Programming With Objects: A REXX-Based Approach

Eric Giguère
Rob Veitch

Computer Systems Group
University of Waterloo
Waterloo, Ontario, Canada
N2L 3G1

giguere@csg.uwaterloo.ca
rgv@csg.uwaterloo.ca

Introduction

The emergence of graphically-oriented user interfaces (GUIs) on a variety of multitasking platforms gives rise to a whole new set of problems for REXX language implementors. What do you do when a console-oriented language like REXX is to be ported to an environment like Microsoft Windows that lacks any kind of command-line environment? How does a user access the GUI from REXX to create dialogs? What changes are required to a REXX interpreter for it to function in a multitasking environment?

These are some of the issues we tackled in implementing a REXX interpreter, WRexx, for use in the Microsoft Windows environment. This paper discusses our approaches to solving these problems, concentrating for the most part on the REXX-to-GUI interface, where we feel the interesting and original work of this implementation lies. (Readers with no Windows programming experience may wish to read the appendix for a quick overview of Windows.)

Note: Throughout this paper, *Windows* refers to the Microsoft Windows environment, *X11* refers to the base X Window System, *Xt* refers to the X Toolkit and *DOS* refers to MS-DOS/PC-DOS.

1. Adapting The REXX Console Model

The REXX language assumes the existence of a *console* through which it can interact with a user. The `SAY` instruction is the most obvious example:

```
say "Please enter your name:"
pull name
```

This model works well on systems like DOS, CMS, Unix (text mode) and OS/2 (text mode), where a console is the normal mode of operation. It also works well on hybrid systems like X11 and the Amiga, where virtual consoles coexist within the GUI environment. Systems like the Macintosh and Windows, however, do not provide operating system support for consoles. Consoles become the responsibility of the REXX environment.

WRexx uses a virtual console to handle user interaction and tracing, and a separate virtual console for displaying error messages. The consoles are windows that can be moved and resized like any conventional window. Users can also scroll through the console's contents using the cursor keys or the scrollbars. Neither console is displayed until input or output occurs, and once visible remains onscreen until explicitly closed.

WRexx also adds a virtual console stream type to the REXX I/O model:

```
call lineout 'con:My Window', 'Hello, world'
```

The consoles can be used with any of the stream-based functions.

2. UI Options for REXX

While virtual console support allows a REXX interpreter to *function* in a GUI environment, the interpreter will be more useful if it can also *use* the environment. Instead of consoles, REXX programs can use windows, buttons, edit fields and other *user interface objects* to interact with the user.

When designing WRexx we considered three options for adding GUI access to REXX:

1. **Language extensions.** Extending the REXX language to include new instructions and programming structures for building dialogs, menus and so on.
2. **UI-oriented functions.** Adding functions like `CreateMenu()`, `CreatePushButton()`, `ShowWindow()`, etc., as BIFs or through an external function library.
3. **Object-oriented functions.** Adding functions like `UICreate()`, `UISet()`, `UIGet()`, etc. These functions work on generic user interface objects.

There are advantages and disadvantages to each approach. Language extensions make it easy to connect individual objects and events with REXX code:

```
menu "File"
    item "Open..."
        call OpenFile
    item "Exit"
        exit
endmenu
```

But such extensions are also completely non-portable and may require other changes to the REXX language. We rejected this approach because we wanted to remain faithful to the language as defined by Cowlishaw's book [Cowlishaw 90].

Once the function-based approach was chosen, it became a matter of choosing between the two kinds of function libraries: very specific, UI-oriented functions, or more generic, object-oriented functions. We eventually settled on the object-oriented approach (described in the next section) because we felt it would be a more consistent and extensible interface, even though UI-oriented functions are the more traditional approach for REXX extensions.

3. The OOUI Library

The WRexx GUI library is known simply as the "OOUI" (object-oriented user interface, pronouced *oo-ee*) library. It is implemented as a Windows dynamic link library (DLL) and is only needed by REXX programs that wish to access the Windows GUI.

3.1 Objects and Classes

The OOUI library implements a hierarchical class structure of *window objects* such as edit fields, buttons and various containers. Each object has a set of *properties* that determines its current state and behaviour, as well as a set of *methods* to alter that state. The properties, methods and behaviour of an object are defined by its *class*. The library is hierarchical in the sense that each class *inherits* properties, methods and behaviour from a parent class or *superclass*. The *subclass* usually adds new properties or methods to those of the superclass. The current OOUI class hierarchy is shown in Figure 1. It is based for the most part on the window types defined by Microsoft Windows.

C programmers can also use the facilities provided by the OOUI DLL to write their own DLLs to implement new classes and subclasses.

3.2 Object Manipulation

Objects are manipulated from within WRexx using five functions. `UICreate()` creates an object of a given class and `UIDestroy()` destroys an object. `UISet()` and `UIGet()` are used to set and retrieve property values, while `UIMethod()` invokes a method. Objects are identified by *handle* (returned by `UICreate()`) or by *name* (assigned by the user).

Objects are also created hierarchically. Except for objects called *Forms*, each object has a parent object on the screen which affects the child's positioning and other properties. Each object tree is rooted on a Form, which is a top-level (application or dialog) window.

For example, the following code creates a blank Form on the screen and immediately centers it:

```
f = UICreate( '', 'Form', 'visible', 'true', ,
              'height', 100, 'width', 200 )
call UIMethod f, 'centerwindow'
```

This example attaches some text and a button to the Form:

```
f = UICreate( '', 'Form', 'visible', 'false' )
```

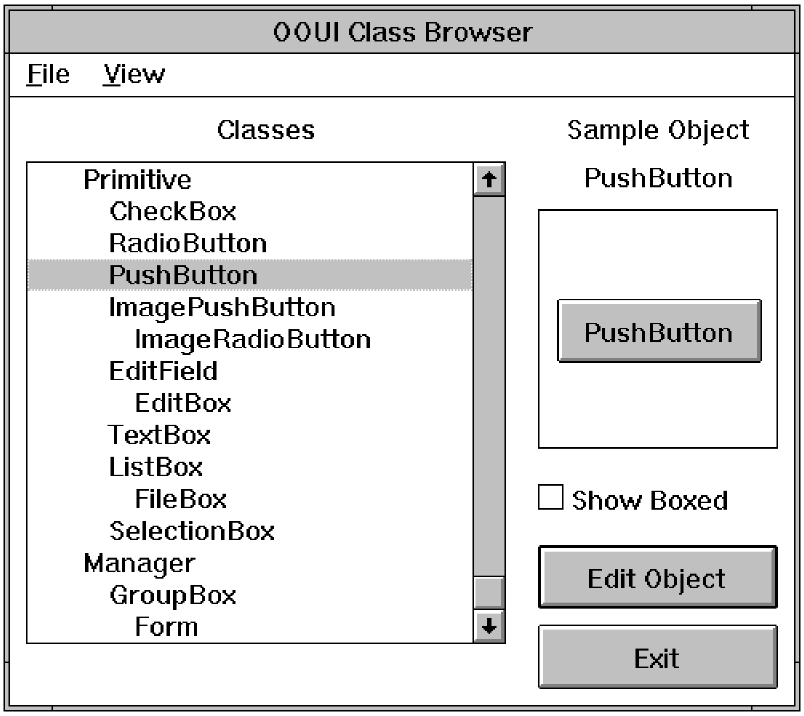

Figure 1: Viewing the OOUI Class Hierarchy

```
t = UICreate( f, 'TextBox', 'caption', 'This is some text' )
p = UICreate( f, 'PushButton', 'caption', 'Press Me!' )
call UISet f, 'visible', 'true'
```

Because the Form is the parent object for both the TextBox and the PushButton, neither child object will be shown until the Form itself is made visible.

Note: Form and GroupBox objects include behaviour (which may be turned off) for automatically resizing and positioning their children, thus freeing the programmer from having to specify absolute coordinates when positioning objects.

When finished with an object, a call to `UIDestroy()` recursively destroys an object and all of its children.

3.3 Events and REXX

Objects will generate events whenever something interesting occurs; for example, when a pushbutton is clicked. These events must be passed to the REXX program that created the objects so that the program can respond to the user. This is done using *event strings* for each object's events. The event string is merely a string that is associated with a specific event. The string will be returned to the REXX program whenever that event occurs. The REXX program checks for pending events by calling the `UIEvent()` function, which will return the next event string. For example, the PushButton object has a "click"

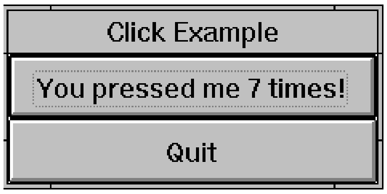

Figure 2: Running 'click.rex'

event signifying that the user has clicked on the button. The following program demonstrates the use of event strings:

```
/* click.rex */

f = UICreate( '', 'Form', 'caption', 'Click Example' )
p = UICreate( f, 'PushButton', 'caption', "You haven't pressed me!", ,
              'click', 'call FirstPress' )
e = UICreate( f, 'PushButton', 'caption', 'Quit', ,
              'click', 'exit 0' )

do forever
    interpret UIEvent()
end

FirstPress:
    call UISet p, 'caption', 'You pressed me once!'
    count = 1
    call UISet p, 'click', 'call NextPress'
    return

NextPress:
    count = count + 1
    call UISet p, 'caption', 'You pressed me' count 'times!'
    return
```

The program creates a form with two pushbuttons and then enters an *event loop*, waiting for user events to occur. When the user presses a button, an event string is returned to the program and the program executes it using the `interpret` statement.

> Notice that **no language modifications or extensions** were necessary to add GUI support to REXX, only clever use of the `interpret` instruction.

In some cases it may not be obvious to which object an event belongs. The `UIInfo()` function can be used to obtain this and other information on the string most recently returned by `UIEvent()`.

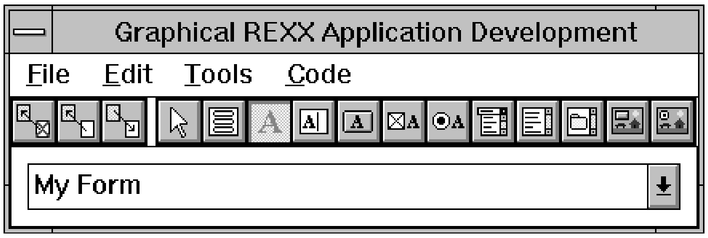

Figure 3: The GRAD Tool

4. Programming With OOUI

After using the OOUI library and REXX, three things become apparent:

1. **The traditional REXX program structure is no longer suitable.** REXX programs typically consist of a single file, augmented with external (and independent) functions. However, even the simplest REXX application under Windows may display several Forms with numerous objects on each form. The single-file approach in this case leads to monolithic programs that take longer to load and are harder to debug. Performance is improved and debugging made simpler (and code reuse encouraged) if an OOUI-based program is split across multiple small REXX files.

2. **Exposing variables across files is extremely useful.** Splitting a program into several files is much more tolerable if variables can be exposed across files. WRexx has been extended so that `procedure expose` will expose variables across file boundaries. (This feature becomes invaluable to the programmer in a very short time.)

3. **OOUI programming is ugly, so automated tools are needed.** Adding object-oriented concepts to a procedural language almost always seems to lead to ugly code, and REXX is no exception to the rule. Writing the REXX programs to display complicated dialogs is itself a complicated process if all the programmer has is a text editor to work with. Tools such as the class browser (Figure 1) and GRAD[1] (Graphical REXX Application Development, Figure 3) can be of immense help.

An issue that also comes up when using the OOUI library is that of multiple independent (i.e., modeless) Forms. There is only one call to `UIEvent()` active at any time (because there is only a single thread of execution within a REXX program), and it may be in a different file or procedure. Problems can then arise due to scoping issues. Luckily, there are few situations where modeless Forms are required. (Problems do not arise with modal Forms because the previously active Form is always disabled before the new one is made active.)

[1]The reader may find it interesting to note that both the browser and the GRAD tool are themselves written in REXX.

5. Conclusions

Virtual consoles and the OOUI function library allow WRexx to thrive in the Windows environment. With them, REXX can be used both as a general-purpose scripting language (which Windows lacks) or for implementing real applications.

Appendix A. A Crash Course on Windows Programming

Readers with no GUI programming experience will discover that there is a substantial learning curve involved in developing for systems such as Microsoft Windows. This section is intended to provide you with enough information to understand the rest of the paper, but for more complete treatments of GUI programming models please refer to the bibliography. (**Note:** The Windows programming model is almost identical to the model used by the OS/2 Presentation Manager. Readers with PM experience should have little trouble understanding the terminology used throughout this paper.)

What is Microsoft Windows?

Windows is a multitasking environment built on top of DOS. It provides a windowing environment, device-independent graphics and inter-application communication (IAC) facilities. Windows applications will not run under DOS, as they use a completely different application programming interface (API) and a different programming model. Windows can emulate a DOS environment (the so-called "DOS box") in which to run DOS programs, but such programs cannot take advantage of Windows' features.

The multitasking model used by Windows is often termed *cooperative multitasking*: each Windows application will run until it voluntarily releases control of the CPU, at which time Windows will switch control to another application. Well-behaved applications must ensure that they give up the CPU at small time intervals. Unlike OS/2, Windows is not a preemptive system, nor does it support threads (lightweight processes). Because of this there are no semaphores or other means of task synchronization.

Programs and User Interaction

Like other GUI platforms, Windows uses an *event-driven* programming model. Applications create one or more windows, to which are attached user interface objects such as buttons and menus. The programs then wait for user events (such as clicking on a button or pressing a key) to occur. When an event occurs, Windows sends a *message* to the application that "owns" the event. The message is added to the end of a queue which the application continually checks for new messages. Each Windows application has a loop in it to do this (in pseudo-code):

```
do forever
    get next event
    process event
end
```

The same type of loop is used in Macintosh, Amiga and X11 applications. In Windows (and PM) the loop serves mainly to demultiplex the application message queue, dispatching messages to the appropriate *window procedure*. When you create a window (or more accurately, a *window class*) you register a window procedure to handle that window's events, including those that bypass the application message queue.

```
do forever
    get next event
    dispatch event
end

window procedure:
    case message is BUTTONDOWN
        ....
    etc.
end procedure
```

Note that Xt applications (this includes Motif applications) take this demultiplexing one step further by registering *callback routines* for each *event* of interest.

Dynamic Link Libraries

The Dynamic Link Library (DLL) is a method for sharing code and resources between Windows applications. (Windows itself is implemented as a set of DLLs.) A DLL is a run-time library that is loaded into memory on demand and dynamically linked to an application. Applications can call DLL routines just like normal (statically-linked) library routines.

One important feature of a DLL is that it has its own dataspace, shared by all tasks using the DLL. (Note: OS/2 has DLLs as well, but OS/2 DLLs have a separate dataspace for each process.)

Dynamic Data Exchange

Dynamic Data Exchange (DDE) is a form of inter-application communication. Applications communicate by setting up DDE "conversations" using invisible windows and a well-defined protocol. Communication is done by sending messages to these windows. The DDE protocol includes facilities for sending commands and for maintaining data links.

References

[Cowlishaw 90] M. F. Cowlishaw. *The REXX Language: A Practical Approach to Programming, 2nd edition*, Prentice-Hall, 1990.