figure 2: BSF4Rexx linking Java and Rexx [Flat02]

2.2.4 History

The first proof of concept was done in 2000/2001 by a student named Peter Kalender. As he was a student of the University Essen the first BSF4Rexx version was named „Essener Version“. Work went on and in Spring 2001 BSF4Rexx was presented to the RexxLA. Two years later a new version of BSF4Rexx, called the „Augsburger Version“, was introduced. It had some bugs fixed and external Rexx functions added, including the loading of Java on Windows and Linux platforms. [Flat02]

“Vienna Version”

The current Version, named “Vienna Version”, has been developed at the Vienna University of Economics by Prof. Dr. Rony G. Flatscher, is a complete revision of the Essener Version: fully compatible, and with extended functionality for an easier usage. The most recent version is 2.6 and is freely available for download.

Getting BSF functionality in Rexx

In order to get access to the vast Java resources, the BSF4Rexx class (`BSF.CLS`) has to be included in the Rexx script. The two alternate methods are either via the CALL statement, which has to be placed before any code which needs the BSF4Rexx functionality. Another method would be the inclusion of BSF4Rexx via a `::REQUIRES` directive, which has to be placed at the end of the script; the Rexx interpreter will invoke the directive at the beginning of the script, thus the BSF4Rexx functionality will be given for the whole script[2].

The two methods demonstrated in code snippets:

```
/* Including BSF4Rexx in a script, method one */
/* 1. via the CALL statement     */

CALL BSF.CLS

/* BSF4Rexx functionality now available */
```

```
/* Including BSF4Rexx in a script, method two */
/* 2. via the ::REQUIRES directive     */

/* BSF4Rexx functionality is available from the beginning! */

::REQUIRES BSF.CLS
```

2.3 OpenOffice.org

OpenOffice.org (OOo) is an Open Source office suite similar to Microsoft Office. The most recent version is 2.4, and one key feature of the suite is its cross-platform approach utilizing the JRE (Java Runtime Environment). Supported platforms include Windows, Linux, BSD, MacOS X and Solaris. [Port01]

2.3.1 History

The application, originally titled “Star Office”, was developed by the German company StarDivision. In 1999 the code was purchased by Sun Microsystems, which soon released the code under the the GNU LGPL, creating an Open Source office suite which should be in the future developed by the community. The mission statement of OOo:

[2] Note: The examples in this paper will all use the `::REQUIRES` directive

"*To create, as a community, the leading international office suite that will run on all major platforms and provide access to all functionality and data through open-component based APIs and an XML-based file format.*" [OOo01]

2.3.2 Applications

The functionality of OOo is offered by several different components. The main components of OOo in a nutshell:

	Writer	Word processor
	Calc	Spreadsheet calculation
	Impress	Presentation program
	Base	Database
	Draw	Vector graphics editor
	Math	Formula editor

2.3.3 Programming OOo

OOo features an own macro language, OOo Basic, in order to perform repetitive tasks and expand the functionality of the office suite. OOo Basic is comparable to VBA (Visual Basic for Applications) found in Microsoft's Office suite. The OOo macro language is very flexible and powerful, and as it is based on the BASIC programming language, so many programmers are already familiar with the concepts of this language.

OOo offers a library concept in order to organize macros: OOo Basic code is stored as a "macro", and related macros are stored in modules. Modules can be grouped in a library, and libraries are stored in a library container. The OOo application can act as a library container, as can any OOo document.

OOo macros can be accessed and executed via Extras – Macros – Organize Macros. Note that the macros are separately organized for each programming language!

With OOo 2.0 and above, the office suite also supports various other programming languages like JavaScript and Python beside OOo Basic. Thanks to a standardized architecture, which will be discussed in the following chapter, programmers are no longer bound to a single macro language and thus free to use their preferred language.

An additional way to create macros is via the macro recorder. The macro recorder, in essence, records the keystrokes and the mouse inputs performed by the user and creates OOo Basic code which replicates the outcome of the user input. While this is a very convenient method to create macros, it allows only little control over the created code and does not allow the user to take full advantage of the OOo Basic capability. It is thus only recommended for inexperienced users or very simple tasks.

2.3.4 UNO architecture

The internals of OOo are based on Universal Network Objects (UNO). The UNO structure is a component model that offers interoperability between different programming languages, object models, machine architectures and processes.[Pito04] It is somewhat comparable to Java packages, in a way that UNO follows the object oriented concept, allows inheritance, objects offer certain interfaces, communicate with each other etc.

In essence, various internal components of OOo are implemented using UNO, and programming languages can access these objects and manipulate them. Programming OOo is thus independent of the programming language and follows the UNO structure.

Additionally, UNO also facilitates a client/server architecture. Usually the client and the server will run on the same machine, but it is also possible to separate these two. The communication is realized via TCP/IP sockets, the protocol used is called urp (UNO remote protocol) and the operating system of the client as well as of the server machine is irrelevant.[Flat03]

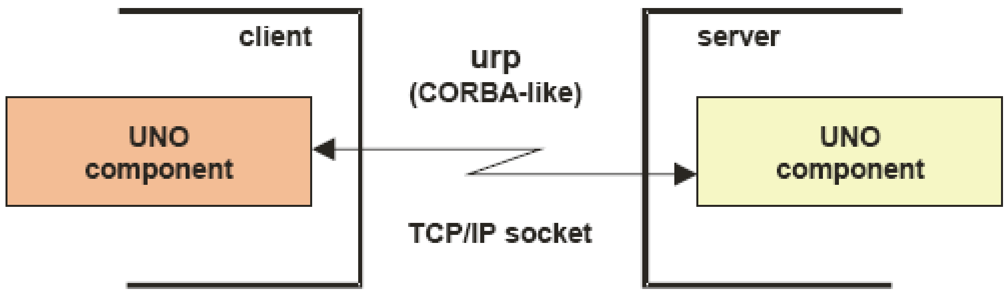


Figure 3: The urp protocol [Flat03]

2.3.5 Object Model of OpenOffice.org

It is important to understand how the object model of OOo is realized in order to automate OOo applications.

Services

Simply stated, objects in OOo are called services.[3]

”A service abstractly defines an object by combining interfaces and properties. A UNO service typically consists of one or more interfaces and one or more UNO structures combined to encapsulate some useful functionality. A UNO interface defines how an object interacts with the outside world; a UNO structure defines a collection of data; and a UNO service combines them together.” [Pito04]

One of the most frequently needed services is for example the Desktop service. It is used to load documents, create new documents and navigate between open documents.

Interfaces

Every service exposes a range of interfaces to manipulate the object. An example for an interface would be a function which returns data from the service, or a method used to modify data within a service.

Interfaces are identified via the leading “X” in the name, for example the service TextCursor offers an interface called XTextCursor.

[3] Strictly speaking, a service is only the abstract definition of an object. For simplicity's sake, the word “object” will be used as a synonym for “service” in the following chapters of this paper.

UNO structures

The properties of a service are called structs (structures). The properties of an object are used to describe an object itself. An real-world example for the properties of an apple would be the color, the weight and the size. Properties, or actually structs, of an UNO service from the Writer application could be font weight, font size and font color.

Examples for services

Figure 4 on the next page illustrates two services in UML notation. The two services offer three interfaces each, and each interface in turn offers several methods. The OfficeDocument closely inspected:

OfficeDocument

- *XPrintable*
    - getPrinter()
    - setPrinter()
    - print()
- *XStoreable*
    - hasLocation()
    - getLocation()
    - isReadOnly()
    - store()
- *XModel*
    - attachResource()
    - getURL()
    - getArgs()

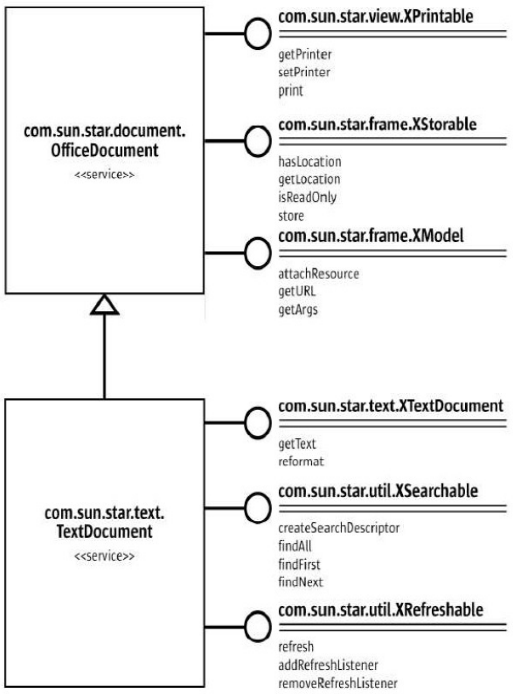


Figure 4: Service example [Flat03]

Note that the TextDocument service is a child of the OfficeDocument service, hence it supports and implements all the interfaces declared by OfficeDocument.

Service Manager

The service manager in OOo is responsible for providing access to various services (objects) from the UNO framework. Before an object can be manipulated, it has to be requested from the service manager. It is used to control and create services, and could be pictured as a factory offering various services:

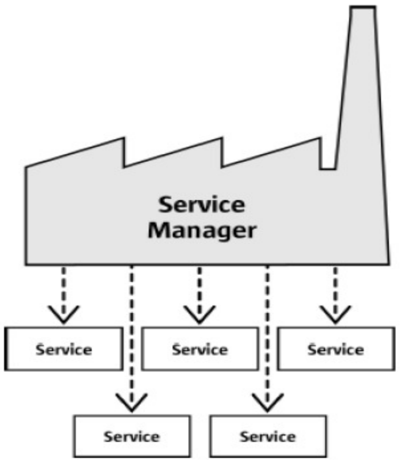


Figure 5: The service manager [Flat03]

It is important to note that in OOo there is no single service manager; instead it is available from various objects. Each document has its own service manager for instance. It is used to manipulate data within the document, and hence only provides access to document-related services. The Writer document service manager, for instance, offers different services than the Calc service manager. Of course, some services and interfaces are supported by more than one document type, for instance the XPrintable interface (to print data) or the XCloseable interface (to close the document). It is important to note that content within a specific document can only be accessed via its respective document service manager.

To get access to services beyond the document's scope, OOo provides access to a global service manager. It is used to create and obtain instances of general UNO services, for example to utilize the FilePicker dialog. As indicated before, the global service manager has no access to objects within a document.

2.3.6 Adding UNO support to Rexx: UNO.CLS

OOo does not support Rexx as a macro language by default - it does, however, support Java. The chapter “BSF4Rexx” explained how the bean scripting framework can be used to bridge Rexx with Java, so utilizing BSF4Rexx there is a possibility to use Rexx to automate OOo applications.

The previous chapter has explained the UNO framework, and Rexx, like any other programming language, has to use this concept in order to access OOo objects. The Viennese version of BSF4Rexx includes a Rexx class called `UNO.CLS` which offers additional UNO support and eases UNO programming considerably.

Key features of UNO.CLS

- Works closely together with BSF4Rexx
- Initializes the OOo session
- Makes it easy to get the XInterfaces of the object
- Offers public routines to inspect/handle UNO objects (e.g. uno.queryInterfaceNames())
- Generalizes interaction at the granular level of "UNO" service objects

[Flat03]

Loading `UNO.CLS` works similar to loading `BSF.CLS`. The most convenient way is to use the `::REQUIRES` directive at the end of the code:

```
/* Adding UNO.CLS functionality */
::REQUIRES UNO.CLS
```

Note that the UNO class already invokes `BSF.CLS`, so loading the `BSF.CLS` separately is not necessary!

The following figure tries to illustrate the communication process from Rexx to OOo. It shows that a number of different layers is used to automate OOo applications via UNO and the Java “bridge”:

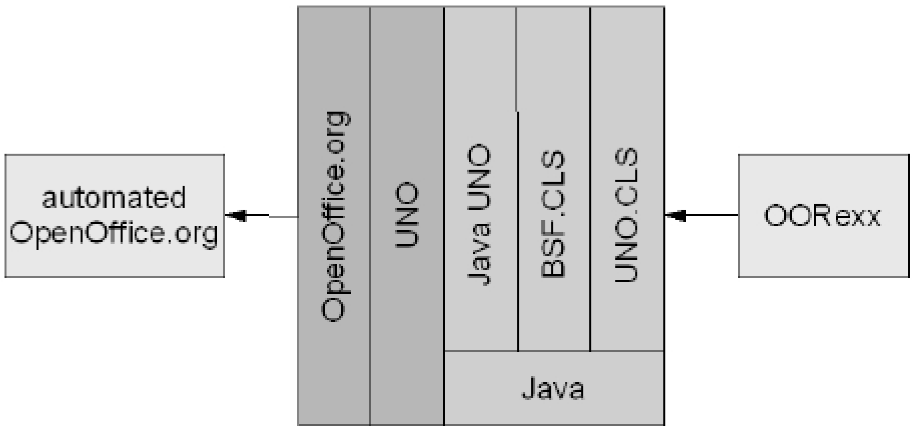


Figure 6: From ooRexx to OpenOffice.org [Augu05] cited by [Aham05]

2.4 Roundup and “Hello World” example:

To summarize the previous chapters, Rexx can be used as a macro language in OOo thanks to several concepts and tools.

First of all, `UNO.CLS` has to be loaded in every script. It invokes BSF4Rexx, the “Java bridge”, and offers control over the UNO services offered by OOo. Rexx can now be used manipulate these services and to automate OOo applications. UNO services are documented in the OOo API (application programming interface), which can be found at http://api.openoffice.org.

The following “Hello World” example, which is automatically pasted into the macro editor when a new ooRexx macro is created[4], is used to explain the UNO concept and components in practice. To run this macro, open an instance of the OOo Writer component and create an empty document.

```
/* Hello World in ooRexx, cf. http://www.ooRexx.org, version: 2007-09-21, */
/* Original version by rgf */
/* Simplified version by Michael Gmeiner */

xScriptContext=uno.getScriptContext() /* get the xScriptContext object */
oDoc=xScriptContext~getDocument -- get the document service (a XModel) object

str="Hello world by http://www.ooRexx.org!" /* define some text */

xTextDoc=oDoc~XTextDocument /* get the XTextDocument interface */
xTextDoc~getText~getEnd~setString(str) /* add text at the end of the text
                                          document */

::requires UNO.CLS /* load UNO support (OpenOffice/StarOffice) for ooRexx */
```

Figure 7: HelloWorld.rex

4 The Hello World snippet has been stripped down to the necessary instructions in for simplicity's sake.

After the first two lines, which are comment lines, the UNO function `getScriptContext()` is used to get a reference to the default script context, which is stored in the variable `xScriptContext`.

From the `xScriptContext`, the `getDocument` method returns the active component (the empty Writer document). This method returns a reference to the document service object, which can now be used to manipulate the active document.

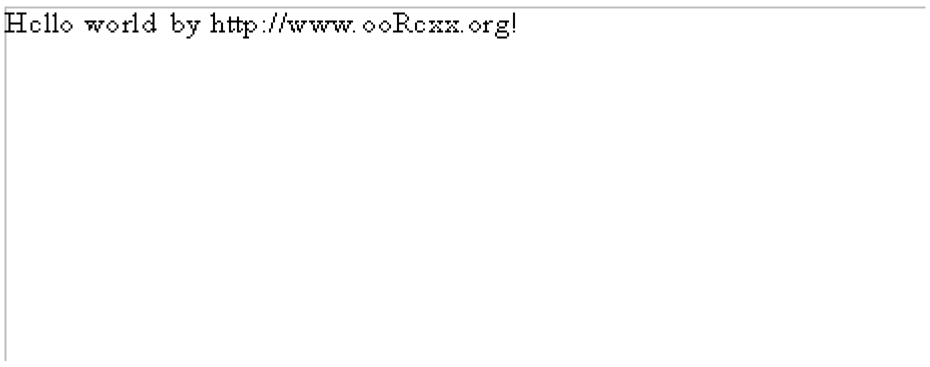
The next line defines a variable, which will be used to store the infamous Hello World text. It is obviously not UNO-related at all, as there is no object involved in this case.

In the next line, the `XTextDocument` interface is retrieved from the current document. Remember that interface names can be identified with the leading “X” in the name. The `XTextDocument` was already discussed in the UNO chapter (“Examples for services”), and offers a method to obtain the text the document contains (`getText()`).

This function is used in the next line to get the text of the document. In order to append text to the end of text, the function `getEnd()` is used, which returns the end of the specified text. Finally, the text contained by the variable str is inserted into the text document using the `setString()` method.

Both functions `getEnd()` and `setString()` are derived from the `XTextRange` interface, which is implemented by the Text service object the function `get-Text()` returned. To get to know which interfaces are supported by a service, either the UNO function `uno.queryInterface()` can be used, or the OOo API reference must be consulted. Example 3.2.1 in the following chapter will provide an example for the use of the function `uno.queryInterface()`.

The last line is needed to enable UNO support in Rexx: the `::REQUIRES` directive loads the class `UNO.CLS`.



```
Hello world by http://www.ooRexx.org!
```

Figure 8: HelloWorld.rex output

3 Examples

3.1 Writer examples

3.1.1 DisplayAllStyles.rex

This example examines the document libraries and displays available all styles. After getting the `xScriptContext` and the active component (document), the `XTextDocument` interface is requested. This interface is needed to get access to text document-specific functions.

```
xxTextDoc = oDoc~XTextDocument
```

The available style families are retrieved via the method `getStyleFamilies` from the `XStyleFamiliesSupplier`. This supplier-concept is very common in the UNO framework. Many objects or properties have to be requested through a supplier interface.

The `getStyleFamilies` function returns a collection of style families. To iterate through the collection the following construction is used:

```
DO style OVER vStyleNames
```

This `DO` loop iterates through all items in the collection. In every cycle an item from the collection `vStyleNames` is assigned to the variable `style`. The names of the elements are determined via the `getElementNames` function. A message box is displayed which usually contains 34 elements. The message box functi-on is provided by BSF4Rexx:

```
.bsf.dialog~MessageBox(s, n)
```

There are three different dialog types available, `MessageBox` (which displays a simple message only), `DialogBox` (which can be customized, e.g. buttons can be defined) and `InputBox` (which requests a user input).

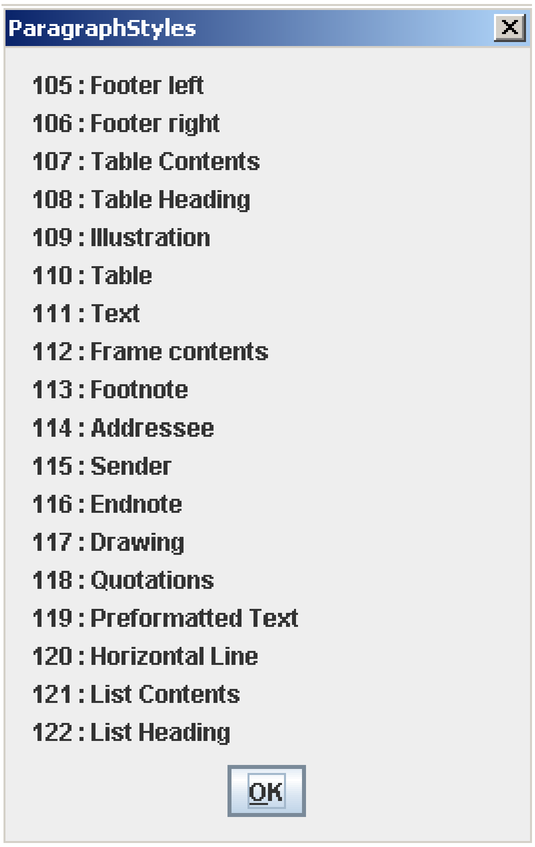
Note that the square brackets can be used to address a single element in an array/a collection, for example:

```
vFamNames[n]
```

returns the n^{th} element of `vFamNames`.

```
/* DisplayAllStyles.rex, version 08-08-22     */
/* ported from OOo Basic as found in Pitonyaks "OOo Macros Explained"    */
/* Listing 12, page 179cf */
/* Rexx code by M. Gmeiner*/

xScriptContext=uno.getScriptContext()     /* get the xScriptContext object */
oDoc=xScriptContext~getDocument -- get the document service (a XModel) object
xTextDoc = oDoc~XTextDocument    -- get the XTextDocument interface

vFamilies = xTextDoc~XStyleFamiliesSupplier~getStyleFamilies
vFamNames = vFamilies~getElementNames    -- get the available styles

DO n OVER vFamNames                -- iterate through the styles
    s = ""

    vStyles = vFamilies~getByName(n)
    xStyleNames = vStyles~XNameAccess
    vStyleNames = xStyleNames~getElementNames

    j = 1
    DO style OVER vStyleNames
        s = s j ":" style "0a"x

        IF ((j + 1) // 35 = 0) THEN -- display 35 elements each time
        DO
            .bsf.dialog~MessageBox(s, n)
            s = ""
        END
        j = j + 1
    END

    IF Length(s) > 0 THEN
        .bsf.dialog~MessageBox(s, n)
END


::requires UNO.CLS /* load UNO support (OpenOffice/StarOffice) for ooRexx    */
```

Figure 9: DisplayAllStyles.rex



```
105 : Footer left
106 : Footer right
107 : Table Contents
108 : Table Heading
109 : Illustration
110 : Table
111 : Text
112 : Frame contents
113 : Footnote
114 : Addressee
115 : Sender
116 : Endnote
117 : Drawing
118 : Quotations
119 : Preformatted Text
120 : Horizontal Line
121 : List Contents
122 : List Heading
```

Figure 10: ParagraphStyles

3.1.2 ShowDocumentSummary.rex

This nutshell can be used to display additional document information. A message box is displayed which contains information like the author of the document and the creation date.

At first a useful concept is used to check whether or not an object offers a certain interface. Using the function `uno.queryInterfaceName()` provided by `UNO.CLS` this can be achieved easily:

```
oDoc~uno.queryInterfaceName("XDocumentInfoSupplier")
```

In the next step, the document info is retrieved. First the filename, which is returned by the oDoc component in URL notation. The function `uno.convertFromUrl()` converts to URL notation to a simple file name.

Afterwards, the document info is obtained using the `XDocumentInfoSupplier`. The properties of the document are accessed via the `getPropertyValue` function, which needs a string argument containing the desired property name.

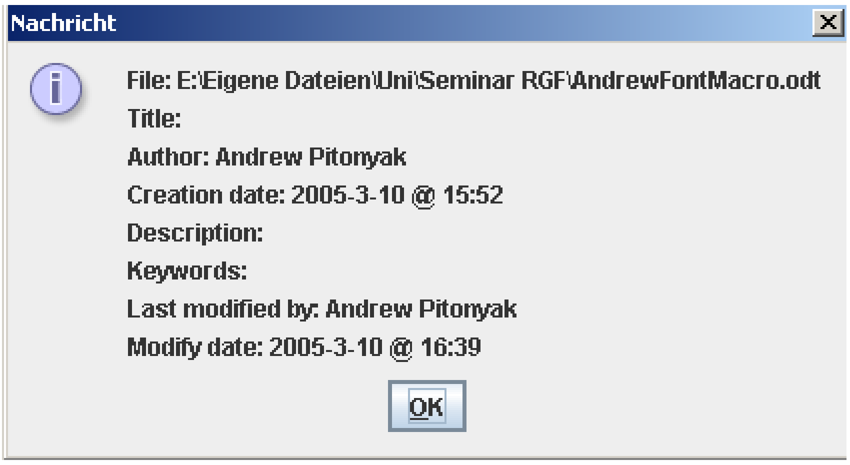
The dates (`creationDate`, `modifyDate`) are no simple data types but structures. Because of that, they cannot be processed like other properties which contain a string only. In order to properly display the date and time, the following method is used:

```
sCreationDate~Hours /* or */ sCreationDate~Minutes -- etc.
```

Finally a message box is displayed containing the retrieved information. Figure 12 shows the document summary of Andrew Pitonyak's macro document ([Pito08]).

```
/* ShowDocumentSummary.rex --- display document info */
/* Based on A. Pitonyak's OOo Basic macro:           */
/* http://www.pitonyak.org/AndrewMacro.odt           */
/* Page 54, Listing 5.45, Chapter 5.18.1             */
/* Rexx code by M. Gmeiner, May 2008                 */

xScriptContext=uno.getScriptContext() /* get the xScriptContext object */
oDoc=xScriptContext~getDocument -- get the document service (a XModel object)

LF = "0a"x -- linefeed control character (Hex

IF (oDoc~uno.queryInterfaceName("XDocumentInfoSupplier") <> "") THEN
DO
    infoString = ""
    infoString = " File:" uno.convertFromUrl(oDoc~getURL) LF  -- the file
                                                              -- name

    .bsf.dialog~messageBox("trying to fetch the document infos...")

    docInfo = oDoc~XDocumentInfoSupplier~getDocumentInfo -- document info
    -- read properties
    infoString = infoString "Title:" -
          docInfo~XPropertySet~getPropertyValue("Title") LF

    infoString = infoString "Author:" -
          docInfo~XPropertySet~getPropertyValue("Author") LF

    sCreationDate = docInfo~XPropertySet~getPropertyValue("CreationDate")

    creationDate = sCreationDate~Year || "-" || -
          sCreationDate~Month || "-" || sCreationDate~Day "@" -
          sCreationDate~Hours || ":" || sCreationDate~Minutes

    infoString = infoString "Creation date:" creationDate LF
    infoString = infoString "Description:" -
         docInfo~XPropertySet~getPropertyValue("Description") LF
    infoString = infoString "Keywords:" -
         docInfo~XPropertySet~getPropertyValue("Keywords") LF
    infoString = infoString "Last modified by:" -
         docInfo~XPropertySet~getPropertyValue("ModifiedBy") LF

    sModifyDate = docInfo~XPropertySet~getPropertyValue("ModifyDate")

    modifyDate = sModifyDate~Year || "-" || sModifyDate~Month || -
         "-" || sModifyDate~Day "@" sModifyDate~Hours || -
         ":" || sModifyDate~Minutes

    infoString = infoString "Modify date:" modifyDate LF
    .bsf.dialog~messageBox(infoString)
END


::requires UNO.CLS   /* load UNO support (OpenOffice/StarOffice) for ooRexx */
```

Figure 11: ShowDocumentSummary.rex



File: E:\Eigene Dateien\Uni\Seminar RGF\AndrewFontMacro.odt
Title:
Author: Andrew Pitonyak
Creation date: 2005-3-10 @ 15:52
Description:
Keywords:
Last modified by: Andrew Pitonyak
Modify date: 2005-3-10 @ 16:39

Figure 12: Document summary

3.1.3 SearchAndFormat.rex

The next macro is used to demonstrate how to perform a search, and additionally how to format text.

Character/font properties are usually defined via constants, for example the font weight. In order to get a constant, the BSF4Rexx function `bsf.getConstant()` is used. As the font weight requires a FLOAT value, the constant has to be converted accordingly using the `box()` function.

The search descriptor has to be retrieved fist using the `XSearchable` interface. Afterwards the search term (`searchString`) is defined. This can be achieved either via an interface and the `setSearchString` method, or directly via the search descriptor:

```
oDescriptor~SearchString = sWord                -- our search term (DIRECT ACCESS)
oDescriptor~XSearchDescriptor~setSearchString(sWord) -- interface access
```

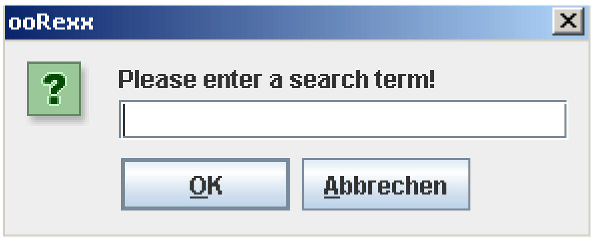
The `findFirst` function tries to find the first occurrence of the search term. If there is matching text in the document, the `findNext` function continues to search. The `.nil` object (which basically stands for “nothing”) is returned in case that nothing is found.

The font properties of the found text are accessed via the `XPropertySet` interface. Using the `setPropertyValue` method, the property `charWeight` is set to bold (using the constant that was retrieved in the beginning)

Finally, the number of occurences found is displayed in a popup (Figure 15).

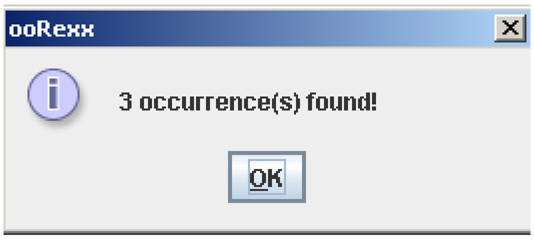
```
/* SearchAndFormat.rex: search a text and set the font weight to BOLD   */
/* Based on A. Pitonyak's OOo Basic macro:                         */
/* http://www.pitonyak.org/AndrewMacro.odt                         */
/* Page 185, Listing 7.41, Chapter 7.14                            */
/* Rexx code by M. Gmeiner, May 2008                               */

xScriptContext=uno.getScriptContext()  /* get the xScriptContext object */
oDoc=xScriptContext~getDocument -- get the document service (a XModel object)
fwBold = box("float", bsf.getConstant("com.sun.star.awt.FontWeight", "BOLD"))

sWord = .bsf.dialog~inputBox("Please enter a search term!", "ooRexx", -
      "question") -- request a search term

-- use the XSearchable-Interface to get a com.sun.star.util.SearchDescriptor
oDescriptor = oDoc~XSearchable~createSearchDescriptor

-- Method 1: our search term (DIRECT ACCESS)
-- oDescriptor~SearchString = sWord

-- Method 2: our search term (INTERFACE ACCESS)
oDescriptor~XSearchDescriptor~setSearchString(sWord)

-- don't search complete words only
oDescriptor~setPropertyValue("SearchWords", .false)

-- search is case insensitive
oDescriptor~setPropertyValue("SearchCaseSensitive", .false)

found = 0
oFound = oDoc~XSearchable~findFirst(oDescriptor)

DO WHILE (oFound <> .nil)
    -- get the property set of the text range
    props = oFound~XTextRange~XPropertySet

    -- set the font weight -> bold
    props~setPropertyValue("CharWeight", fwBold)

    found = found + 1

    -- find the next occurrence
    oFound = oDoc~XSearchable~findNext(oFound~XTextRange~getEnd, oDescriptor)
END

.bsf.dialog~messageBox(found "occurrence(s) found!", "ooRexx", "information")


::requires UNO.CLS   /* load UNO support (OpenOffice/StarOffice) for ooRexx */
```

Figure 13: SearchAndFormat.rex



Please enter a search term!

Figure 14: Input box


ooRexx

3 occurrence(s) found!

Figure 15: Message box

3.1.4 FontSummary.rex

This is a useful macro which creates a sample text in all available fonts installed on the system.

It first requests a sample phrase from the user, which will be formatted in various font styles. Should the user enter no text, a standard phrase will be used instead (`"ooRexx owns OpenOffice. Nice, innit?"`).

The method `lockControllers` in the next line could be used to disable screen updates. If desired, this prevents that the process of filling the document with sample phrases can be masked this way.

The `TextCursor` interface is used to format text. Using the `XPropertySet` interface of the `TextCursor` the font is changed at the position of the text cursor, thus any text entered afterwards will have the new font style.

The available font list is obtained from the `XDevice` interface (function `getFontDescriptors`). The following `DO` loop iterates through the font collection and displays the sample text in all available font styles.

Text is pasted into the document using the `insertString` method provided by the `XText` interface:

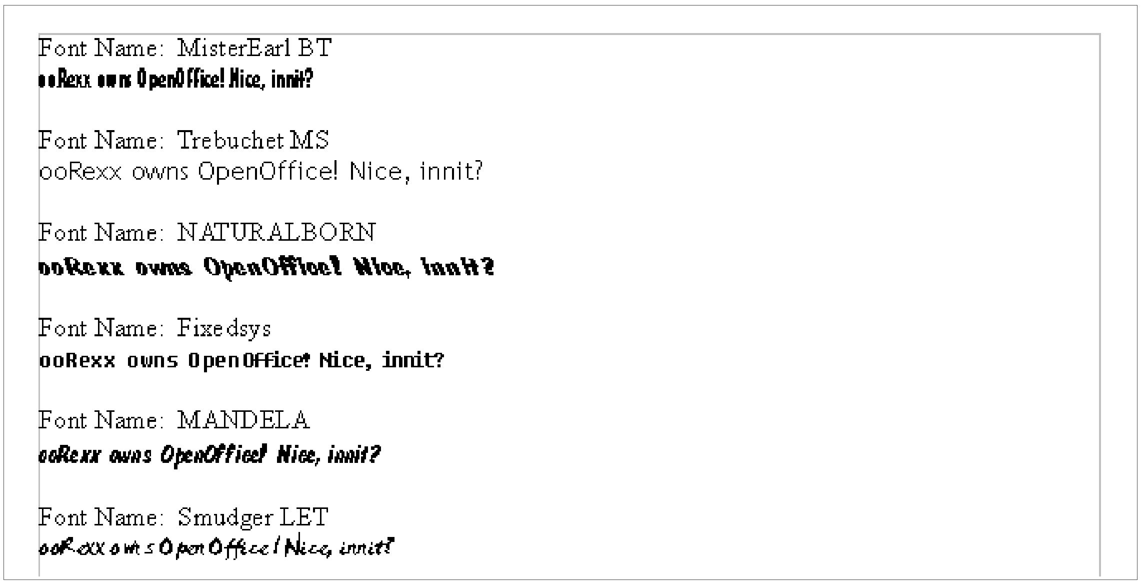
```
xText~insertString(xText~getEnd, "Font Name: " fontName || LF, .false)
```

The first parameter ensures that the text is inserted at the end of the document; the next parameter is the string to be inserted, and the last parameter specifies whether or not the text will be replaced (in case that it is not inserted at the end of the document).

If the controller was locked at the beginning of the macro, it is important to unlock it at the end. Otherwise the document will not be updated (refreshed) properly, resulting in an odd behavior.

```
/* FontSummary.rex  --  displays a sample text in all available fonts   */
/* Based on A. Pytoniak's second macro (Listing 2) found in:             */
/* http://www.pitonyak.org/AndrewFontMacro.odt                          */
/* Rexx code by M. Gmeiner, June 2008                                   */

xScriptContext=uno.getScriptContext()  /* get the xScriptContext object  */
oDoc=xScriptContext~getDocument -- get the document service (a XModel object)

sampleText = .bsf.dialog~ -
    inputBox("Please enter a phrase for the sample text", "ooRexx", -
    "question") -- request a sample text

-- Set a default text if no text is entered by the user
IF sampleText = "" THEN sampleText = "ooRexx owns OpenOffice! Nice, innit?"

-- oDoc~lockControllers /* could be used to freeze the screen */

xText = oDoc~XTextDocument~getText      -- get the XText interface

-- create the TextCursor interface
xCursor=xText~createTextCursorByRange(xText~getEnd)

-- get the ContainerWindow for the XDevice
xWindow = oDoc~getCurrentController~getFrame~getContainerWindow

xDevice = xWindow~XDevice     -- the XDevice knows all available fonts

-- the function getFontDescriptors returns a collection
oFonts = xDevice~getFontDescriptors

-- save the original font
originalFont = xCursor~XPropertySet~getPropertyValue("CharFontName")

LF = "0a"x     -- line feed control character
lastFont = ""

DO font OVER oFonts
    fontName = font~name       -- get the font name

    IF fontName <> lastFont THEN    -- used to eliminate duplicate fonts
    DO
        lastFont = fontName

        xText~insertString(xText~getEnd, "Font Name: " fontName || -
             LF, .false)  -- insert the name

        -- switch font
        xCursor~XPropertySet~setPropertyValue("CharFontName", fontName)

        -- insert the sample text
        xText~insertString(xText~getEnd, sampleText, .false)

        -- back to original font
        xCursor~XPropertySet~setPropertyValue("CharFontName", originalFont)

        -- two line feed characters
        xText~insertString(xText~getEnd, LF || LF, .false)
    END
END

-- oDoc~unlockControllers /* unlock the screen */


::requires UNO.CLS /* load UNO support (OpenOffice/StarOffice) for ooRexx   */
```

Figure 16: FontSummary.rex


Font Name: MisterEarl BT
ooRexx owns OpenOffice! Nice, innit?

Font Name: Trebuchet MS
ooRexx owns OpenOffice! Nice, innit?

Font Name: NATURALBORN
ooRexx owns OpenOffice! Nice, innit?

Font Name: Fixedsys
ooRexx owns OpenOffice! Nice, innit?

Font Name: MANDELA
ooRexx owns OpenOffice! Nice, innit?

Font Name: Smudger LET
ooRexx owns OpenOffice! Nice, innit?

Figure 17: Miscellaneous fonts

3.1.5 InsertBitmap.rex

This code example shows how to insert a bitmap (actually any image file supported by OOo) into a Writer document.

First of all a graphic has to be specified, as always with OOo in URL notation. This is especially handy as it also allows inserting images from a remote location, e.g. a graphic hosted on a web server.

The following line requests the service factory of the document. It is used to create an instance of a `TextGraphicsObject`, which will be used to internally store the graphic.

```
xServiceManager = oDoc~XMultiServiceFactory
oTextGraphic = xServiceManager -
      ~createInstance("com.sun.star.text.TextGraphicObject")
```

The object's properties are accessed via the already known `XPropertySet` interface. It should be clear by now that all objects supporting the `XPropertySet` interface behave the same (ie. offer the same functions to get and set properties). The first property ("GraphicURL") contains the URL of the file. The `AnchorType` property determines how a graphic is anchored. Using the value "`AS_CHARACTER`" the graphic will be inserted like a character at the desired cursor position. Other anchor types would result in a centered image across the paragraph, etc.

In the next step the image is inserted at the text cursor position. Finally, the XGraphicObjectsSupplier interface from the current document is used in order to get access to the graphic objects in the document. The number of graphics in the document is displayed using a message box.

```
/* InsertBitmap.rex  --- how to insert a bitmap into a Writer document  */
/* loosely based on Christian Lohmaier's Java snippet found at          */
/* http://codesnippets.services.openoffice.org/Writer/ ~                */
/*   Writer.EmbedAGraphicIntoATextdocument.snip                         */
/* Rexx code by M. Gmeiner, June 2008                                   */

xScriptContext=uno.getScriptContext()  /* get the xScriptContext object */
oDoc=xScriptContext~getDocument -- get the document service (a XModel object)

-- Web source, requires an active internet connection!
sGraphicURL = "http://api.openoffice.org/branding/images/logonew.gif"
xServiceManager = oDoc~XMultiServiceFactory  -- get the document's
                                             -- service manager

-- create a new graphic object which contains the image
oTextGraphic = xServiceManager -
    ~createInstance("com.sun.star.text.TextGraphicObject")

xGraphicProperties = oTextGraphic~XPropertySet  -- get the object's properties

-- set the image URL and the anchor type
xGraphicProperties~setPropertyValue("GraphicURL", sGraphicURL)
xGraphicProperties~setPropertyValue("AnchorType", -
   bsf.getConstant("com.sun.star.text.TextContentAnchorType", "AS_CHARACTER"))

xText = oDoc~XTextDocument~getText      -- get the XText interface
xCursor=xText~createTextCursor()        -- create the TextCursor interface

-- insert the TextContent of the image
-- at the TextCursor position
xText~insertTextContent(xCursor, oTextGraphic~XTextContent, .false)

-- get the GraphicObjects collection
graphicObjects = oDoc~XTextGraphicObjectsSupplier~getGraphicObjects

-- display the number of images in the collection
.bsf.dialog~messageBox("Number of graphic objects:" -
    graphicObjects~XIndexAccess~getCount)


::requires UNO.CLS /* load UNO support (OpenOffice/StarOffice) for ooRexx   */
```

Figure 18: InsertBitmap.rex

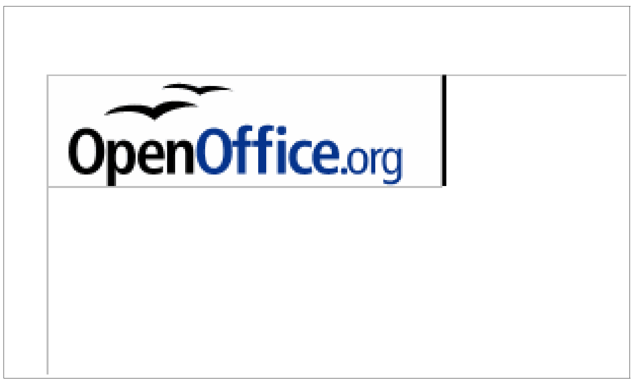


Figure 19: Web graphic inserted

3.1.6 FilePicker.rex

This example uses the FilePicker dialog to open a Writer document.

First of all the service manager is requested. Note that this time not the document's service manager is used, but the `xContext` service manager instead. As explained before, the document service manager can only offer document-related services. Using the global service manager an instance of the `FilePicker` object is created:

```
sm = xContext~getServiceManager /* get the service manager  */

-- get FilePicker service
s_FileDialog = -
     sm~createInstanceWithContext("com.sun.star.ui.dialogs.FilePicker", -
     xContext)
```

Via the `XFilterManager` the filter is set in order to display Writer documents only (*.odt). Furthermore the title of the dialog is set, and the `MultiSelection-Mode` is set to false, hence only one file can be selected only.

The execute function displays the file dialog. As it is modal, the code has to wait until the dialog is closed again. The function will return either `TRUE` if a file has been selected or `FALSE` in case that no file was selected (usually the user pressed 'Cancel'). If a file has been selected, the macro tries to open the file via the `XComponentLoader`. The `newDoc` variable is used to store the reference to the new document. Should OOo be unable to open the file (a user could still select an other file type than *.odt, even with the filter set properly) a `NIL` object is returned.

```
/* FilePicker.rex  -- open a Writer file using the FilePicker dialog    */
/* Based on a Rexx snippet by R. G. Flatscher                           */
/* Example by M. Gmeiner, June 2008                                     */

xScriptContext=uno.getScriptContext()  /* get the xScriptContext object */
oDoc=xScriptContext~getDocument -- get the document service (a XModel object)
xDesktop=xScriptContext~getDesktop     /* get the desktop (XDesktop) object */
xContext=xScriptContext~getComponentContext  /* get the context (a
                                              XComponentContext) object */

sm = xContext~getServiceManager /* get the service manager  */
-- get FilePicker service
s_FileDialog = -
    sm~createInstanceWithContext("com.sun.star.ui.dialogs.FilePicker", -
    xContext)

x_FileDialog = s_FileDialog~XFilePicker     -- get XFilePicker interface
x_FileDialogFilters = s_FileDialog~XFilterManager  -- get XFilterManager int.
x_FileDialogFilters~appendFilter("ODT *.odt", "*.odt") -- set file extension
x_FileDialog~setTitle("Select OOo Writer document") -- set title
-- allow multi selection mode (but in this example we want a single file only)
-- x_FileDialog~setMultiSelectionMode(.true)
x_FileDialog~setMultiSelectionMode(.false)   -- dis-allow multi selection mode
```

```
IF x_FileDialog~execute = .true THEN
DO
    files = x_FileDialog~getFiles
    odtFile = files[1] -- only one file could be selected!

    -- try to open the file
    .bsf.dialog~messageBox("file chosen:" uno.convertFromUrl(odtFile))

    newDoc = xDesktop~XComponentLoader~LoadComponentFromUrl(odtFile, -
       "_blank", 0, .uno~noProps)

    IF (newDoc = .nil) THEN
        .bsf.dialog~messageBox("Sorry - could not open" -
        uno.convertFromUrl(odtFile) || "!")
END ELSE
DO
    .bsf.dialog~messageBox("no file chosen!")
END


::requires UNO.CLS /* load UNO support (OpenOffice/StarOffice) for ooRexx   */
```

Figure 20: FilePicker.rex

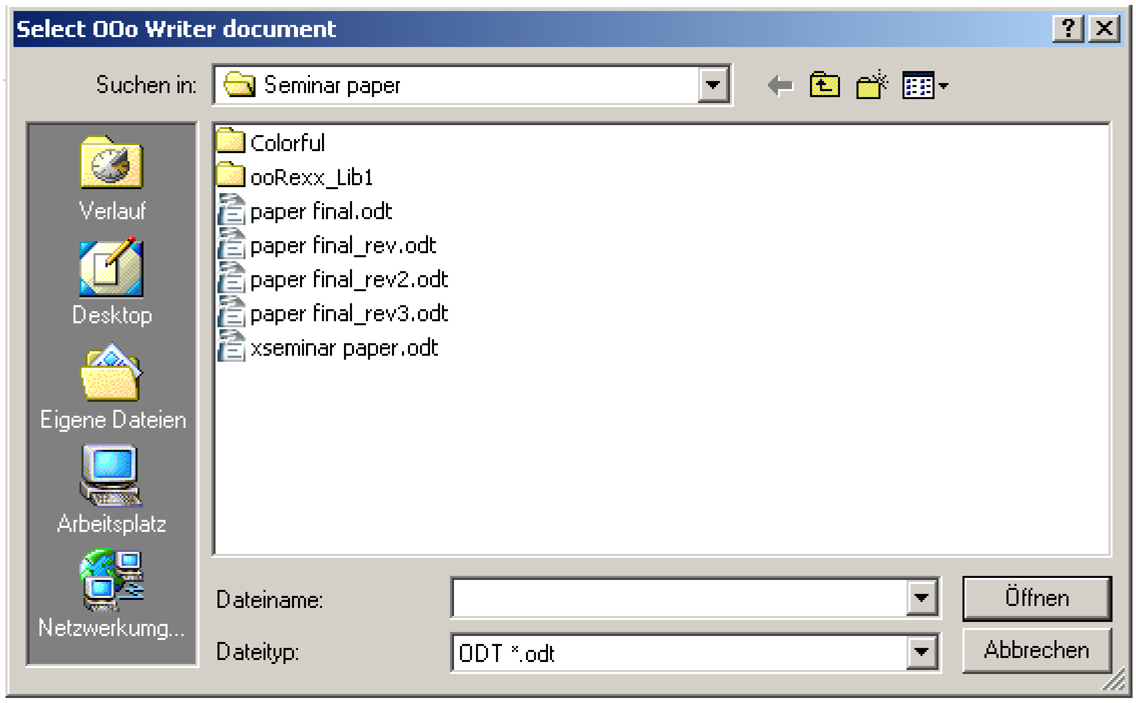


Figure 21: File dialog

3.1.7 StatusIndicator.rex

This code snippet explains how to write text to the status bar using the StatusIndicator service.

In order to get to the desired `XStatusIndicatorFactory` interface, the `XFrame` interface has to be requested from the current document (`oDoc`). The `Sta-`

`tusIndicator` is now created using the method `createStatusIndicator` offered by the `XStatusIndicatorFactory` interface.

In order to display text for the first time, the start method has to be used. It aspects two arguments, the first one contains the text to be displayed, the second argument (numerical) can be used to set the status (progression) bar. So for example if the macro calculates or processes some data, the progression can be displayed.

Once the `StatusIndicator` has been started, further text can only be displayed via the `setText` method. Another call of the start method would not work, unless the `StatusIndicator` service has been stopped using the end method before.

```
/* StatusIndicator.rex  -- display text in the status bar              */
/* Original OOo Basic macro found in A. Pitonyak's AndrewMacros.odt:     */
/* http://www.pitonyak.org/AndrewMacro.odt                              */
/* Page 27, Listing 5.1 (Chapter 5.1)                                   */
/* Original author: Sasa Kelecevic                                      */
/* Rexx code by M. Gmeiner, May 2008                                    */

xScriptContext=uno.getScriptContext()  /* get the xScriptContext object  */
oDoc=xScriptContext~getDocument -- get the document service (a XModel) object
xDesktop=xScriptContext~getDesktop     /* get the desktop (XDesktop) object */
xFrame = oDoc~getCurrentController~getFrame     -- get the XFrame interface

-- this interface lets us create an Indicator
xStatusIndicatorFactory = xFrame~XStatusIndicatorFactory
xStatusIndicator = xStatusIndicatorFactory~createStatusIndicator
xStatusIndicator~start("Hello from ooRexx!       ", 0) -- start the service
xStatusIndicator~setText("...some more text!")     -- display more text


::requires UNO.CLS /* load UNO support (OpenOffice/StarOffice) for ooRexx   */
```

Figure 22: StatusIndicator.rex

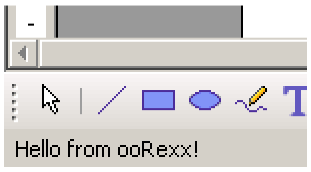

Hello from ooRexx!

Figure 23: Hello from ooRexx

3.2 Calc macros

3.2.1 CellValues.rex

The first Calc macro demonstrates how to actually address cells and read their contents.

First of all the sheets collection from the current document is obtained via the `XSpreadsheetDocument` interface (using the function `getSheets`). In order to access individual sheets one possibility would be via the sheet's name. The names can be requested using the `getElementNames` function which returns an array of all sheet names. In this macro we simply choose the **first** sheet of the document:

```
xSheets = oDoc~XSpreadsheetDocument~getSheets
sheetNames = xSheets~getElementNames
xSheet1 = xSheets~getByName(sheetNames[1])~XSpreadSheet
```

There are several methods to access a certain cell. This macro shows four ways how to obtain a cell object:

```
xCell = xSheet1~getCellByPosition(0, 0) -- method 1
xCell = uno.getCell(xSheet1, 0, 0)      -- method 2 provided by UNO.CLS
xCell = xSheet1~getCellRangeByName("A1")-- method 3, alphanumeric address
xCell = uno.getCell(xSheet1, "A1")      -- method 4 provided by UNO.CLS
```

Method one and three are via the `XSheet` interface, either via a numerical address (format col, row) or via an alphanumerical address usually preferred by users (format “A1B1”). Method two and four are shortcuts provided by `UNO.CLS`. Internally the cells are addressed via the `XSheet` interface, but it saves few characters and makes life easier.

A cell can have either of four different cell types:

TYPE	Contains	Functions
TEXT	Standard text (strings), not processed	getString (XText interface)
VALUE	Numerical values	getValue
FORMULA	Mathematical formulae/expressions	getFormula, getValue
EMPTY	Empty cell	-

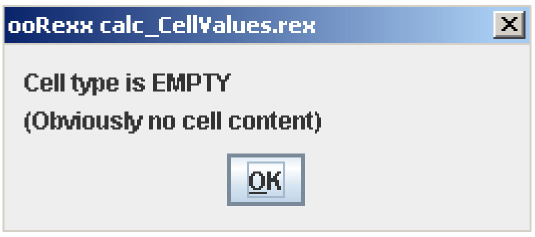
Figure 24: Cell Types

These cell types are obtained via the `bsf.getConstant` function which looks up the respective constant value. The cell type is needed in order to process the content of the cell accordingly. Cells with text values cannot be used for mathematical operations for instance. Note that FORMULA cells can both return the value of the formula as well as the formula itself (eg. “`SUM(A1:B1)`”)

In order to test the cell type against the obtained constants, the function `UNO.areSame` is used. A regular comparison via the “`=`” operator will not work as the two operands are not compared by value in this case; instead, the references are compared.

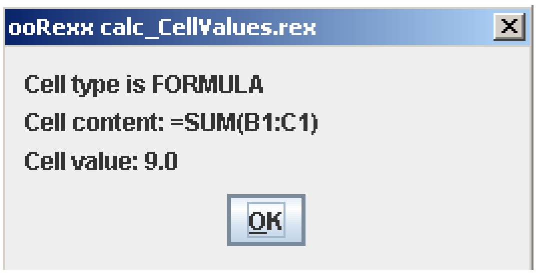
```
/* CellValues.rex  --- read cell values and determine the cell type     */
/* Loosely based on Sasa Kelecevic' example found in AndrewMacro.odt:   */
/* http://www.pitonyak.org/AndrewMacro.odt                              */
/* Page 117, Listing 6.3 (Chapter 6.2)                                  */
/* Rexx code by M. Gmeiner, May 2008                                    */

xScriptContext=uno.getScriptContext() /* get the xScriptContext object */
oDoc=xScriptContext~getDocument       /* get the document service (a XModel)
                                         object */

xSheets = oDoc~XSpreadsheetDocument~getSheets
sheetNames = xSheets~getElementNames

-- select the first sheet of the active document
xSheet1 = xSheets~getByName(sheetNames[1])~XSpreadSheet

xCell = xSheet1~getCellByPosition(0, 0) -- method 1
xCell = uno.getCell(xSheet1, 0, 0)      -- method 2 provided by UNO.CLS
xCell = xSheet1~getCellRangeByName("A1")-- method 3, alphanumeric address
xCell = uno.getCell(xSheet1, "A1")      -- method 4 provided by UNO.CLS

cellType = xCell~getType     -- determine the cell type (text, value, formula)


/* Define the cellType values */
enumTEXT  = bsf.getConstant("com.sun.star.table.CellContentType", "TEXT")
enumVALUE = bsf.getConstant("com.sun.star.table.CellContentType", "VALUE")
enumFORMULA = bsf.getConstant("com.sun.star.table.CellContentType", "FORMULA")
enumEMPTY   = bsf.getConstant("com.sun.star.table.CellContentType", "EMPTY")

LF = "0a"x -- line feed control character

IF uno.areSame(enumTEXT, cellType) THEN
    msg = "Cell type is TEXT" || LF || "Cell content:" xCell~XText~getString
ELSE IF uno.areSame(enumVALUE, cellType) THEN
    msg = "Cell type is VALUE" || LF || "Cell content:" xCell~getValue
ELSE IF uno.areSame(enumFORMULA, cellType) THEN
    msg = "Cell type is FORMULA" || LF || "Cell content:" -
        xCell~getFormula || LF || "Cell value:" xCell~getValue
ELSE IF uno.areSame(enumEMPTY, cellType) THEN
    msg = "Cell type is EMPTY" || LF || "(Obviously no cell content)"
ELSE -- should never arrive here...
    msg = "Unknown cell type!"

.bsf.dialog~messageBox(msg, "ooRexx calc_CellValues.rex")


::requires UNO.CLS /* load UNO support (OpenOffice/StarOffice) for ooRexx   */
```

Figure 25: CellValues.rex


ooRexx calc_CellValues.rex

Cell type is EMPTY
(Obviously no cell content)

Figure 26: Empty cell


ooRexx calc_CellValues.rex

Cell type is FORMULA
Cell content: =SUM(B1:C1)
Cell value: 9.0

Figure 27: Cell with a formula

3.2.2 FillCells.rex

This next short example demonstrates how to fill cells with values. It uses the same mechanism to access sheets and address a single cell like in the previous example.

The user is asked to enter some data for the cell, using the `inputBox` function provided by BSF4Rexx. The data is afterwards stored in cell A1 using the following `UNO.CLS` shortcut:

```
CALL uno.setCell xSheet1, "A1", s
```

This method puts the value of the third parameter (`s` in this case) into the sheet object `xSheet1` (first parameter) into the cell A1 (second parameter)

```
/* FillCell.rex  -- fill a cell with a value entered by the user       */
/* Based on Sasa Kelecevic' OOo Basic macro found in AndrewMacro.odt   */
/* http://www.pitonyak.org/AndrewMacro.odt                             */
/* Page 118, Listing 6.4 (Chapter 6.3)                                 */
/* Rexx code by M. Gmeiner, May 2008                                   */

xScriptContext=uno.getScriptContext()  /* get the xScriptContext object */
oDoc=xScriptContext~getDocument -- get the document service (a XModel) object
xSheets = oDoc~XSpreadsheetDocument~getSheets
sheetNames = xSheets~getElementNames

/* select the first sheet of the active document */
xSheet1 = xSheets~getByName(sheetNames[1])~XSpreadSheet

s = .bsf.dialog~inputBox("Enter some data for cell A1!")

CALL uno.setCell xSheet1, "A1", s       -- function provided by UNO.CLS


::requires UNO.CLS /* load UNO support (OpenOffice/StarOffice) for ooRexx   */
```

Figure 28: FillCells.rex

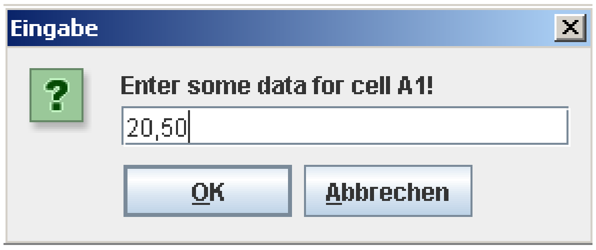


Figure 29: Data input

3.2.3 NamedRanges.rex

Named ranges are a useful concept both for programming macros as well as for using the front-end of the spreadsheet application. It allows to label (name) certain cells or ranges and access them via the given name. This makes life much easier for both programmers as well as end users, especially because named ranges shift with their contents. So one does not need to bother with absolute cell ranges like “A1:D20”, but can address this range with a name like “Sales 2007”.

The interface `XNamedRanges`, which provides all necessary functions, can be obtained from the current component (document). In order to define a new named range, the function `addNewByName` is used:

```
xNamedRanges~addNewByName("Name", "content", aCellAddress, 0)
```

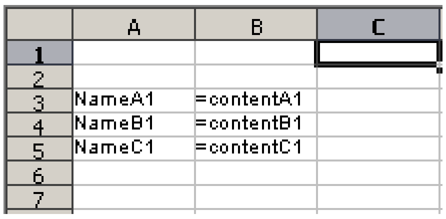
This would create a new named range with the name “`Name`”, a content “`content`” and a cell address `aCellAddress` (which must have a `XCellAddress` format).

This example creates three named ranges, and outputs a list of all named ranges into the document. The method used for this is again offered by `XNamedRanges` and is called `outputList`. It only needs a cell address which will be used to contain the list of named ranges.

```
/* NamedRangeDemo.rex    --- demonstrates the basics of named ranges     */
/* Idea from A. Pitonyak's OOo Basic macro found in AndrewMacros.odt     */
/* http://www.pitonyak.org/AndrewMacro.odt                               */
/* Page 134, Listing 6.29 (Chapter 6.19)                                 */
/* Rexx code by M. Gmeiner, May 2008                                     */

xScriptContext=uno.getScriptContext()  /* get the xScriptContext object  */
oDoc=xScriptContext~getDocument -- get the document service (a XModel) object
xSheets = oDoc~XSpreadsheetDocument~getSheets
sheetNames = xSheets~getElementNames
```

```
-- select the first sheet of the active document
xSheet1 = xSheets~getByName(sheetNames[1])~XSpreadSheet
oNamedRanges = oDoc~XPropertySet~getPropertyValue("NamedRanges")
xNamedRanges = oNamedRanges~XNamedRanges   -- get the XNamedRanges interface

/* define three named ranges (A1, B1, C1) */
xCell = uno.getCell(xSheet1, "A1")
a = xCell~XCellAddressable~getCellAddress

-- add the first named range
xNamedRanges~addNewByName("NameA1", "contentA1", a, 0)
xCell = uno.getCell(xSheet1, "B1")
a = xCell~XCellAddressable~getCellAddress

-- add the second named range
xNamedRanges~addNewByName("NameB1", "contentB1", a, 0)
xCell = uno.getCell(xSheet1, "C1")
a = xCell~XCellAddressable~getCellAddress

-- add the third named range
xNamedRanges~addNewByName("NameC1", "contentC1", a, 0)

/* dump the list of named ranges into the sheet, starting at position A3 */
xCell = uno.getCell(xSheet1, "A3")
a = xCell~XCellAddressable~getCellAddress
xNamedRanges~outputList(a)


::requires UNO.CLS /* load UNO support (OpenOffice/StarOffice) for ooRexx   */
```

Figure 30: NamedRanges.rex



	A	B	C
1			
2			
3	NameA1	=contentA1	
4	NameB1	=contentB1	
5	NameC1	=contentC1	
6			
7			

Figure 31: Three named ranges

3.2.4 ProtectSheets.rex

Protecting sheets is a nice way to prevent unintentional changes, or even changes made by purpose (manipulation). It is also a nice way to control the contents through the macro, as it can protect and unprotect the sheets at will (assuming that the password is known), whereas the “typical user” has no clue on how to unlock the sheet. Of course the password is stored within the macro, but most users do not know how to access the macro code, let alone interpret it. Especially if the password variable is not declared in an obvious way, or if a function generates the password from an algorithm, this method of protecting sheets is actually quite safe and proven.

In this example, after the password has been set (in a very unsafe way), the user is prompted to select a sheet from the document. The function `.bsf.dialog.inputBox` is used again, this time with an additional parameter: the sheet names, contained in an array variable. The `inputBox` function will handle this parameter in a way that it creates a list of items from the contents of the array. The user will see a list of available sheet names and can select one of it.

```
sheetName = .bsf.dialog~inputBox("Choose a sheet to protect the contents", -
        "ProtectSheets.rex", "OkCancel", , sheetNames)
```

Setting the password is rather easy: select the sheet using its name and use the `XProtectable` interface to set the password, using the `protect` method.

Finally, the status of the sheet is checked in respect of the protection: the function `isProtected` returns true or false depending on the protection status.

```
/* ProtectSheets.rex -- protect single sheets from a spreadsheet document */
/* Based on A. Pitonyak's OOo Basic macro found in AndrewMacro.odt        */
/* http://www.pitonyak.org/AndrewMacro.odt                                */
/* Page 132 (Chapter 6.16)                                                */
/* Rexx implementation by M. Gmeiner, June 2008                           */

xScriptContext=uno.getScriptContext()  /* get the xScriptContext object   */
oDoc=xScriptContext~getDocument -- get the document service (a XModel) object

xSheets = oDoc~XSpreadsheetDocument~getSheets
sheetNames = xSheets~getElementNames            -- get all sheet names

password = "trustno1"                           -- set a password

--  display an inputBox with the sheet names as option values
sheetName = .bsf.dialog~inputBox("Choose a sheet to protect the contents", -
    "ProtectSheets.rex", "OkCancel", , sheetNames)

IF sheetName <> "" THEN  -- if a sheet was selected, protect the sheet!
DO
    xSheet = xSheets~getByName(sheetName)~XSpreadSheet
    xSheet~XProtectable~protect(password)

    -- check the 'protected'-status:
    IF xSheet~XProtectable~isProtected THEN
        .bsf.dialog~messageBox("Sheet '" || sheetName || "' protected!")
END


::requires UNO.CLS /* load UNO support (OpenOffice/StarOffice) for ooRexx   */
```

Figure 32: ProtectSheets.rex

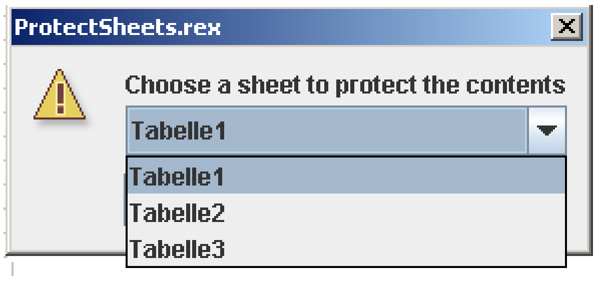


Figure 33: Sheet selection list

3.2.5 SelectRange.rex

This macro example shows how to select a range in a spreadsheet document. Additionally, is explains how to determine the used area of the sheet (ie, the range that contains data). This is useful for example for copying all the data in a sheet, if the whole sheet cannot be copied for certain reasons.

Again in this example, the first sheet is used, and a cell object is needed in order to get the `XCursor` interface, which implements the function `gotoEndOfUsedArea` from the `XUsedAreaCursor` interface:

```
xCursor~XUsedAreaCursor~gotoEndOfUsedArea(.true)
```

This method places the (invisible) cursor to the last used cell, and it's address can now be extracted. The parameter `.true` specifies that the cursor should span the whole range from the start cell to the last used cell.

In order to select the determined range, a range object is created with the addresses of the first and last cell. This range object is afterwards passed on to the `XSelectionSupplier`, which is used to select data in a document:

```
xSelection = oDoc~getCurrentController~XSelectionSupplier
xSelection~select(xRange)
```

The method select is used to select the range, which now contains all the data of the sheet. The selection could now be copied to the clipboard for instance, or processed otherwise.

```
/* SelectRange.rex  --- select the used area in a spreadsheet          */
/* Based on a Basic macro found in Pitonyak's AndrewMacro.odt submitted  */
/* by Gerrit Jasper:                                                     */
/* http://www.pitonyak.org/AndrewMacro.odt                              */
/* Page 138 (Chapter 6.19)                                               */
/* Rexx code written by M. Gmeiner, June 2008                           */

xScriptContext=uno.getScriptContext() /* get the xScriptContext object */
oDoc=xScriptContext~getDocument -- get the document service (a XModel) object
xSheets = oDoc~XSpreadsheetDocument~getSheets
```

```
sheetNames = xSheets~getElementNames

-- select the first sheet of the active document
xSheet1 = xSheets~getByName(sheetNames[1])~XSpreadSheet

xCell = uno.getCell(xSheet1, 0, 0) -- get a cell object

-- create a cursor
xCursor = xSheet1~createCursorByRange(xCell~XSheetCellRange)
xCursor~XUsedAreaCursor~gotoEndOfUsedArea(.true) -- obtain the used area

-- extract the address
address = xCursor~XCellRangeAddressable~getRangeAddress

/* specify a range object with the determined address */
xRange = xSheet1~getCellRangeByPosition(address~startColumn, -
    address~startRow, address~endColumn, address~endRow)

-- select the range via the supplier
xSelection = oDoc~getCurrentController~XSelectionSupplier
xSelection~select(xRange)


::requires UNO.CLS /* load UNO support (OpenOffice/StarOffice) for ooRexx */
```

Figure 34: SelectRange.rex

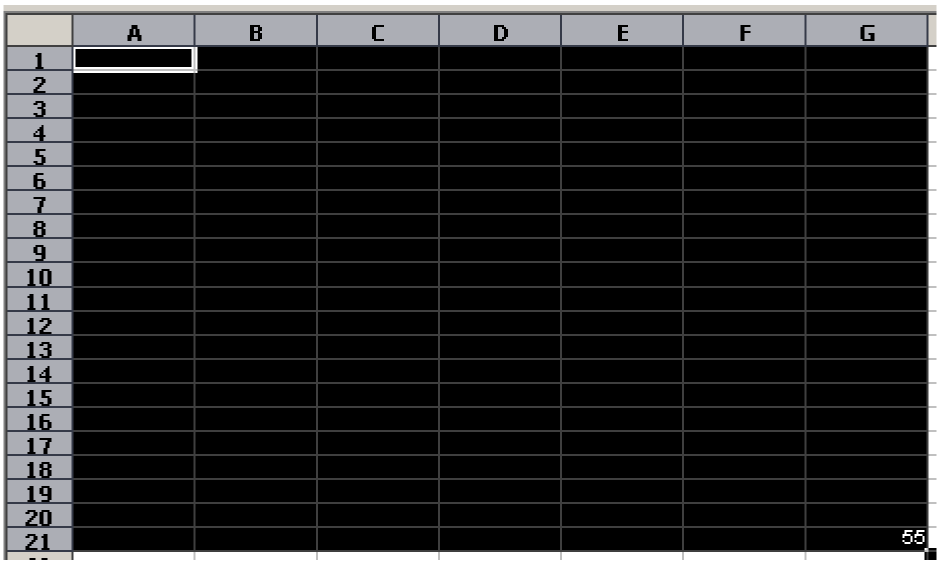


Figure 35: Cell selection

3.3 Impress Example

3.3.1 CustomPresentation.rex

The last example of this paper demonstrates how to set up a custom presentation using the Impress application. It will also show how to address individual slides, and how to select (jump to) slides.

First of all the XCustomSupplier interface is used to request the custom presentation service from the current document. It contains all custom presentations already stored in the document and can be used to define new ones.

Access to the individual slides of the presentation is gained via the XDraw-PagesSupplier interface: it returns a collection of all slides.

In the next step two custom presentation are defined. To avoid double definitions, the current presentation is checked for existing custom presentations. Note that the "\" operator is used to negate the boolean expression returned by the hasName function (true becomes false and vice versa)! A new custom presentation is inserted using the createInstance method provided by the XSingleServiceFactory interface.

```
IF \oPresentations~hasByName("backwards") THEN
    xCustom = oPresentations~XSingleServiceFactory~createInstance
```

For the first presentation, all slides are defined in backward order. The DO loop iterates through all slides and adds them to the presentation accordingly. A slide is added using the insertByIndex method:

```
xCustom~XIndexContainer~insertByIndex(<index>, <draw page>)
```

An other possibility would be to insert a slide via a name, but in order to ensure the correct sequence of the slides, the index-function seemed more reliable.

The actual presentation is informed about the custom selection via a property of the presentation. If the value "CustomShow" contains a valid show (actually the name of the show), it will be used for the next presentation.

Before the presentation starts, the user is asked to select a custom presentation. This time the inputBox is used again to display a number of options (the two presentation names in this case).

Another important step is to select the first slide of the presentation before it starts to run. Should the last slide of the selected presentation be selected, which would be slide #1 for the "backwards" presentation, the show will end after this very slide.

To select a slide, the XSelectionSupplier is used again. The first slide is handed over to the select method, which selects the slide

```
s0 = selectedCustom~XIndexContainer~getByIndex(0)

/* ... */

xSelection~select(s0)
```

Finally, the presentation is started with the method `rehearseTimings`, which displays a stopwatch in order to check the timing of each slide.

```
/* CustomPresentation.rex  -- setting up a custom presentation          */
/* Based on A. Pitonyak's OOo Basic macro found in his book             */
/* "OpenOffice.org Macros Explained", listing 30, page 402cf            */
/* Rexx code by M. Gmeiner, June 2008                                   */

xScriptContext=uno.getScriptContext()  /* get the xScriptContext object */
oDoc=xScriptContext~getDocument -- get the document service (a XModel) object

-- get the custom presentation service
oPresentations = oDoc~XCustomPresentationSupplier~getCustomPresentations

oPages = oDoc~XDrawPagesSupplier~getDrawPages   -- get all slides

IF \oPresentations~hasByName("backwards") THEN
-- insert the custom presentation called "backwards"
DO
    xCustom = oPresentations~XSingleServiceFactory~createInstance
    j = oPages~getCount - 1

    DO i = 0 TO (oPages~getCount - 1)
    -- add all slides starting from the last one
        xCustom~XIndexContainer~insertByIndex(i, oPages~getByIndex(j)
        j = j - 1
    END

    oPresentations~insertByName("backward", xCustom)
    .bsf.dialog~messageBox("Presentation 'backward' added!")
END

IF \oPresentations~hasByName("simple") THEN
-- insert the custom presentation "simple"
DO
    xCustom = oPresentations~XSingleServiceFactory~createInstance

    DO i = 0 TO 4   -- add only the first 5 slides
        xCustom~XIndexContainer~insertByIndex(i, oPages~getByIndex(i))
    END

    oPresentations~insertByName("simple", xCustom)
    .bsf.dialog~messageBox("Presentation 'simple' added!")
END


/* let the user select the custom presentation */
custom = .bsf.dialog~inputBox("Please select a custom presentation", -
    "CustomPresentation.rex", "question", , "backwards simple")

IF custom <> "" THEN DO -- start the presentation
    oPres = oDoc~XPresentationSupplier~getPresentation

     -- add the CustomShow property
    oPres~XPropertySet~setPropertyValue("CustomShow", custom)
     selectedCustom = oPresentations~XNameAccess~getByName(custom)

    -- get the first slide
    s0 = selectedCustom~XIndexContainer~getByIndex(0)

    -- select the range via the supplier
    xSelection = oDoc~getCurrentController~XSelectionSupplier
    xSelection~select(s0)
    oPres~rehearseTimings   -- start the presentation in training mode
END


::requires UNO.CLS /* load UNO support (OpenOffice/StarOffice) for ooRexx   */
```

Figure 36: CustomPresentation.rex

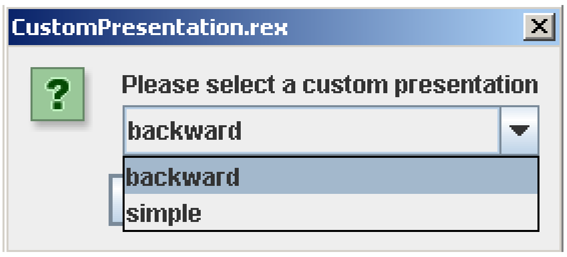


Figure 37: Presentation selection

Figure 38: Rehearsal timer

4 Conclusion

Examining the problem statement again, this paper has clearly demonstrated how to use ooRexx to automate OpenOffice.org: Using concepts like UNO and BSF the bridge between ooRexx and OpenOffice.org has been spanned, hence ooRexx *can* be used to write OpenOffice.org macros. All components which were used are open source and thus freely available, and the cross-platform obligation has also been met.

The provided examples gave a broad overview on how to automate various applications (Writer, Calc, Impress) and furthermore gave an insight into the application of the UNO framework. Fundamentals of the framework, like services, interfaces and structures have been explained both theoretically as well as practically.

Most macro examples are based on OOo Basic nutshells, which were found to be a good guideline on macro programming. As OOo Basic offers various shortcuts and sometimes follows different approaches, however, OOo Basic code most often cannot be directly ported to ooRexx. One crucial difference between the two programming languages is how functions and methods are accessed. In OOo Basic the appropriate interface is usually automatically provided, which eases coding quite a lot. Using ooRexx all interfaces have to be requested first in order to access their functions and methods. Figuring out the complex interface structure of UNO services is a tedious and time consuming task, however, programming “the hard way” eases portability as all other programming languages also require a dedicated access via interfaces too. Porting OOo macros from other languages, like Java, is much easier because of the common use of interfaces. Example 3.1.5 for instance is based on a Java snippet, and as the proper interfaces can be already found in the code, no additional API reference is needed in order to port the snippet.

Working with OpenOffice.org internals yields a steep learning curve, the complex architecture of the UNO framework however slows down the process of creating macros considerably, especially in the beginning phase, where the huge UNO concept can seem intimidating and confusing. Also lacking a "proper" IDE for ooRexx macros with class/object browsers etc. means that the programmer constantly has to consult the UNO API documentation in order to find the right mechanisms (services, interfaces, properties) needed to manipulate objects.

`UNO.CLS`, on the other hand, offers many useful functions and tools which aid the programmer, but the time consuming and often tedious task to find out the required interfaces and methods needed to solve a programming task accordingly still has to performed by the programmer.

For ooRexx there actually exists a similar programming shortcut similar to the OOo Basic approach: UNO Magic, which enables the programmer to access functions and methods without requesting the appropriate interface first. As this additional feature has officially been released after the creation of this seminar paper, chapter 5 only briefly discusses this topic. Further empirical research is definitely worthwhile as UNO Magic offers a lot of benefits which have to be explored and documented.

All in all OpenOffice.org automation bears a lot of potential, especially since version 2.0 the office suite is a serious competitor to Microsoft Office. The macro capabilities offered by OpenOffice.org are on par with the ones from the expensive office suite offered by it's big competitor. The flexibility of the UNO framework actually offers additional freedom (platform independence, support of various macro languages) at the cost of a more complex programming experience, but these drawback however could be offset in the future by UNO Magic.

5 Outlook: UNO Magic

5.1 Introduction

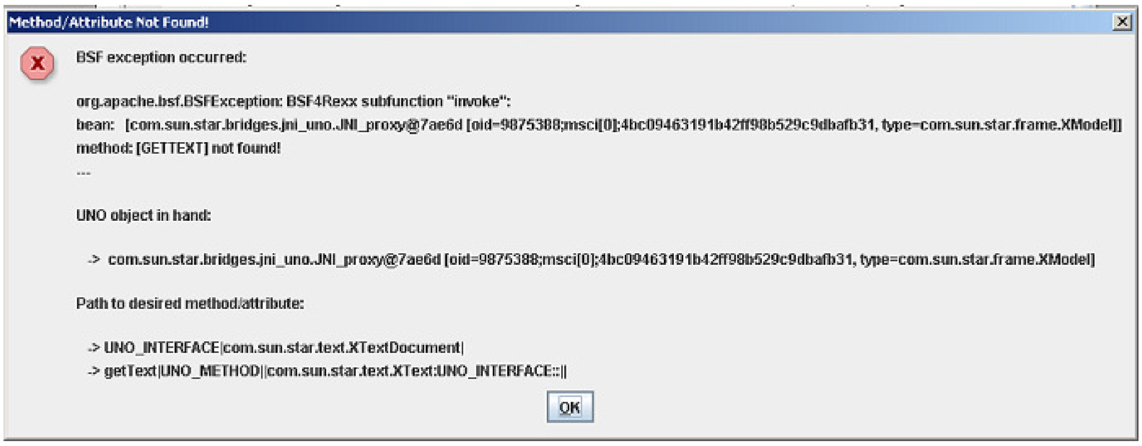
The recent version of `UNO.CLS` (dist. 20080901) includes an additional feature which makes transcribing OOo Basic macros tremendously easier: UNO Magic. This feature was actually developed some years ago by R. G. Flatscher and aids the programmer in a way that functions and methods of services can be accessed directly without the proper interface, very similar to the solution OOo Basic offers.

UNO Magic was never officially released before, because of several drawbacks:

- Transcribing ooRexx macros to languages other than OOo Basic gets significantly harder, as those languages expect the 'proper' access via interfaces (e.g. Java)
- Without the knowledge of the UNO framework structure it is actually more difficult to create own macros because of the missing "basics"
- Reproducing code programmed in such a way with the aid of the online API is very confusing as one needs to guess which interface a function or method is derived from

Transcribing OOo Basic macros however gets significantly faster, so UNO Magic offers help in a way that it displays an error message once a function or method is called without the appropriate interface. The message box contains valuable information including the proper path (via interfaces) to the desired method/function (refer to figure 22 for an example). The execution of the macro stops at this "incomplete" line and the programmer can now use the information to adapt the code accordingly. So in essence, if a programmer tries to port an OOo Basic example 1:1, or simply omits the interface when he tries to call a method or function, UNO Magic provides a message box with all information needed to complete the code in the "proper" way.

This solution offers a nice shortcut, because there is no longer the need to consult the API documentation for interfaces and thus speeds up the coding process. Using the information provided results in a nicely coded program which can serve as a nutshell example for any other programming language.



```
BSF exception occurred:

org.apache.bsf.BSFException: BSF4Rexx subfunction "invoke":
bean:   [com.sun.star.bridges.jni_uno.JNI_proxy@7ae6d [oid=9875388;msci[0];4bc09463191b42ff98b529c9dbafb31, type=com.sun.star.frame.XModel]]
method: [GETTEXT] not found!
---

UNO object in hand:

  -> com.sun.star.bridges.jni_uno.JNI_proxy@7ae6d [oid=9875388;msci[0];4bc09463191b42ff98b529c9dbafb31, type=com.sun.star.frame.XModel]

Path to desired method/attribute:

  -> UNO_INTERFACE|com.sun.star.text.XTextDocument|
  -> getText|UNO_METHOD||com.sun.star.text.XText:UNO_INTERFACE::||
```

Figure 39: UNO Magic info popup

Using the following instruction UNO Magic does not stop after the info message box, instead it tries to continue the program using the interfaces found by UNO Magic:

```
.uno~bAutoResolve=.true
```

Using this feature an OOo Basic code can be more or less transcribed 1:1 to ooRexx. Even if this seems tempting, one has to be aware of the drawbacks this shortcut entails. The execution of the macro is also interrupted by the info popups everytime an unknown function or method (access without an interface) is encountered.

5.2 UNO Magic example

The following example shows a simplified version of example 1 (DisplayAll-Styles.rex, page 24cf), where no interfaces are invoked:

```
/* DisplayAllStyles.rex, version 08-08-22                              */
/* ported from OOo Basic as found in Pitonyaks "OOo Macros Explained"  */
/* Simplified version using UNO Magic capabilities                     */
/* Listing 12, page 179cf                                              */
/* Rexx code by M. Gmeiner                                             */

.uno~bAutoResolve=.true -- continue after info popups

xScriptContext=uno.getScriptContext()     /* get the xScriptContext object */
oDoc=xScriptContext~getDocument -- get the document service (a XModel) object
```

```
vFamilies = oDoc~getStyleFamilies -- **** UNO-Magic DIRECT ACCESS ****
vFamNames = vFamilies~getElementNames   -- get the available styles
DO n OVER vFamNames                 -- iterate through the styles
    s = ""

    vStyles = vFamilies~getByName(n)
    vStyleNames = vStyles~getElementNames -- ** UNO Magic DIRECT ACCES **

    j = 1
    DO style OVER vStyleNames
        s = s j ":" style "0a"x

        IF ((j + 1) // 35 = 0) THEN -- display 35 elements each time
        DO
           .bsf.dialog~MessageBox(s, n)
            s = ""
        END
        j = j + 1
    END

    IF Length(s) > 0 THEN
        .bsf.dialog~MessageBox(s, n)
END


::requires UNO.CLS /* load UNO support (OpenOffice/StarOffice) for ooRexx   */
```

Figure 40: DisplayAllStyles.rex (UNO Magic)

Compared to the original macro UNO Magic allows some shortcuts like

```
vFamilies = oDoc~getStyleFamilies
```

instead of

```
xTextDoc = oDoc~XTextDocument
vFamilies = xTextDoc~XStyleFamiliesSupplier~getStyleFamilies
```

Note the complete lack of interfaces like `XTextDocument` and `XStyleFamiliesSupplier`!

In order to execute the macro listed above a recent version of `UNO.CLS` is mandatory!

6 References

[Aham05] Andreas Ahammer, OpenOffice.org Automation: Object Model, Scripting Languages, „Nutshell“-Examples, Bachelor Course Pager, 2005

[Augu05] Walter Augustin, Examples for Open Office Automation with Scripting Languages, Bachelor Course Pager, 2005

[ApJa01] The Jakarta Project, http://jakarta.apache.org/bsf/manual.html, retrieved on 2008-05-30

[Flat01] Flatscher, Rony G.: Automatisierung von Java Anwendungen (2) (Course slides), http://wwwi.wu-wien.ac.at/Studium/LVAUnterlagen/rgf/autojava/folien/

[Flat02] Flatscher, Rony G.: Automatisierung von Java Anwendungen (7) (Course slides), http://wwwi.wu-wien.ac.at/Studium/LVAUnterlagen/rgf/autojava/folien/

[Flat02] Flatscher, Rony G.: Automatisierung von Java Anwendungen (10) (Course slides), http://wwwi.wu-wien.ac.at/Studium/LVAUnterlagen/rgf/autojava/folien/

[King01] King, K. N: The Case for Java as a First Language, http://www2.gsu.edu/~matknk/java/reg97.htm, retrieved on 2008-05-22

[Lohm07] Lohmaier, Christian: OO-Snippets: Embed a Graphic into a Textdocument, http://codesnippets.services.openoffice.org/Writer/Writer.EmbedAGraphicIntoATextdocument.snip, retrieved on 2008-06-01

[OOo01] OpenOffice.org, http://www.openoffice.org/about_us/ooo_release.html, retrieved on 2008-05-21

[Pito04] Pitonyak, Andrew: OpenOffice.org Macros Explained (2004)

[Pito08] Pitonyak, Andrew: Useful Macro Information For OpenOffice, revision 1035, http://www.pitonyak.org/AndrewMacro.odt, retrieved on 2008-06-01

[Port01] Porting Project: The OpenOffice.org Porting project home, http://porting.openoffice.org/, retrieved on 2008-06-03