ooRexx Documentation 4.2.4

ooDialog User Guide

Friday, December 23, 2022 svn revision 12583

ooRexx Documentation 4.2.4 ooDialog User Guide
Friday, December 23, 2022 svn revision 12583
Edition 2022.12.22

Author	Open Object Rexx™
Author	W. David Ashley
Author	Rony G. Flatscher
Author	Rick McGuire
Author	Mark Miesfeld
Author	Lee Peedin
Author	Oliver Sims
Author	Jon Wolfers

Preface

This book provides a general user's guide to the ooDialog framework, and is a companion book to the *ooDialog Reference*. The ooDialog framework is part of the Open Object Rexx distribution on the Windows® platform. The *User Guide* discusses the general ideas needed to use the framework to its best advantage.

1. Document Conventions

This manual uses several conventions to highlight certain words and phrases and draw attention to specific pieces of information.

1.1. Typographic Conventions

Typographic conventions are used to call attention to specific words and phrases. These conventions, and the circumstances they apply to, are as follows.

`Mono-spaced Bold` is used to highlight literal strings, class names, or inline code examples. For example:

> The **`Class`** class comparison methods return **`.true`** or **`.false`**, the result of performing the comparison operation.

> This method is exactly equivalent to **`subWord(`***n*`, 1)`**.

`Mono-spaced Normal` denotes method names or source code in program listings set off as separate examples.

> This method has no effect on the action of any `hasEntry`, `hasIndex`, `items`, `remove`, or `supplier` message sent to the collection.

```
-- reverse an array
a = .Array~of("one", "two", "three", "four", "five")

-- five, four, three, two, one
aReverse = .CircularQueue~new(a~size)~appendAll(a)~makeArray("lifo")
```

Proportional Italic is used for method and function variables and arguments.

> A supplier loop specifies one or two control variables, *index*, and *item*, which receive a different value on each repetition of the loop.

> Returns a string of length *length* with *string* centered in it and with *pad* characters added as necessary to make up length.

1.2. Notes and Warnings

Finally, we use three visual styles to draw attention to information that might otherwise be overlooked.

> **Note**
>
> Notes are tips, shortcuts or alternative approaches to the task at hand. Ignoring a note should have no negative consequences, but you might miss out on a trick that makes your life easier.
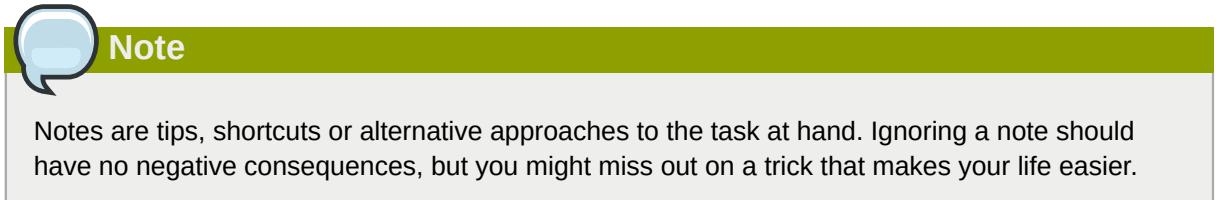
Important

Important boxes detail things that are easily missed, like mandatory initialization. Ignoring a box labeled 'Important' will not cause data loss but may cause irritation and frustration.

Warning

Warnings should not be ignored. Ignoring warnings will most likely cause data loss.

2. How to Read the Syntax Diagrams

Throughout this book, syntax is described using the structure defined below.

• Read the syntax diagrams from left to right, from top to bottom, following the path of the line.

The **>>---** symbol indicates the beginning of a statement.

The **--->** symbol indicates that the statement syntax is continued on the next line.

The **>---** symbol indicates that a statement is continued from the previous line.

The **--->< ** symbol indicates the end of a statement.

Diagrams of syntactical units other than complete statements start with the **>---** symbol and end with the **--->** symbol.

• Required items appear on the horizontal line (the main path).

```
>>-STATEMENT--required_item------------------------------------><
```

• Optional items appear below the main path.

```
>>-STATEMENT--+---------------+--------------------------------><
              +-optional_item-+
```

• If you can choose from two or more items, they appear vertically, in a stack. If you must choose one of the items, one item of the stack appears on the main path.

```
>>-STATEMENT--+-required_choice1-+-----------------------------><
              +-required_choice2-+
```

• If choosing one of the items is optional, the entire stack appears below the main path.

```
>>-STATEMENT--+------------------+-----------------------------><
              +-optional_choice1-+
              +-optional_choice2-+
```

- If one of the items is the default, it appears above the main path and the remaining choices are shown below.

```
                +-default_choice--+
   >>-STATEMENT--+----------------+----------------------------><
                +-optional_choice-+
                +-optional_choice-+
```

- An arrow returning to the left above the main line indicates an item that can be repeated.

```
                +----------------+
                V                |
   >>-STATEMENT----repeatable_item-+---------------------------><
```

A repeat arrow above a stack indicates that you can repeat the items in the stack.

- A set of vertical bars around an item indicates that the item is a fragment, a part of the syntax diagram that appears in greater detail below the main diagram.

```
   >>-STATEMENT--| fragment |-------------------------------><
```

fragment:

```
   |--expansion_provides_greater_detail--------------------------|
```

- Keywords appear in uppercase (for example, **PARM1**). They must be spelled exactly as shown but you can type them in upper, lower, or mixed case. Variables appear in all lowercase letters (for example, **parmx**). They represent user-supplied names or values.

- If punctuation marks, parentheses, arithmetic operators, or such symbols are shown, you must enter them as part of the syntax.

The following example shows how the syntax is described:

```
        +-,------+
        V        |
  >>-MAX(----number-+--)----------------------------------------><
```

3. Getting Help and Submitting Feedback

The Open Object Rexx Project has a number of methods to obtain help and submit feedback for ooRexx and the extension packages that are part of ooRexx. These methods, in no particular order of preference, are listed below.

3.1. The Open Object Rexx SourceForge Site

The *Open Object Rexx Project*[1] utilizes *SourceForge*[2] to house the *ooRexx Project*[3] source repositories, mailing lists and other project features. Over time it has become apparent that the Developer and User mailing lists are better tools for carrying on discussions concerning ooRexx and

---

[1] http://www.oorexx.org/

[2] http://sourceforge.net/

[3] http://sourceforge.net/projects/oorexx

that the Forums provided by SourceForge are cumbersome to use. The ooRexx user is most likely to get timely replies from one of the mailing lists.

Here is a list of some of the most useful facilities provided by SourceForge.

The Developer Mailing List

You can subscribe to the oorexx-devel mailing list at *ooRexx Mailing List Subscriptions*[4] page. This list is for discussing ooRexx project development activities and future interpreter enhancements. It also supports a historical archive of past messages.

The Users Mailing List

You can subscribe to the oorexx-users mailing list at *ooRexx Mailing List Subscriptions*[5] page. This list is for discussing using ooRexx. It also supports a historical archive of past messages.

The Announcements Mailing List

You can subscribe to the oorexx-announce mailing list at *ooRexx Mailing List Subscriptions*[6] page. This list is only used to announce significant ooRexx project events.

The Bug Mailing List

You can subscribe to the oorexx-bugs mailing list at *ooRexx Mailing List Subscriptions*[7] page. This list is only used for monitoring changes to the ooRexx bug tracking system.

Bug Reports

You can create a bug report at *ooRexx Bug Report*[8] page. Please try to provide as much information in the bug report as possible so that the developers can determine the problem as quickly as possible. Sample programs that can reproduce your problem will make it easier to debug reported problems.

Documentation Feedback

You can submit feedback for, or report errors in, the documentation at *ooRexx Documentation Report*[9] page. Please try to provide as much information in a documentation report as possible. In addition to listing the document and section the report concerns, direct quotes of the text will help the developers locate the text in the source code for the document. (Section numbers are generated when the document is produced and are not available in the source code itself.) Suggestions as to how to reword or fix the existing text should also be included.

Request For Enhancement

You can suggest ooRexx features at the *ooRexx Feature Requests*[10] page.

Patch Reports

If you create an enhancement patch for ooRexx please post the patch using the *ooRexx Patch Report*[11] page. Please provide as much information in the patch report as possible so that the developers can evaluate the enhancement as quickly as possible.

Please do not post bug fix patches here, instead you should open a bug report and attach the patch to it.

---

[4] http://sourceforge.net/mail/?group_id=119701

[5] http://sourceforge.net/mail/?group_id=119701

[6] http://sourceforge.net/mail/?group_id=119701

[7] http://sourceforge.net/mail/?group_id=119701

[8] http://sourceforge.net/tracker/?group_id=119701&atid=684730

[9] http://sourceforge.net/tracker/?group_id=119701&atid=1001880

[10] http://sourceforge.net/tracker/?group_id=119701&atid=684733

[11] http://sourceforge.net/tracker/?group_id=119701&atid=684732

The ooRexx Forums

The ooRexx project maintains a set of forums that anyone may contribute to or monitor. They are located on the *ooRexx Forums*[12] page. There are currently three forums available: Help, Developers and Open Discussion. In addition, you can monitor the forums via email.

3.2. The Rexx Language Association Mailing List

The *Rexx Language Association*[13] maintains a mailing list for its members. This mailing list is only available to RexxLA members thus you will need to join RexxLA in order to get on the list. The dues for RexxLA membership are small and are charged on a yearly basis. For details on joining RexxLA please refer to the *RexxLA Home Page*[14] or the *RexxLA Membership Application*[15] page.

3.3. comp.lang.rexx Newsgroup

The *comp.lang.rexx*[16] newsgroup is a good place to obtain help from many individuals within the Rexx community. You can obtain help on Open Object Rexx or on any number of other Rexx interpreters and tools.

4. Related Information

Open Object Rexx: Windows ooDialog Reference

Open Object Rexx: Programming Guide

Open Object Rexx: Reference

---

[12] http://sourceforge.net/forum/?group_id=119701

[13] http://www.rexxla.org/

[14] http://rexxla.org/

[15] http://www.rexxla.org/rexxla/join.html

[16] http://groups.google.com/group/comp.lang.rexx/topics?hl=en

About This Book

This book provides a general user's guide to the ooDialog framework, and is a companion book to the *ooDialog Reference*. The ooDialog framework is part of the Open Object Rexx distribution on the Windows® platform. The *User Guide* discusses the general ideas needed to use the framework to its best advantage.

1.1. Who Should Use This Book

This book is intended for Open Object Rexx programmers who want to design graphical user interfaces for their applications using ooDialog. It is intended to paint a broader picture of how to use ooDialog than the a purely reference manual can. Readers will gain a better understanding of the general concepts used in the ooDialog framework.

In addition to ooDialog concepts, some discussion of how the underlying Windows dialogs and controls behave and are normally programmed is included. The will give the reader some idea of what can and can not be done using ooDialog. Knowing some of the inherent capabilities and restrictions of the operating system allows ooRexx programmers to better design their programs.

1.2. How This Book is Structured

This book takes the reader through a series of exercises, each exercise building on the previous one. The final exercise will complete a simple Sales Order Management application. The intent is not to provide a realistic application, but rather to illustrate the use of the main ooDialog features while building a working if simplistic application. While not production-strength, this sample application will include a number of different dialogs, together with, in later chapters, a sample model-view framework of the kind often used to provide the application programmer with greater development simplicity.

In the course of these exercises, various key ooDialog concepts are discussed. However, this book is not, and is not intended to be, a technical reference for ooDialog. Rather it is an introduction to ooDialog. Its intent is merely to familiarize the reader with the basics. The reader is assumed to be reasonably familiar with the object-orientation concepts, and also with programming using ooRexx.

Completed code for each exercise is available from the **samples** sub-directory of the ooRexx installation directory. Specifically, the code for the exercises will be in **samples\oodialog \userGuide\exercises**. Note that exercise numbers are aligned with chapter numbers. Thus the first exercise, introduced in Chapter 2, is found in the folder **Exercise02**.

Hello ooDialog World

The purpose of ooDialog is to enable ooRexx developers to provide users with a graphical user interface or "GUI". A GUI is a collection of windows and dialogs. Each contains a number of controls, such as edit controls, push buttons, list boxes, and so forth. The user keys data into controls (e.g. types into an edit control) or manipulates controls with a mouse (e.g. selects an item in a listbox). Some of these actions invoke application code which in turn makes some change to the window or dialog, or causes some other action such as data access, or both.

Before continuing, it's worth distinguishing between a "window" and a "dialog". A dialog is a stylized form of window that is familiar to most users. As dialogs have evolved they have become more useful, and can now provide the user interface function for many applications. Also, a dialog is drawn by the operating system, while drawing a normal window is mostly the programmer's responsibility. Thus producing an application needs much less programming work because the ooRexx programmer doesn't need to know or understand the low-level mechanics of drawing to the screen. In summary, dialogs now have many window functions, and are much easier to produce. And it's this that makes ooDialog a particularly useful extension to ooRexx.

There are three general areas of concern in designing an ooDialog application:
• Designing the appearance of a dialog

• Designing the desired user interactions with the dialog

• Designing the code that implements both appearance and interactions

And there are three corresponding areas of implementation concern:
• Laying out the dialog

• Implementing the actions requested by users of a dialog

• Showing the results of those actions to the user.

This document does not pretend to be a guide to best practice in the areas of design, although it tries to conform with good design principles. However, this document *does* aim to familiarize readers with the essentials of ooDialog application implementation.

So, before starting the first exercise, please make sure that you have downloaded and installed the latest versions of ooRexx and ooDialog. Please also run one or more of the samples in `Start --> All Programs --> Open Object Rexx --> ooRexx Samples --> ooDialog` to ensure that your installation works properly.

And please do use the ooDialog Reference for details on any ooDialog class, method or function mentioned in this Guide.

2.1. Getting Started

The first exercise creates and displays a blank dialog with the title "Hello World!". Try running it - it's the file **HelloWorld.rex** in the folder **C:\Program Files\ooRexx\samples\oodialog\userGuide \exercises\Exercise02** (exercise numbers map to chapter numbers). *Figure 2.1, "The 'Hello World' Dialog"* shows what you should see.

The Command Prompt window that appears with the **Hello World** dialog can be useful for debugging, but it can be dispensed with, and later we'll find out how. For now, let's look at the **Hello World** code (excluding comments):
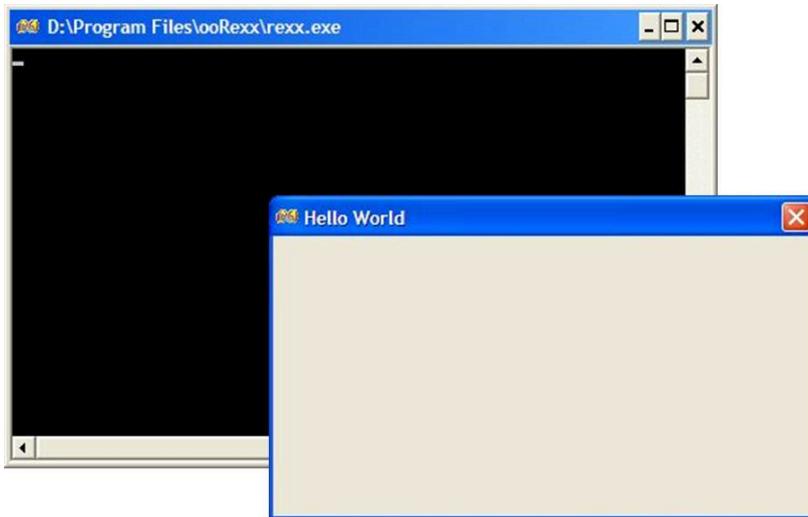
Figure 2.1. The 'Hello World' Dialog

First, there's code that kicks things off:

```
dlg = .HelloWorld~new
dlg~execute("SHOWTOP", IDI_DLG_OOREXX)
```

The first statement creates an instance of the class **HelloWorld**, and assigns the instance to the variable **dlg** (the **HelloWorld** class is defined in the third part of the code). The second statement invokes the **execute** method of **HelloWorld**, and it is this that displays the dialog. The first parameter **SHOWTOP** is one of several ways of defining how the dialog is surfaced (see the ooDialog Reference for details). The second parameter states that the icon at the extreme top left of the dialog should be the normal ooRexx icon. This graphic is termed a "resource" in ooDialog, and there are a number of such predefined constants (again, see the ooDialog Reference for details).

Note that the usual naming conventions are observed: upper camel case for classes (e.g. **HelloWorld**) and lower camel case for variables (including of course instance variables - e.g. **dlg**).

Second, there's a directive to ooRexx to use ooDialog:

```
::REQUIRES "ooDialog.cls"
```

This directive allows all the classes defined by ooDialog to be accessible to our program. If this statement is absent, then the following error appears on the initiating command prompt:

```
16 *-* ::class 'HelloWorld' subclass UserDialog
Error 98 running ... Exercise02\HelloWorld.rex line 58: Execution error
Error 98.909:  Class "USERDIALOG" not found
```

Note that it's **UserDialog** (the superclass for **HelloWorld**) that's not found. This is because the **::requires ooDialog.cls** statement not only says we're using ooDialog, but also provides access to all the classes provided *by* ooDialog.

Finally, there's the definition of the class **HelloWorld**:

```
::CLASS HelloWorld SUBCLASS UserDialog
  ::METHOD init
    forward class (super) continue
    self~create(30, 30, 257, 123, "Hello World", "CENTER")
```

The first line defines a class called **HelloWorld** as a subclass of the ooDialog-provided class **UserDialog** (and yet again, but finally, see the ooDialog Reference for full details). Among other things, **UserDialog** enables the programmer to define the dialog layout in code. This can get cumbersome in more complex dialogs, and later we'll meet a simpler way of defining the dialog layout.

The second line defines the **init** method of **HelloWorld**, and the third line forwards the **init** message to the superclass which then does the heavy work of creating the dialog. But why use **forward** instead of **self~init:super**? The reason is that **forward** applies not only to the method but also to all its arguments, whatever these may be. Which is exactly what's required here.

Finally, the last line sends a **create** message to **self** and hence to **UserDialog**. This method defines the "template" to be used for the dialog. The parameters are as follows:
- The first two parameters define, in "dialog units", the x and y position on the screen of the top left corner of the dialog. Dialog units (rather than pels) are used to provide device independence.

- The third and fourth parameters define the width and height of the dialog, again in dialog units.

- The fifth parameter is the dialog's title, and the last parameter - **CENTER** - is the dialog "style" (of which there are several). **CENTER** states that regardless of the first two parameters, the dialog will be positioned in the center of the screen. Styles are an important part of dialog definition. Try removing the **CENTER** parameter (and its preceding comma of course) and see what happens. Then replace the **CENTER** parameter, and (just to make sure that you've replaced it correctly) run the program again, and this time try to re-size the dialog. You can't. Then replace **"CENTER"** by **"CENTER THICKFRAME"** and re-run the program. The dialog now has a sizable border. Thus styles not only affect appearance, they can also define behavior.

With this instance of the **HelloWorld** class having been set up properly, the dialog is actually surfaced (made visible) by the **execute** message (handled by HelloWorld's superclass) sent by the second statement in the program which was:

```
dlg~execute("SHOWTOP", IDI_DLG_OOREXX)
```

Now let's add some behavior to the dialog. We're going to build a "Words of Wisdom" dialog that will display words of wisdom when a button is pressed.

2.2. Visible Behavior

This section is in two parts. First we create a dialog that invites the user to press a button for more "words of wisdom" - but the button doesn't work. Second, we make the button work. In this way, we both add to the way dialogs are populated with controls, and also show how user input is handled.

2.2.1. Adding Controls to the Dialog

First, try running **Wow.rex** from the **Exercises\Exercise02** folder ("Wow" being short for "Words of Wisdom"). You should see a dialog entitled "Words of Wisdom" which is blank with the exception of some (alleged) words of wisdom and two push-buttons. One of these does nothing, the other (Cancel) closes the dialog. These buttons are examples of "controls" - sometimes called "widgets" - that populate a dialog and enable both display of information and user interaction with the dialog.

Now let's look at the code. The first seven lines (excluding comments and blank lines) are essentially the same as the first seven in **HelloWorld.rex**:

```
dlg = .WordsOfWisdom~new
dlg~execute("SHOWTOP", IDI_DLG_OOREXX)
::REQUIRES "ooDialog.cls"
::CLASS 'WordsOfWisdom' SUBCLASS UserDialog
```

```
    ::METHOD init
      forward class (super) continue
      self~create(30, 30, 257, 123, "Words of Wisdom", "CENTER")
```

However, a *defineDialog* method has been added:

```
  ::METHOD defineDialog
    self~createPushButton(901, 142, 99, 50, 14, "DEFAULT", "More wisdom")
    self~createPushButton(IDCANCEL, 197, 99, 50, 14, ,"Cancel")
    self~createStaticText(-1, 40, 40, 200, 20, , -
     "Complex problems have simple solutions"||.endofline||"- which are wrong.")
```

This method is invoked automatically by the superclass, and consists of three statements each of which creates a control. The first two each create a pushbutton:

- A "More Wisdom" pushbutton, which has been given the resource ID of "901" (the first parameter). Controls are identified by numbers (IDs) or, as we'll see later, by symbolic names. You can pick any number although numbers -1 and 1 through 50 are pre-defined by ooDialog. For example, resource ID "1" is an "OK" button. Resource numbers or IDs identify controls to the underlying Windows platform, and can be given in either numeric or symbolic form, as will be discussed in Chapter 3. Each control should have a different number (although there are some situations, which will be met later, where it's useful for two or more controls to have the same number). The next four parameters define the position of the button in the dialog, and the sixth ("DEFAULT") specifies that this button is to be the default action for pressing the enter key. The seventh parameter is the text shown on the button.

- A "Cancel" pushbutton, whose ID **IDCANCEL** makes this button perform the standard dialog cancel action - that is, close the dialog without saving any changes. (An OK button should save any changes made by the user, and then close the dialog - preferably with an intervening "Save changes?" message box with options "Yes", "No, and "Cancel".)

The third statement creates some static text (text that cannot be changed by the user), with the text itself as the last parameter. The first parameter is the resource number "-1" which is the pre-defined resource ID for a Static Text control (although you can use other numbers above 50 - for example if you want to distinguish between different static text controls or when you want to programmatically change the text - neither of which is a requirement here). The next four parameters define the size and position of the static text control. Last but not least is the text to be displayed. This text comprises the initial (and so far only) "words of wisdom".

Now we need to make the **More wisdom** button work, which we now do in **Wow2.rex**.

2.2.2. Making The Controls Work

First, run **Wow2.rex**. When you click the **More wisdom** pushbutton, you see different text appearing in the center of the screen, replacing the previous text. By the way, you'll also see debug information appear on the command prompt each time you click the **More wisdom** button - there's a "say" statement in the code that's not mentioned here, but you can easily find it. The real question at the moment is: how do we create the pushbutton's visible behavior?

When a control is actioned by a user (e.g. pressing a pushbutton), we need some way in the program of kicking off a method that provides a visible response. Remember that a key principle of UI design is that of least astonishment: if the user is astonished by what the computer is doing in response to a wholly innocent user interaction with some UI widget, then that principle is breached. Of course, astonishment can be pleasant or unpleasant; and if you can create ooDialog GUIs which *pleasantly* astonish their users, then you probably don't need to read this Guide.

In ooDialog, there are a number of ways of connecting a user action to a method. However, a user action is actually signaled by an event emitted from the underlying Windows GUI software

infrastructure. ooDialog connects that event to a method in the ooRexx dialog object. So the source of the event (the user action) is not ooDialog. This means that if you want to capture a user action that the Windows infrastructure doesn't capture (for example hovering the mouse over an edit control), then there's no way ooDialog can do it either.

Having said that, one of the simplest ways of having a user action connected to an ooRexx method is by supplying the name of the method as a parameter of the actionable control. And, since a pushbutton control provides for just such an approach, that's what we'll do here. The first lines of code in **Wow2.rex** - down to the **defineDialog** method - are almost identical to those of **wow.rex**, with two significant changes. The method is as follows:

```
::method defineDialog
  self~createPushButton(901, 142, 99, 50, 14, "DEFAULT", "More wisdom", OkClicked)
  self~createPushButton(IDCANCEL, 197, 99, 50, 14, ,"Cancel")
  self~createStaticText(101, 40, 40, 200, 40, , "Click 'More wisdom'")
```

The two significant changes are as follows. Firstly, the statement **self~createPushButton** has an additional parameter **okClicked**. This is the name of the method that is automatically invoked by ooDialog when the user clicks on the **More wisdom** pushbutton. Secondly, the statement **self~createStaticText** has the ID 101 rather than -1 since we want to have the text changed when the "More Wisdom" button is pressed. Also, the space for text to be displayed has increased from 20 to 40, and the initial text has been changed to "Click 'More wisdom'".

However, the major change to the program is the new **okClicked** method that picks a "words of wisdom" text and displays it. Pseudocode for this method is as follows:

```
Method okClicked
  Create array 'arrWow' and add a 'words of wisdom' text strings to each of seven
  array elements.
  Create an object representing the static text field
  Pick a "words of wisdom" text randomly from 'arrWow'
  Show that text in the static text field.
  return
```

Have a look at it the code that implements this method in **Wow2.rex**. Note that the penultimate pseudocode statement above - "show that text in the static text field" - is implemented in two steps. First the statement **newText = self~newStatic(101)** creates an object that represents the static text control, the resource number 101 defining the control to be represented. Second, that object is used to change the control's text in the statement **newText~setText(arrWow[i])**.

Finally, you may ask why a full code listing is not shown here. The reason is that I'd really rather not. Why not? Doesn't it work? Well, yes, the code works OK. But the design is not good. Aside from performance considerations (the code creates and populates a new array each time the button is clicked), there is absolutely no separation of design concerns. And this is arguably the most important thing for any application - especially one with a GUI. There are three important areas of design concern - UI display and interaction, the "business" that the dialog performs (here the business of selecting the text), and the provision of persistent data (here the seven strings comprising the "words of wisdom"). But we'll start to fix this in the next chapter.

Re-Structuring the Code

The current code is not good. It works - but only because it's very simple. The problem is its design - its structure. There are three quite different concerns which, for all but he simplest of programs, should be separated. These are: the user interface (aka UI or GUI) including both presentation and user action; the data (in our case a set of text strings); and the "business" concept that we're implementing. And the "business" of this code is picking a single text string from a set of "words of wisdom" strings. The code in the **Exercise03** folder separates these concerns, with no change as far as the user's concerned.

The three areas of concern have a relationship with the Model-View-Controller or MVC concept (see *Model-View-Controller*[1]). However, the role of the Controller in classic MVC is handled largely by ooDialog and the underlying Windows UI platform. This leaves us with the View and the Model, where the Model is the "business" - that is, an implementation of the relevant part of the real business - and the View is the part that provides for user interaction. But these two concepts - View and Model - are insufficient. To these two must be added data - that is, data-on-disk (aka "persistent data"). So three areas of concern are required: View, Model, and Data.

At this point, the reader may wish to skip the next three paragraphs which provide a rationale for the View-Model-Data terminology, and introduce the concept of software "components". However, please come back here if later the use of the term "component" is not obvious.

While the model-view-data scheme works well for the PC-resident single-user applications introduced in this Guide, it does not scale to distributed systems with multiple concurrent users where, aside from anything else, the data is on a remote server or available from a remote service. For such systems, additional architectural concepts are required. See, for example, chapters 1 and 2 of "Enterprise Service Oriented Architectures" by McGovern, Sims, Jain & Little; or "Business Component Factory" by Herzum & Sims. Dealing with large-scale distributed systems, both of these references use the terms "user", "workspace" and "workspace-resource" instead of "view", "model" and "data". Although the semantics are identical, this document uses the latter terms since they are both simpler and shorter.

Model components implement the essence of an application. Views enable the user to take action and see the result. Data components know where the data is, and handle the mechanics of reading and writing to disk. (For distributed multi-user applications, the Model would invoke some service on a back-end server, where there would probably be another kind of Model component which in turn would use a separate Data component that accesses a corporate database or remote service).

But why use the term "component" instead of "class"? The answer is (as will be seen in later exercises) that for industrial-strength apps, a component generally consists of a number of classes. And a single class can seldom be independently "plugged in" to a runtime environment without the other classes required fully to implement a single business concept in its View, Model or Data role. A component, however, is intended to be "pluggable" into the runtime, since it is all and only the implementation of one of the "view", "model", or "data" aspect of a business concept. This distinction between class and component is not so obvious using ooRexx as it is with compiled languages, where classes are seen by the developer in source code, but the artifact that is loaded into the runtime is a compiled *.exe or *.dll. Thus one of the purposes of a component is to extend the concept of low coupling and high cohesion into the runtime. Finally, the name given to the set of "view", "model" and "data" components that implement a given business concept such as "Customer" is "Business Component". The interface of a Business Component is defined as the interface of the "model" component.

---

[1] http://en.wikipedia.org/wiki/model-view-controller

Let's now look at the implementation of each of these three areas of concern. In the second part of this chapter, we'll further reduce coupling.

3.1. Fixing the Structure

First, re-run **Wow2.rex** in the **Exercise02** folder, and then run **Wow3.rex** from the **Exercise03** folder. To the user, they're identical. However, in **Wow3.rex** the code has been re-structured so that there are now three different classes, each implementing one of the three areas of concern. We'll look at each class in turn, but first here's a whiteboard-level picture of how the three classes interact to produce a "words of wisdom" string on the screen.
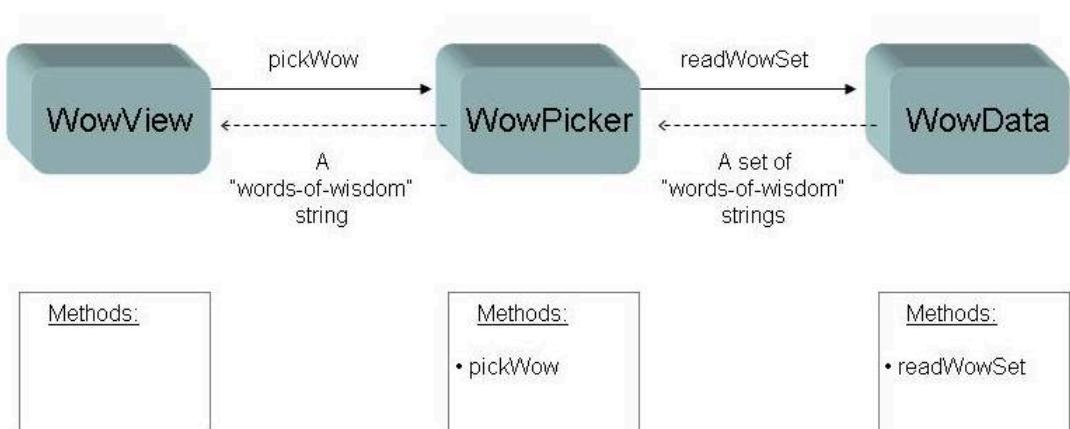
Figure 3.1. Exercise03 Structure

Now look at the **Wow3.rex** code. It consists of the three classes shown in the diagram: **WowView**, **WowPicker**, and **WowData**. "Wow" in the names is short for "words of wisdom".

3.1.1. The "View" Component

The **WowView** class implements the UI area of concern (it's a single-class component). The **init** method is identical to that of Exercise02's **Wow2** except that it also creates an instance of **WowPicker** called (unsurprisingly) **wowPicker**. There's also an **expose** statement to make the **wowPicker** object available to other methods.

The **defineDialog** method has not changed. But the **okClicked** method is different - and much shorter:

```
::method okClicked
  expose wowPicker newText
  wow = wowPicker~pickWow
  newText~setText(wow)
  return
```

To get the "words of wisdom" to display, it now merely asks **wowPicker** for a string to display.

One other change is that instead of creating a new static text control every time the button is pressed, the control is created once in the new **initDialog** method and re-used in the **okClicked** method. The **initDialog** method is called automatically by ooDialog after the dialog has been created in order to allow controls to be initialized (the "init" in "initDialog" stands for "initialize").

In summary, all knowledge of picking a string, and of the set from which to pick, has been exported elsewhere. The **WowView** class now addresses only the areas of GUI display and GUI interaction. This

is crucially important. A good way to make a complex task hopelessly complicated is to mix "model" and "data" concerns with the "view" concerns.

3.1.2. The "Model" Component

The class implementing the model component - **WowPicker** - is very simple:

```
    ::METHOD init
      expose wowSet
      dataSource = .WowData~new
      wowSet = dataSource~readWowSet
      return

    ::METHOD pickWow
      expose wowSet
      i = random(1,7)
      return wowSet[i]
```

The **init** method gets a reference to an instance of the class **WowData** - which handles the data area of concern - and then gets a set of Words of Wisdom into the array variable **wowSet**. Then in the method **pickWow** a Words of Wisdom string is picked randomly from **wowSet** and returned.

3.1.3. The "Data" Component

The last (extremely simple) class in **Wow3.rex** is **WowData**. In its **init** method, it loads up an array of seven text strings into the instance variable **arrWow**, and in its **readWowSet** method returns that array to the caller. One can see how this it might be enhanced, for example by providing a method that renews the set of "words of wisdom" from a larger set in a disk file.

3.2. Reducing Coupling

The three classes in **Wow3.rex** are reasonably decoupled: the dialog is in one class, the business logic (such as it is) in another, and the data in a third. Notice however that both **WowView** and **WowPicker** create a reference to another class (**WowPicker** and **WowData** respectively) in order to invoke them. Each of these three classes can be called a "main" class, since each is the main (and in this case only) class implementing a separate area of responsibility. In more complex applications, each component (area of responsibility) will have one main class and a number of subsidiary classes - for example, a (main) SalesOrder class with subsidiary OrderLine and DeliveryInstructions classes. The intent of a component is to be, as much as possible, a self-contained unit of business function.

Now, when considering more complex applications, it is arguable that it is not the responsibility of either class to know about the creation of instances of other classes. Later we will see that, for each important business concept (such as SalesOrder, Customer, or Product), each component (view, model, and data) will have a number of classes, and each will have one main class expressing the core of the business concept. If these three kinds of component are to be as independent as possible, then each should know as little as possible about the others. Such independence is usefully enhanced if a way is found to move the knowledge of how to get references to the main classes to a fourth area. And there is just such an area - the application.

In ooDialog programs, there is often a block of code at the beginning of the program file that kicks off a dialog by instantiating an ooDialog class. From there, all the behavior is in the dialog classes. This "kick-off" block of code can be used to reduce coupling by pre-instantiating the main classes, and storing the object references in **.local**. Thus no main class has to know how to instantiate any other main class. But, when a main class gets the object reference for another main class instance, doesn't the first class have to know the correct name of the object reference in **.local**? Well, yes, but

even that could be fixed - for example by providing a business-oriented instance reference such as Customer Number, and having some third party object handle the instantiation.

An example of decoupling the three areas of concern is provided in the **Exercise03** folder. Try running **Startup.rex**. It behaves exactly as **Wow3.rex** does. However, the code is now structured into four *.rex files: **Startup**, **WowView**, **WowModel**, and **WowData**. The code in **Startup.rex** is very simple:

```
.local~my.idWowData = .WowData~new
.local~my.idWowPicker = .WowPicker~new
 dlg = .WowView~new

.local~my.idWowData~activate
.local~my.idWowPicker~activate
dlg~activate

::REQUIRES "WowView.rex"
::REQUIRES "WowModel.rex"
::REQUIRES "WowData.rex"
```

The first three statements create the three classes, with the ids of the first two being stored in **.local**. Creation of the dialog is done by the third statement (**dlg = .WowView~new**). The next three statements send an **activate** message to each of the three classes. This is because when dealing with complex applications with "main" classes (each in their own component), it is very useful to distinguish between two kinds of class setups: firstly the technical creation of a class (done by invoking the **init** method), and secondly the initial setup of various application-related things (done by invoking an **activate** method.

Notice that **WowView**'s **init**> method returns to the caller. The **activate** method, on the other hand, does not return until the dialog is closed. This is because the statement that actually surfaces the dialog - **self~execute("SHOWTOP", IDI_DLG_OOREXX)** is the last statement in **WowView**'s **activate** method. And once the dialog is surfaced using **SHOWTOP**, control only returns to the application (that is, to **Startup.rex**) when the dialog is closed (although, as will be discussed later, there are ways to return control much sooner).

But why move the **self~execute("SHOWTOP"...)** statement into the **activate** method of the **MyDialog** class? After all, it would work just as well if it were the last statement in the Startup file. The reason is that the business of surfacing the dialog window is arguably not that of the application; rather it's the business of the view's class object. Thus the application is reduced as much as possible to a simple "kickoff" script, while the real work is done by the classes that are kicked off.

There is, however, one important consequence of this move. Since the **self~execute("SHOWTOP"...)** statement does not return until the dialog is closed, the method blocks on this statement, and there is potential for a hang. In **Wow3.rex**, this statement was at the end of the "application" part of the program, and as there was nothing after it, the block didn't matter. But re-factoring the classes into different files has moved it to the **activate** method of **WowView**. This introduces a concurrency issue. If not dealt with, then when the user clicks the **More wisdom** button, the **okClicked** method can not run until the **activate** method ends - that is, until the user closes the dialog window - a real catch 22, where the result is that no words of wisdom will appear.

The reason **WowView** works is because its **activate** method has the **unguarded** option specified on its method statement. Try commenting "UNGUARDED" out and running the exercise without it.

As a general rule, event handling methods such as **okClicked** should be unguarded. Indeed, **WowView** runs happily if the "unguarded" option is moved to the **okClicked** method statement - or indeed if it's on both method statements.

In the next chapter, we leave "words of wisdom", and start building a more realistic application.

Using Resource Dialogs

This chapter starts to build the components of the eventual sample application. The completed application will be a somewhat simplistic sales order processing application, and will look something like this:

Figure 4.1. The Sales Order Management Application

The purpose of this application is to provide a vehicle for exploring various ooDialog concepts and facilities, and this chapter addresses the use of "resource files" in the context of a "View" component called "CustomerView" - that is, a view of a customer.

Designing what a dialog will look like on the screen involves positioning and sizing various controls such as edit fields, lists, buttons, menus, etc., as well as defining how the window itself will appear. The resulting set of control and window definitions is called a "resource definition" or "dialog template". A file that contains a dialog template is called a "resource file". There are two kinds of resource file: a "resource script file" that's human-readable with (say) Notepad, and a compiled "binary resource file".

ooDialog provides two classes that read their dialog template from a resource file: **RcDialog** and **ResDialog**. The former gets its resource template from a resource script file, the latter from a compiled binary resource file (a dll). This chapter addresses the use of **RcDialog**; the next chapter discusses **ResDialog**.

The easiest and arguably the best way to define the layout of a dialog template is to use a "resource editor". A resource editor is a "wysiwyg" (what you see is what you get) development tool that allows a developer to design a window layout visually. The output is a resource file. This avoids the sometimes tortuous effort of laying out the dialog programmatically. Although using a resource editor is certainly not the be-all and end-all of ooDialog programming, it's very useful for getting started quickly, and is the recommended way to define ooDialog window layouts.

The vehicles for exploring resource files will be the Customer View and (in the next chapter) the Product View parts of the sample application. Although simplistic, these parts of the eventual order management application are sufficiently complex for some naming and coding conventions to be useful, and the next section describes these conventions. Then the use of a resource script is introduced in the context of the "CustomerView" dialog. Finally, the three major parts of a dialog are discussed.

4.1. Naming and Coding Conventions

4.1.1. Naming Conventions

Readers may prefer to skip this section, at least for the time being, and go straight to Resource Scripts and Resource File Editors (*Section 4.2, "Resource Scripts and Resource File Editors"*).

In Chapter 3 there was a brief discussion about separation of concerns into three areas: the UI including both presentation and user action, the "business" or rather the "model" of the business, and accessing data. From here onwards, this approach becomes an important convention for the structure of the sample Order Management application. Essentially we adopt a "component" approach to the application. Thus the "customer" concept is implemented by three "main" classes: **CustomerView**, **CustomerModel**, and **CustomerData**. That is, the naming convention used to distinguish between the three different kinds of "main" classes is to append one of the suffices "View", "Model", or "Data" to the class name. Each of these main classes is a component in its own right, as opposed to subsidiary classes such as an "address" class used within a Customer View main class. Such subsidiary classes are generally included in the same file as the main class (but in cases where several main classes use the same subsidiary class, they are usually stored in a separate file). The name given to the group of main classes that contribute to a single important business concept such as "customer" is "business component". Thus in the sample application, CustomerView, CustomerModel and CustomerData are three "main" component parts of the "Customer Business Component". (By the way, the interface of a business component is generally considered to be the interface of the Model main class). So - components can be made up of other components.

Normally, each main class (plus any subsidiary classes) would be in its own file. However, since the focus is on View components, the Model and Data components are placed in a single file, called **xxxModelData.rex**, where "xxx" is the business component name such as "Customer".

By the way, in real-life systems there would probably be four parts to a concept such as "Customer" - a view and a user-oriented model both supporting the user, and, supporting multiple concurrent users on a server or back-end system, a business-oriented "model" plus a data part that accesses the corporate database. Also by the way, in real-life supply chain management applications, addresses are typically treated as separate entities rather than being lumped in with such concepts as Customer, Employee or Supplier.

Finally, variables often have a prefix that indicates what the variable is. For example, an edit control that holds a customer number would be named **ecCustNo**, the **ec** being short for "edit control". And a data-only class such as an address is prefixed by **bt** for "business type"; so an address class would be named **btAddress**.

4.1.2. Coding Conventions

The following coding conventions are used in the exercise code. First, ooRexx directives and their options are capitalized. Second, classes, methods, and routines are separated from each other by dotted or solid lines which in some editors are displayed in a different color from the executable code. This provides visual separation of methods and classes which is useful in larger programs. Third, camel case is used for variable names, with class names having their first letter capitalized. Finally, when an ooRexx program in one of the exercises is run, comments produced with an ooRexx "say" instruction may appear in the command prompt window. The format used as a prefix for such comments is **class-method-nn** - a little excessive for simple single-class programs, but useful for larger multi-class applications.

4.2. Resource Scripts and Resource File Editors

Our first foray into the sample Order Management application is to examine a simple Customer View component built using a resource editor.

But which resource editor? Well, if you happen to have Microsoft's development kit, you'll find it has a resource editor. Alternatively, there are a number of fee and free resource editors available on the web. The author of this Guide happened to use a freeware product called "ResEdit", available from this link:

ResEdit Home Page[1]

Occasional hints about ResEdit usage will appear from time to time. In addition, comments about the use of resource file editors will assume ResEdit, and may well be inapplicable to other resource editors. If you plan to use ResEdit, please be aware that a number of Microsoft header files are required. These can be obtained at no charge from Microsoft Windows SDK under "Developer Tools" at this link:

Microsoft Windows Software Development[2]

Or do a web search for "microsoft windows sdk". The header files should be downloaded into a folder, and the full path name of that folder must be specified to ResEdit in "Options - Preferences - General - Include paths".

A resource file editor outputs a window layout to a "resource file", which ooDialog can then use to lay out controls on a dialog automatically. There are two kinds of resource file: a human-readable file with the extension ".rc" (and sometimes ".dlg"), and a binary (compiled) file with the extension ".dll".

Locate the folder **Exercise04**, and run **Startup.rex**. You see a "Customer" dialog. Explore the menu and behavior of this dialog. Note the following:

- A number of comments appear on the console; ignore them for the time being.

- The title bar (the blue bar right at the top of the dialog window) shows the string ***CustomerName*** rather than the Customer's name, suggesting that the programmer has either made an error or (as in this case) has left a marker for future modification.

- Edit controls are shown grayed out or "disabled" - that is, not editable.

- The "Action" menu has four items.

- One button - "Record Changes" is disabled, the other is not.

Make sure you exercise the menu items and buttons to explore the dialog's behavior. You'll find that some expected behavior is not implemented, and results in a message-box - for example **"The 'Print...' menu item is not yet implemented."**. Note also the tab order - that is, the order of controls reached as you press the tab key. This is defined by the sequence in which controls appear in the *.rc file. If the tab order is not as you'd like it, you can edit the *.rc file with NotePad (or some other text editor) and use cut-and-paste to achieve the desired tab order.

By the way, note that the menu item **Last Order** and the pushbutton **Show Last Order** should produce the same result, but they don't. This is nothing more than a development trick to check whether the right event-handler is invoked by the right control. An alternative is to use a **say** instruction. And, of course, a given function should use a single method, no matter how many different ways it's invoked.

---

[1] http://www.resedit.net/
[2] http://msdn.microsoft.com/en-us/windows/bb980924

Now double-click the file **CustomerView.rc** in the **Exercise04a** folder. The file should open in ResEdit (or your own preferred resource editor). In the ResEdit "Resources" window, click on **IDD_CUST_DIALOG** and the dialog layout tool opens, looking like this:
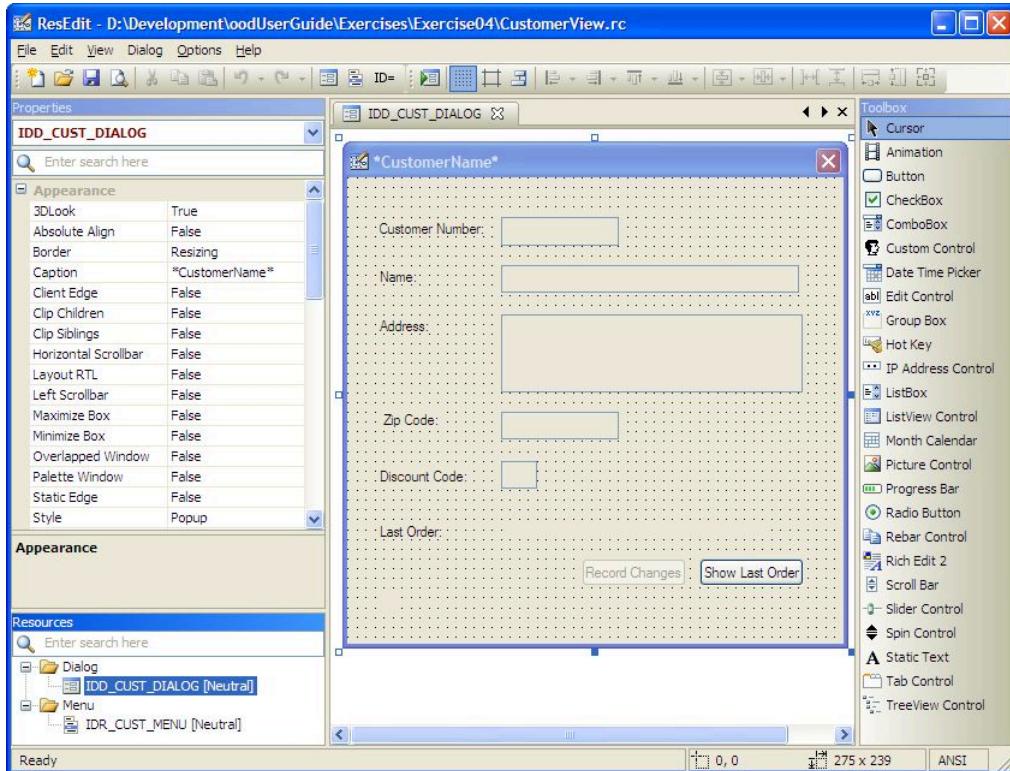
Figure 4.2. A Resource Editor

You might move or re-size some of the controls, save the file, then re-run to see your changes implemented. Check the files in the **Exercise04** folder. The files needed by ooDialog to create the window are **CustomerView.rc** and **CustomerView.h**. Both of these are generated by the resource editor. Why two files? Well, names for resources in the .rc file are intended to be reasonably comprehensible - e.g. **IDC_CUST_EDT_CUSTNO**. But Windows requires resources at run-time to be identified by numbers. The mapping between resource names and resource numbers is done in the .h file. (ResEdit tip: to cause the .h file to be named the same as the .rc file, on the menu bar select **Options - Preferences - Code Generation - Files**, then set the **Header file name** to **%barefilename %.h**.)

Finally, a hint from hard experience. Some resource editors have been known, very occasionally, to assign the same ID number to two different controls in the .h file, or to omit a resource from the .rc file. So, if some error occurs which, on re-checking the code, seems inexplicable, it could be worth checking the .h file to see whether the same number has been assigned to two symbolic IDs. If so, you can try hand-editing the .h file then re-starting the resource editor. If the .h file looks OK, then you might check the .rc file to see if all the resources are there.

4.3. Coding an RcDialog Class

Having discussed coding conventions and resource editors, this section now looks in detail at the code in the **Exercise04** folder. First, look at **Startup.rex** in an editor. Aside from creating and activating **CustomerData** and **CustomerModel** classes in the same way as in the previous exercise, there's only one other executable statement: **call startCustomerView**. This routine is in the **CustomerView.rex** file (it's generally good practice to separate application startup concerns - such as (in this case) creating new dialogs - from the various working parts of the application).

Now look at **CustomerView.rex** in an editor. Look for the **CLASS** statement:

```
::CLASS "CustomerView" SUBCLASS RcDialog PUBLIC
```

CustomerView is a subclass of the ooDialog-provided class **RcDialog**, which gets its dialog layout from a resource script file that is human-readable (using a text editor). RcDialog is one of two important ooDialog classes that use resource scripts; the other is **ResDialog**, which uses a binary (compiled) resource file as illustrated in the next chapter. More information on resource files can be found in the ooDialog Reference.

View classes can be seen as consisting of three major parts: setting up the dialog window, specifying the "active" controls (i.e. controls that need to be accessed programmatically), and handling the application data and function. Let's look at each of these in the context of **CustomerView.rex**.

4.3.1. Setting Up the Dialog Window

When you ran **StartUp.rex**, there were an initial set of comments displayed in the command prompt window, as follows:

```
StartCustomerView Routine-01: Start.
CustomerView-init-01.
CustomerView-createMenuBar-01.
StartCustomerView Routine-02: dlg~activate.
CustomerView-activate-01.
CustomerView-initDialog-01.
CustomerView-getData-01.
CustomerModel-query-01.
CustomerData-getData-01.
CustomerView-showData-01.
```

These comments trace the process of establishing the dialog to the point of making the dialog visible and getting the application data to display - in other words, setting up the dialog. One routine and seven methods of **CustomerView** are involved, as follows:

1. First, the routine (at the end of the **CustomerView.rex** file) uses the **.Application** class to set application defaults in the statement **.Application~setDefaults(...)**. The first parameter - "O" for "only" - specifies that only the *.h file provided as the second parameter is to be used for symbolic IDs. The IDs in this file are added to the application's global constants directory (aka "globalConstDir"). The third (optional) parameter turns autoDetection off. Try commenting out this third parameter to see the result of leaving autodetection switched on (which is the default). Also, see *Appendix A, Dialog Attributes and AutoDetection* for a discussion of what autodetection is and where it may be useful.

    The routine then creates an instance of the **CustomerView** class as a subclass of **RcDialog**.

2. In the **init** method of the new view instance, first the superclass is invoked (this is an ooDialog requirement), and then the **createMenuBar** method is called. Note that if the menubar creation fails (i.e. returns **.false**), then arguably the dialog should not be created. In this case, **initCode**, which is an attribute of the **.Dialog** class, should be set to a non-zero value. This attribute represents the success of initialization of a dialog object. After the **init** method of the **RcDialog** superclass has executed, **initCode** will be zero if the dialog initialization detected no errors, but will be non-zero if initialization failed or an error was detected.

3. The **createMenuBar** method creates a menubar (in this case an instance of the **ScriptMenuBar** class) that specifies the name of the *.rc file and also the menubar's symbolic ID in both the *.h file and the *.rc file. Note that after creation, the menubar is just another object, and is not yet associated with the dialog. The code at this point boldly assumes that the menubar instance was successfully created (not really best practice) and returns to the **init** method and from there back to the ...

4. ...**StartCustomerView** routine, which invokes the dialog's **activate** method.

5. The **activate** method issues **SHOWTOP** to the view's superclass, which then sends itself an **initDialog** message.

6. The **initDialog** method attaches the menubar to itself (that is, to the dialog instance). The remainder of the method specifies the active controls (addressed in the next section), and finally invokes the **getData** and **showData** methods.

The above sequence may seems a little heavy just to show a dialog. But much of it is concerned not only with getting the data to show in the dialog's controls, but also with providing for the user to modify that data. Focusing only on what is required to display the dialog with no data, then the process requires only four methods and a total of 20 ooRexx statements including the **::Method** statements but excluding the **say** instructions. And if we didn't care too much for effective program structure or error checking, it could be squished down to just ten instructions as follows:

```
::ROUTINE startCustomerView PUBLIC
  .Application~setDefaults("O", "CustomerView.h", .false)
  dlg = .CustomerView~new("customerView.rc", IDD_CUST_DIALOG)
  dlg~activate
::CLASS CustomerView SUBCLASS RcDialog PUBLIC
  ::METHOD init
    forward class (super) continue
    self~execute("SHOWTOP")
  ::METHOD initDialog
    menuBar = .scriptMenuBar~new("CustomerView.rc", "IDR_CUST_MENU", , , .true)
    menuBar~attachTo(self)
```

And if the **::class**, **::method**, and **::routine** directives are excluded, only six statements are required: defining **CustomerView.h** as the *.h file, the **.CustomerView~new** to create a dialog instance, the call to super in the **init** method, issuing **execute("SHOWTOP")**, creating a menubar, and attaching the menubar to the dialog.

In other words, dialogs of significant complexity can be created and displayed with only six executable statements. And *that* is the real power of resource dialogs.

4.3.2. Specifying the Active Controls

An "active control" is a control that requires behavior to be programmed, while a "passive" control (such as text that never changes) appears only in the resource file, and is of no concern to the program. The behavior associated with an active control is of two kinds: outbound or program-to-screen - i.e. providing the user with information; and inbound or keyboard/mouse-to-program - i.e. signaling the program about a user event. Outbound behavior means changing the state of a control - for example, disabling a pushbutton, or displaying text in an edit control. Inbound behavior is a user event that requires the program to take some action - e.g. the user selects a menu item, or clicks a pushbutton. Much inbound behavior is ignored by the program (e.g. the user placing the cursor in an edit control). For both inbound behavior that is relevant to the program, and also for outbound behavior, the relevant controls must be made available to the programmer as ooRexx objects.

Now controls are actually created by Windows, based on information in the resource file, with each control being created and managed by facilities built into the Windows operating system. However, the ooRexx programmer accesses controls via instances of ooDialog classes, so that each control on a window is represented by an ooRexx object in the ooRexx dialog code that serves as a proxy for the underlying Windows control. And it is ooDialog that creates the required link between such ooRexx objects and the underlying Windows controls - and hence between the ooRexx object and the visible controls on the screen. This is illustrated in the following diagram
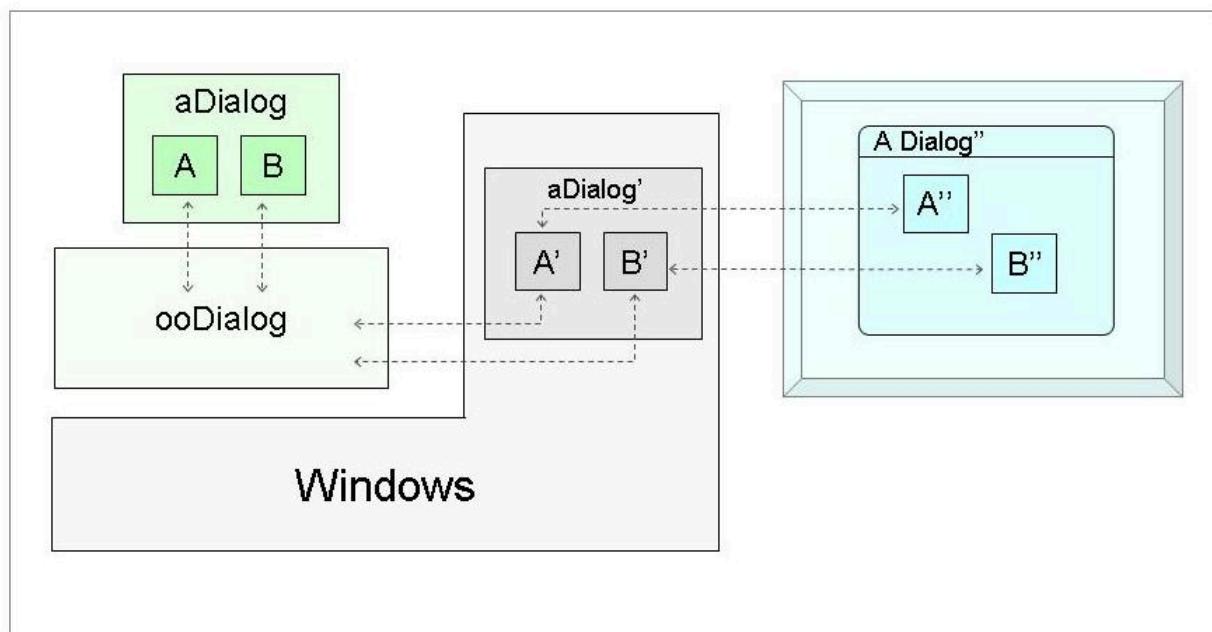
Figure 4.3. ooRexx Proxy Controls and Real Windows Controls

In the diagram, A and B are the ooRexx proxy controls in an ooRexx dialog instance (aDialog). When the dialog is to be displayed, ooDialog communicates with the Windows runtime and directs it to create a native Windows dialog (aDialog'), complete with controls A' and B'. This is known as the "underlying dialog" (see the ooDialog Reference). Windows then displays this dialog on the screen (A Dialog"), where the controls (A" and B") are visible to the user. From then, any user interaction with the visible dialog goes between the screen and the underlying dialog. Of course, ooDialog hooks into these interactions, and makes them available to the ooRexx dialog (e.g. by invoking its event handler methods). By the way, and rather obviously (but we'll say it anyway), this means that ooDialog cannot provide any GUI function that is not already provided by the underlying Windows facilities.

To manage controls, ooDialog provides an ooRexx class for each control type. The link between an ooRexx proxy control and the corresponding control in the underlying dialog is created via the control's symbolic ID in the **.rc** and **.h** files. Creating the ooRexx control proxies is typically done in the **initDialog** method. In the **CustomerView** code for example, in order to display the Customer Number in an edit control (outbound active behavior) an ooRexx proxy is created in the **initDialog** method as follows (where **custControls** is a directory object that makes the **expose** statements shorter):

```
custControls[ecCustNo] = self~newEdit("IDC_CUST_EDT_CUSTNO")
```

The item associated with the index **ecCustNo** is the proxy ooRexx object for the Windows edit control that will contain the customer number; **self** is the dialog instance; **newEdit** is the method of the Dialog Object (see the ooDialog Reference) that creates the ooRexx proxy for the underlying Windows control; and **IDC_CUST_EDT_CUSTNO** is the control's symbolic ID from the **.h** file. After execution of

the statement, **custControls[ecCustNo]** is an instance of the ooDialog **Edit** class (that is, an instance of the proxy edit control), and ooDialog has made sure, in the instance's creation, that it is internally linked via the underlying dialog to the edit control on the screen identified in the .h and .rc files as **IDC_CUST_EDT_CUSTNO**.

To avoid tedious repetition, from now on this document will assume an understanding of the relationship between an ooRexx proxy instance and the instance in the underlying dialog.

A number of other outbound active controls are created in the **initDialog** method - as many as there are fields on the dialog that need to have data placed in them when the dialog opens. In addition, a **Record Changes** pushbutton proxy is created so that the button can be enabled (outbound active behavior) when a user chooses the menu option **Update...** (inbound active behavior).

After this, the following statement appears:

```
self~connectButtonEvent("IDC_CUST_BTN_RECORDCHANGES","CLICKED",recordChanges)
```

This is an example of specifying an "event handler" (inbound active behavior). Suppose the user presses the **Record Changes** button. The Windows runtime signals the event, which ooDialog picks up. The above statement declares that this event - i.e.that the pushbutton identified in the **.h** file as **IDC_CUST_BTN_RECORDCHANGES** has just been **CLICKED** - will invoke the **recordChanges** method. In other words, the statement defines **recordChanges** as the event-handling method for the **Record Changes** pushbutton. The same is done for the **Show Last Order** pushbutton, where the event handler is specified to be the method **showLastOrder**.

Notice that each of the event handler methods are specified as **UNGUARDED**. In general, an event handler should be unguarded to preclude the possibility that some guarded method in the dialog object is executing at the time the event notification is generated. For further information, see the ooDialog Reference. Note also that event-handling methods must be **PUBLIC**, since they are invoked from outside the ooRexx dialog class by the underlying ooDialog code (and of course an ooRexx method is public unless **PRIVATE** is specified).

Specification of active controls is generally done in the **initDialog** method. Indeed, in the **CustomerView** class, specification of active controls occupies most of this method.

Note that menubar actions are not specified. This is because the menu items in **CustomerView.rex** are "auto-connected" (see "Menu Command Event Connections" in the ooDialog Reference). Auto-connection is specified in the last parameter of the following **.ScriptMenuBar~new** statement in the **createMenuBar** method:

```
menuBar = .ScriptMenuBar~new("CustomerView.rc", "IDR_CUST_MENU", , , .true)
```

Setting this parameter to **.true** (the default is **.false**) specifies that all menu items will be connected automatically to a method with the same name as the visible caption or text. In **CustomerView.rc** the "Actions" sub-menu is:

```
    MENUITEM "New Customer...", IDM_CUST_NEW
    MENUITEM "Update...", IDM_CUST_UPDATE
    MENUITEM "Print...", IDM_CUST_PRINT
    MENUITEM "Last Order", IDM_CUST_LAST_ORDER
```

Spaces and trailing dots are stripped, giving method names of "NewCustomer", "Update", "Print", and "LastOrder". In the "MenuBar Methods" part of the **CustomerView** code, a method is provided for each of these menu items. Note that the **print** and **newCustomer** methods do nothing other than show a messagebox saying that the function is not implemented. Best practice suggests that

an explanatory message is much better than the alternative (to see what this alternative is, try commenting out the **print** method).

But before the menu actions will work, the **menuBar** object must be associated with the dialog object. This is done by this statement (at the beginning of the **initDialog** method):

```
        menuBar~attachTo(self)
```

By the way, an alternative approach is to create the menubar in the **initDialog** method, and attach the menu at the same time using the previously omitted 6th parameter, **self**:

```
  menuBar = .ScriptMenuBar~new("CustomerView.rc", "IDR_CUST_MENU", , , .true, self)
```

In this case, the statement **menuBar~attachTo(self)** should be omitted. The point here is that while the menubar can be created any time, it cannot be attached until the underlying dialog has been created; that is in the **initDialog** method at the earliest.

At this point, the dialog is displayed complete with all its controls. But there is no data shown. When executed, it looks as if the data appears at the same time as the window, but it does not. To illustrate this, insert two **call SysSleep(2)** statements, one just before the statement **menuBar~attachTo(self)** and one just after. Run the program and you'll see the window without menubar, then the menubar will appear, and then the data.

The last two statements in the **initDialog** method kick off the initial parts of the Application and Data Function category. The first invokes a method to retrieve the data for this customer, the second to display it. At which point the dialog sits back and waits for the user to do something.

4.3.3. Application Data and Function

Designing the application function/data-handling part of a main view class is more complex than is often thought. The designer has to consider the various possible states of the dialog, and also which state transitions are valid. Sometimes state and state transition charts are used to plan and record UI interactions. And, in doing this design work, the first consideration is the user. Indeed, providing what the user needs and likes is probably the most difficult aspect of GUI development. But who is "the user"? Well, this document would be going well outside its remit to embark on addressing this question. Suffice to say that there are a number of sources for information on usability, among which one of the author's favorites is "The Inmates Are Running The Asylum" by Alan Cooper. But here, the main concern is use of ooDialog rather than UI design per se, and so in this document, the subject of UI design must take a back seat.

In the case of **.CustomerView**, the application behavior is very simple:
- On initial display of the CustomerView instance, populate the controls with data. This is done by invoking (at the end of the **initDialog** method) the **getData** and **showData** methods. The first gets the data for this customer (hard-coded in the **CustomerData** class), and the second displays that data. The dialog then waits for user input.

- On **Update** being menu-selected, the **update** method is automatically invoked. This method first enables the edit controls so that the user can modify the data, and then enables the **Record Changes** button. Looking at the code, you'll see that some methods operate directly on the control, while others operate on the dialog, with the control's symbolic ID being provided as a parameter. Although not a hard and fast principle, the distinction, loosely, is that operating directly is done where there is no ambiguity (e.g. changing the state of an edit control from read-only to read/write), whereas operating indirectly through the dialog is done where the action is in the context of the window (e.g. setting the input focus on a control and hence off another).

- When the **Record Changes** button is pressed, the `recordChanges` method is invoked. This first checks whether anything has in fact been changed. If it has, a comment is output to the console, and the state is set back to the starting position with the **Record Changes** button and edit controls disabled. If nothing has been changed, a message box is displayed.

- Finally, several minimal or dummy actions are provided as place-markers for possible future use: three menu items (**New Customer...**, **Print...**, and **Last Order**) and a **Show Last Order** pushbutton.

The above function is delivered through nine methods: five event handler methods (three for menu items and two for pushbuttons) and four methods supporting the event handlers. Between them, they deliver the application and data function. The next section examines the ooDialog aspects of the application function.

4.3.3.1. The getData and showData Methods

The `getData` method retrieves data from an instance of `CustomerModel` (which in turn gets the data from an instance of `CustomerData`).

The `showData` method uses the `setText` method (see the ooDialog Reference) to set the text of the various controls to the customer's data. There are two things to note here:

- First, each control is in fact a separate window in its own right. Thus the `setText` method can be used to set the text for any control. For example, the text on a pushbutton can be changed using this method. To check this out, try inserting this statement at the end of the `update` method:

```
        custControls[btnRecordChanges]~setText("Press me")
```

When "Update" is menu-selected, the text on the button changes.

- Second, the Customer Address data is an array, which for display in a multi-line edit control must be transformed into a text string with line-ends inserted at appropriate places. This is done in the `showData` method. Data transformation of this sort is very usual within view classes; after all, it's the responsibility of any View class (or of its subsidiary classes or routines) to handle any re-formatting for display purposes.

4.3.3.2. The update and recordChanges Methods

The `update` method "enables" the edit controls and the **Record Changes** button so that the user can make changes and then make the changes permanent (i.e. "record" them). Enabling edit controls is done by sending them the message `setReadOnly` with the parameter `.false`. For example:

```
    custControls[ecCustName]~setReadOnly(.false)
```

Pushbuttons are enabled by invoking `enableControl` on the dialog with the control's symbolic ID as the single parameter, as shown in the first statement below. The second statement below puts focus on the push-button - in this case by invoking the button's `state` method. Finally, the cursor is placed in the Customer Name edit control by invoking the dialog's `focusControl` method.

```
    self~enableControl("IDC_CUST_BTN_RECORDCHANGES")
    custControls[btnRecordChanges]~state = "FOCUS"      -- Put focus on the button
    self~focusControl("IDC_CUST_EDT_CUSTNAME")          -- place cursor in the CustName edit
  control.
```

The dialog is now in a state whereby the user can make changes to the data. When the user presses the **Record Changes** button, the `recordChanges` method is invoked. Processing from this point is almost all plain ooRexx with little ooDialog involvement:

- The **recordChanges** method reads data from dialog controls using the **getText** and **getLine** methods of the Edit Control. Any view-formatted data is transformed into application format (in this case, the Address needs to be transformed from strings with line-end characters to an array).

- Then the **checkForChanges** method is invoked with, as a parameter, the data just read in from the dialog controls.

- If the data has not changed, a message box is displayed. If it has changed, then the old data is replaced with the new. Finally, in either case, the edit controls are set to read-only, and the "Record Changes" button is disabled.

Suppose in the middle of updating, the user presses the Escape key by mistake? Try it. The dialog disappears - together with any changes made. This is certainly not best practice, and is addressed in the next chapter (see *Controlling Dialog Cancel*) which discusses the use of ooDialog's **ResDialog** class. A dialog subclassed from **ResDialog** uses a compiled resource file (a *dll file) instead of the .rc file required by an **RcDialog** subclass.

Using Binary Resource Dialogs

This chapter uses a "Product View" class as the context for discussing the following topics: first, *Dialog Initiation*; second (*Using a Binary Resource File*) (that is, *.dll files) and the differences in using these as opposed to script resource files; third *Dialog Controls* not met in previous exercises; fourth some changes to *Code Structure*; fifth a brief visit to some *Dialog Design* considerations; and sixth, *Controlling Dialog Cancel*.

But first, run **Startup.rex** in the **Exercise05** folder. A ProductView dialog appears. Check out the behavior of the dialog - there are several new behaviors compared to **CustomerView**. In particular, aside from controls not used in previous exercises, the behavior includes more realistic application-level edit checks - that is, implementation of some (fairly trivial) "business rules". For example, menu-select **Actions → Update Product**, then change the UOM (Unit of Measure) from 6 to 20, and then press the **Save Changes** button.

5.1. Dialog Initiation

Previous exercise have used either the "application" or "startup" program, or a separate ooRexx routine, for dialog initiation. By "initiation" is meant the two statements "**dlg=. [DialogClassName]~new**" and "**dlg~execute()**". In other words, the responsibility for issuing these two initiation statements - which are essential for the creation of the dialog - have previously been outside the dialog class. If they could be moved *within* the class, then encapsulation would be enhanced - always a desirable thing. The question is, how? Well, ooRexx has a mature implementation of OO that (among other things) allows for class methods (as opposed to instance methods). Using this feature of ooRexx, the initiation statements can be quite happily moved into a class method. Thus the **ProductView** class has a method called **newInstance** which, with comments and "say" instructions removed, is as follows:

```
::METHOD newInstance CLASS PUBLIC UNGUARDED
  .Application~setDefaults("O", "ProductView.h", .false)
  dlg = .ProductView~new("res\ProductView.dll", IDD_PRODUCT_VIEW)
  dlg~activate
```

The **newInstance** method is invoked from **Startup.rex** by the statement **.ProductView~newInstance**. So all knowledge about initiating a dialog is moved inside that dialog's class, and from now on this approach will be used.   Note also that the first parameter of the **.ProductView~new()** statement allows file paths to be specified.

5.2. Using a Binary Resource File

5.2.1. DLL Compilation

ooDialog's **ResDialog** class (a subclass of **UserDialog**) requires a resource-only DLL. A resource-only DLL is a resource script (*.rc) file that has been compiled into binary (or *.dll) format. Most resource editors have this function. ResEdit is capable compiling a *.rc file, but with three caveats:

- It must be done from the command line:

  resedit -convert filename.rc filename.dll

- At compile time, the *.h file and any *.bmp files must be in the same directory as the .rc file. If present and referenced by the .rc file, *.bmp and *.ico files are compiled into the DLL.

- The version of ResEdit used at the time of writing was 1.5.10-Win32. Comments about usage of ResEdit apply to this version, and may vary in later versions.

At run-time, a ResDialog class needs only the *.dll and the *.h files.

Finally, it's worth mentioning the freely-available Microsoft SDK available at *Microsoft Windows Software Development*[1] which can also be used for compiling resource-only DLLs. The resource compiler is called **RC.exe**, and outputs a *.res file. This is them linked using **link.exe** to produce the DLL. For example, the following illustrates **ProductView.rc** being compiled then linked to produce **ProductView.dll**:

```
C:\>rc ProductView.rc
Microsoft (R) Windows (R) Resource Compiler Version 6.1.7600.16385
Copyright (C) Microsoft Corporation.  All rights reserved.

C:\>link ProductView.res /NOENTRY /DLL /MACHINE:X86 /OUT:ProductView.dll
Microsoft (R) Incremental Linker Version 10.00.30319.01
Copyright (C) Microsoft Corporation.  All rights reserved.
```

Successful compilation depends on both the PATH and the "INCLUDE" environment variable containing the appropriate settings, as follows (at the time of writing and assuming everything is on the C: drive):

```
PATHS
   C:\Program Files\Microsoft SDKs\Windows\v7.0A\bin;
   C:\Program Files\Microsoft Visual Studio 10.0\VC\BIN;
   C:\Program Files\Microsoft Visual Studio 10.0\Common7\IDE

'Include' Environment Variable:
   INCLUDE=C:\Program Files\Microsoft SDKs\Windows\v7.0A\Include;
   C:\Program Files\Microsoft Visual Studio 10.0\VC\include
```

The first two paths should be added automatically when the SDK is installed. If the third (or any of the other two) is/are missing, then add it/them using the PATH command, e.g.:

path=%PATH%;C:\Program Files\Microsoft Visual Studio 10.0\Common7\IDE

To see if the INCLUDE environment variable is present, enter **set** on the command prompt and examine the output. If it is not present, then enter the following in the command prompt:

C:\>set INCLUDE=C:\Program Files\Microsoft SDKs\Windows\v7.0A\Include;C:\Program Files\Microsoft Visual Studio 10.0\VC\include

5.2.2. Differences between RcDialog and ResDialog

The Exercise05 folder contains a Product View component, the main class **ProductView** being a subclass of the ooDialog **ResDialog** class. The difference between **ResDialog** and **RcDialog** (aside from the resource file) is mainly in the handling of the menubar. Console outputs from dialog creation for **CustomerView** (an **RcDialog** subclass) and ProductView (a **ResDialog** subclass) are as follows:

```
CustomerView                            ProductView

StartCustomerView Routine-01: Start.     .ProductView-newInstance-01: Start.
CustomerView-init-01.                    ProductView-init-01.
```

---

[1] http://msdn.microsoft.com/en-us/windows/desktop/bb980924

```
CustomerView-createMenuBar-01.
StartCustomerView Routine-02: dlg~activate.   .ProductView-newInstance-02: dlg~Activate.
CustomerView-activate-01.                     ProductView-activate-01.
CustomerView-initDialog-01.                   ProductView-initDialog-01
```

There are two visible differences. First, as discussed above, instead of creating the **ProductView** instance in a routine, as was the case for CustomerView, the startup file invokes a class method - **newInstance** - which does much the same as CustomerView's routine.

The second difference is the absence of a **createMenuBar** method. Now this method was not strictly necessary in **CustomerView** - the menu could have been created in the **init** or the **initDialog** methods. (See *Appendix C, Dialog Creation Methods* for a comparison of dialog startup methods in an **RcDialog**, a **ResDialog** and a **UserDialog**.)

A third and less visible difference is that when a dialog has multiple resources such as bitmaps and/or icons, the number of files required for an RcDialog class can result in a minor file management challenge in the runtime environment. A ResDialog class, on the other hand, needs only two files: the *.dll and the *.h.

5.3. Dialog Controls

There are five features of **ProductView**'s controls that have not yet been introduced in this Guide. They are: radio buttons, a numeric-only edit field, menu accelerators, an image control (in the "About" dialog), and providing the dialog with minimize and maximize buttons (not really controls, but useful to discuss here).

5.3.1. Radiobuttons

For Radio Buttons to operate automatically - i.e. when an "off" button is clicked the previously "on" button goes off - they must be within a Group Box. This is defined in the *.rc file first as a **GROUPBOX** control with the style **WS_GROUP**. After this is defined, the radio buttons (which must have the **AUTORADIOBUTTON** style) are placed in the groupbox. However, the containment is done through the order of controls in the *.rc file. To achieve this using ResEdit, first drag a Group Box control onto the dialog, and set the "Group" property to "True". Then drag the radio buttons from the controls palette into the group box. Finally, and importantly, set the "Auto" behavior of each radio button to "True" (this sets its style in the *rc file to **AUTORADIOBUTTON** rather than just **RADIOBUTTON**). For a single group, it is not necessary to set the "Group" property to "True". However, if there are two or more independent group boxes, then it *is* required in order to differentiate between the groups.

When initially displayed, no radiobuttons are "on". In the Product View, radiobuttons show whether the size of the product is small, medium or large. Since size is an attribute of the particular product being displayed (i.e. it's a field in the data that was supposedly read from some database), the correct radiobutton must be turned on. This is done in the **showData** method.

5.3.2. The Numeric Edit Control

If you haven't tried entering an invalid number into the List Price or UOM fields of Product View, then try it. You'll find that keying a non-digit (including "-" or "+"), or keying more that two decimal digits in the Price field, or trying to key any decimals in the UOM field, will all result in a warning balloon being displayed. This behavior is provided by a mixin class called **NumberOnlyEditEx.cls**, available from the ooDialog "Samples" folder and copied into this User Guide's **Exercise05\Support** folder for convenience. **NumberOnlyEditEx** illustrates how a control can be extended through ooRexx's mixin capability. The mixin is applied when **NumberOnlyEditEx** is "::required" - its first executable

statement being: **.Edit~inherit(.NumberOnlyEditEx, .EditControl)**, with "Edit" being the name of ooDialog's Edit Control class.

Briefly, numeric-only edit controls are set up as follows (full details are in the comments at the front of the **NumberOnlyEditEx.cls** file):

1. Specify **::requires "Support\NumberOnlyEditEx.cls"** at the top of the dialog class file.

2. Initialize the edit control (the one that's to be restricted to numeric-only entry) in the **initDialog** method by invoking the mixin's **initDecimalOnly** method on the control instance. In **ProductView** this is done for the product price control by this statement:

```
prodControls[ecProdPrice]~initDecimalOnly(2,.false)
```

   The first parameter defines the allowable number of decimal places, the second whether or not a sign is allowed. As in **CustomerView** the controls are grouped in the directory object **prodControls** for ease of "exposing" them across methods; also, edit control instances have the prefix "ec" in conformance with the *Naming Conventions* mentioned in Chapter 4.

3. For each decimal-only edit control, a character event must be connected to an event handler method in the dialog object (ooDialog's edit control sends an event to the dialog when each character is entered). In **ProductView**, this is done in the **initDialog** method as follows:

```
prodControls[ecProdPrice]~connectCharEvent(onChar)
```

4. Provide the event handler method. The event handler method **onChar** in **ProductView** is as follows:

```
::METHOD onChar UNGUARDED
  -- called for each character entered in the price or UOM fields.
  forward to (arg(6))
```

   The sixth argument to the event handler is the control object where the character event occurred, and the event must be forwarded to that object - that is, to the eventful edit control. The event is then handled by the mixin class, where the numeric-only editing is done.

5.3.3. Menu Accelerators

Open the Product View dialog, and then press the Alt key on the keyboard, followed by the down-arrow key. The **Actions** menu is first highlighted and then opened. The top menu item is **Update Product** - with an underscore beneath the "U". Pressing the "U" key will then initiate the Update Product behavior. The underlined letter is known as an "accelerator" key. It is produced by placing an ampersand (&) immediately before the letter that's to be the accelerator key in the *.rc file. In **ProductView.rc**, you'll see the **Update** menu item defined as **MENUITEM "&Update Product",IDM_PROD_UPDATE**.

Interestingly, if you mouse-click on the **Actions** menu to open it, the "U" is not underscored - although pressing the "U" key still initiates the update action. This is standard Windows behavior, and ooDialog does not change it (although some third-party Windows apps such as Adobe's Reader do preserve the underscore when a menu is mouse-opened).

5.3.4. The "About" Dialog

Product View has a "Help" menu with one entry: **About...**. Clicking this menu item surfaces a simple "about" dialog, containing an image of a well-wrapped product. Double-clicking the image results in a message box acknowledging the action. This section discusses firstly how the image is created, and second making the image "active". The code for the About dialog is in the class **AboutDialog** towards the end of the **ProductView.rex** file.

5.3.4.1. Creating the Image

An image is created by placing a bitmap (a file of type "*.bmp") into a "Picture Control". The bitmap and Picture control are both defined in the *.rc file, but placing the image into the picture control is done in code. The following sections provide more detail.

5.3.4.1.1. Defining the Image

Assuming the bitmap image is already created as a bitmap file (a *.bmp file), then, using ResEdit, the *.rc file is created as follows (assuming you've already created a ResEdit project):

1. Select **File --> Add a resource... --> Bitmap**. Two options are presented: "create from an existing file", or "create a new resource". Click the former, which results in a File Open dialog.

2. Select the bitmap file and click **Open**. This produces a **Path designation** messagebox with two options: **Absolute path** or **Relative path**. It is usually best to choose "relative path". On clicking **OK** ...

3. ...a bitmap resource (named **IDB_BITMAP1** or some such) is added to the project and the bitmap image is displayed. If you want to change the name, then right-click on the bitmap in the Resources pane and select **Rename**. Note that the bitmap file is shown in the bitmap resource's "Path" property

4. Finally, drag a Picture Control from the controls palette and place it in the dialog. Then change the Picture Control's **Notify**, **Type**, and **RealSizeControl** attributes to **True**, **Bitmap** and **True** respectively. These attributes define the "styles" of the control to (respectively) issuing a mouse event, allowing a file of type "bmp" to be displayed, and fitting the bitmap to the space available.

   It's worth mentioning that the Picture Control is one of four types of Static Control - text, graphics, image, and owner-drawn. In the *.rc file, these are defined by their "styles". A "style" is an essential and basic concept in Windows. While many styles are shown in the ooDialog Reference, the full authoritative list of styles is found in the *Microsoft Control Library*[2]. Look up the Static Control, and you'll find around thirty different styles.

5.3.4.1.2. Mapping an Image to a Picture Control

As mentioned above, the image (of a parcel) displayed in the About dialog is referenced in the *.rc file, and hence is referenced in the *.dll file. What now needs to happen is to associate the image with the static control that will contain it. This is done in the dialog's **initDialog** method as follows:

```
        resImage = .ResourceImage~new( "", self)
        image = resImage~getImage(IDB_PROD_ICON)
        stImage = self~newStatic(IDC_PRODABT_ICON_PLACE)~setImage(image)
```

---

[2] http://msdn.microsoft.com/en-us/library/bb773169%28v=VS.85%29.aspx

The first statement creates an instance of ooDialog's **ResourceImage** class. The second statement uses the ResourceImages's **getImage** method to return an instance of the **Image** class. The last statement creates a static control proxy and sets the image in it. (And the following two statements in the method create a font and apply it to the static text in the dialog).

5.3.4.2. Making the Image "Active"

Making the image respond to mouse clicks is merely a matter of defining the image-static control in the *.rc file as having the style **SS_NOTIFY** which, using ResEdit (and as mentioned in *Section 5.3.4.1.1, "Defining the Image"*), merely requires the "Notify" attribute to be set to "True". When the image is double-clicked, the dialog is sent an event. This event is connected to the **showMsgBox** event-handler method by the statement **self~connectStaticNotify(... showMsgBox)**. The **showMsgBox** method then displays a messagebox.

You may notice the **leaving** method. This is invoked automatically when the underlying Windows dialog is being closed. Its purpose is to allow for clean-up. In the case of the About dialog, the two resources used (an image and a font) are released. This is not really necessary in this simple application, but is a good habit to get into.

5.3.5. Minimize and Maximize Buttons

The Product View dialog has a minimize button and a disabled maximize button, both at the top right of the dialog to the right of the title bar and to the left of the close button. The minimize button is defined in the *.rc file. When only one button is specified, Windows automatically includes both buttons, but with the non-specified button being disabled.

If you look in **ProductView.rc** file, you will see the following styles defined for the dialog (slightly re-formatted for readability):

```
STYLE DS_3DLOOK | DS_CENTER | DS_SHELLFONT  | WS_CAPTION | WS_VISIBLE |
      WS_GROUP  | WS_POPUP  | WS_THICKFRAME | WS_SYSMENU
```

So where is the **WS_MINIMIZEBOX** style (as mentioned in the ooDialog Reference)? Well, one of the curiosities of Windows, probably historical, is that both the **WS_MINIMIZEBOX** style and the **WS_GROUP** style (defined in the Windows **WinUser.h** file) map to the same numeric value (0x00020000L). Also, the **WS_MAXIMIZEBOX** and **WS_TABSTOP** styles both map the same number (0x00010000L). Clearly ResEdit likes the older form, and includes a **WS_GROUP** when you specify a Minimize Box. Luckily, ooDialog accepts either.

You might try adding either or both of the styles **WS_MINIMIZEBOX** and **WS_MAXIMIZEBOX** to **CustomerView.rc** in Exercise04. The line to change is the one starting: **STYLE DS_3DLOOK**. Then run the exercise,and check the result. You should see the Customer dialog as before except for the minimize and/or maximize buttons.

5.4. Code Structure

Although broadly similar to the code structure in Exercise04, Exercise05 introduces several new structural concepts (at least new in this Guide). These are: the use of "data types", differentiation between view data and application data, more than one dialog in a file, externalized strings, and a more complex dialog design.

5.4.1. Data Types

Most non-trivial software systems consist of a number of components. Each of these could in principle be written in a different programming language (assuming of course that all the languages share are supported by common invocation mechanisms). Within each component there are typically some number of classes, and these interact privately. Because interaction between components tends to be "data-heavy", it is usual to define specific "data-only" classes, so that everyone can be sure of using the same data structures. Examples are: a Customer data class, an Address data class, and a SalesOrder data class. Each such class is often referred to as a "type" (a term that in some quarters is a synonym for "class").

Our sample Sales Order application conforms with this idea, and so a number of "data types" will be introduced. In Exercise05, the Product Data type (the class **ProductDT** at the end of the **ProductModelData.rex** file) specifies the attributes or fields required to fully define product data. This class needs no methods, since the data elements of **ProductDT** are defined as ooRexx attributes. Indeed, the single method in this class is merely a convenience method that lists the contents of a **ProductDT** instance on the console. This method is used (for illustration purposes only) in **ProductView** at the end of the **saveChanges** method.

5.4.2. View Data vs Application data

There is often a difference between data that the user sees or enters on a dialog and the data that flows between components in data types (just as there are differences between data in a normalized database and data as used by application code). For example, on the Product View dialog, a price is shown with two decimal digits after a decimal point. Price in the **ProductDT** data type, on the other hand, has no decimal places - it's expressed in units of 1/100s of the currency unit (that is, in cents if the currency unit is the Dollar). Thus the price data type must be transformed both when displayed to the user and when read in by the program.

The principle for where to do the transformation is simple: do it as close to the screen as possible (just as, at the other end, transformation to database formats are done as close to the DB programming interface (e.g. SQL) as possible, meaning that most of the application code across the system can use the same normative data formats.) Following this principle, the first thing the event handler method **saveChanges** does is to invoke the dialog's method **xformView2App** (transform view to app format). For example, a new price may have been entered with one or zero decimal digits, and so needs to be converted correctly to a whole number of cents. Conversely, reformatting for display is handled in the **showData** method. The end result is that all other methods in **ProductView** can assume that data is in the format defined by the data type. And this simplifies things a great deal - especially given that one never wants to confuse 10000 for ten thousand dollars when it's really 100 dollars!

5.4.3. Multiple Dialogs per File

The file **ProductView.rex** contains two dialogs - the main **ProductView** class and the **AboutDialog** class. Note that the resources for both dialogs are defined in the same ResEdit project, and hence in the same *.rc file. Thus both are compiled into the **ProductView.dll** file. This means that the single statement **.Application~setDefaults("O","ProductView.h",.false)** in the **newInstance** method of the **ProductView** class applies to the **About** dialog as well.

You may notice that the "About" dialog launched in **ProductView**'s **about** method is modal. That is, the Product View window cannot be accessed while the About window is open. This is because "About" is launched using the **execute** method. Making an "about box" modal seems quite reasonable. But in the next chapter, alternatives to **execute** will be used in order to launch non-modal dialogs.

5.4.4. Externalized Strings

It is generally deemed to be good practice to externalize any strings that are visible to the user. This enables someone who needs to translate the application for use by speakers of a different language to do so without touching any executable code. Providing such a facility is often called "internationalization" or "I18N" for short. While this Guide does not pretend to have addressed all I18N requirements, at least it indicates an understanding of the need. Thus the human-readable strings that appear in the **HRS** class ("HRS" = "Human-Readable Strings") at the end of the **ProductView.rex** file are used for messages in messageboxes. Other strings such as the static text on the dialogs, and text in the About dialog are hard-coded either in code or in the *.rc file (and hence in the *.dll file). While not good practice, this is done in the Exercises for code readability reasons - the alternative being to add more code to the dialogs.

5.5. Designing a Dialog

A program does only what its programmer specifies. But a user could do anything. This is why designing dialogs is often quite complicated. What could the user do? What must the dialog do? These are two questions that sometimes seem to intertwine into an irresolvable mess. In **ProductView**, the most complex piece of behavior is when the user chooses the **update** menu item. A useful way to plot the possible interactions is to use a UML Activity Diagram, with user actions on one side and the corresponding program actions on the other. The following figure shows one such diagram.
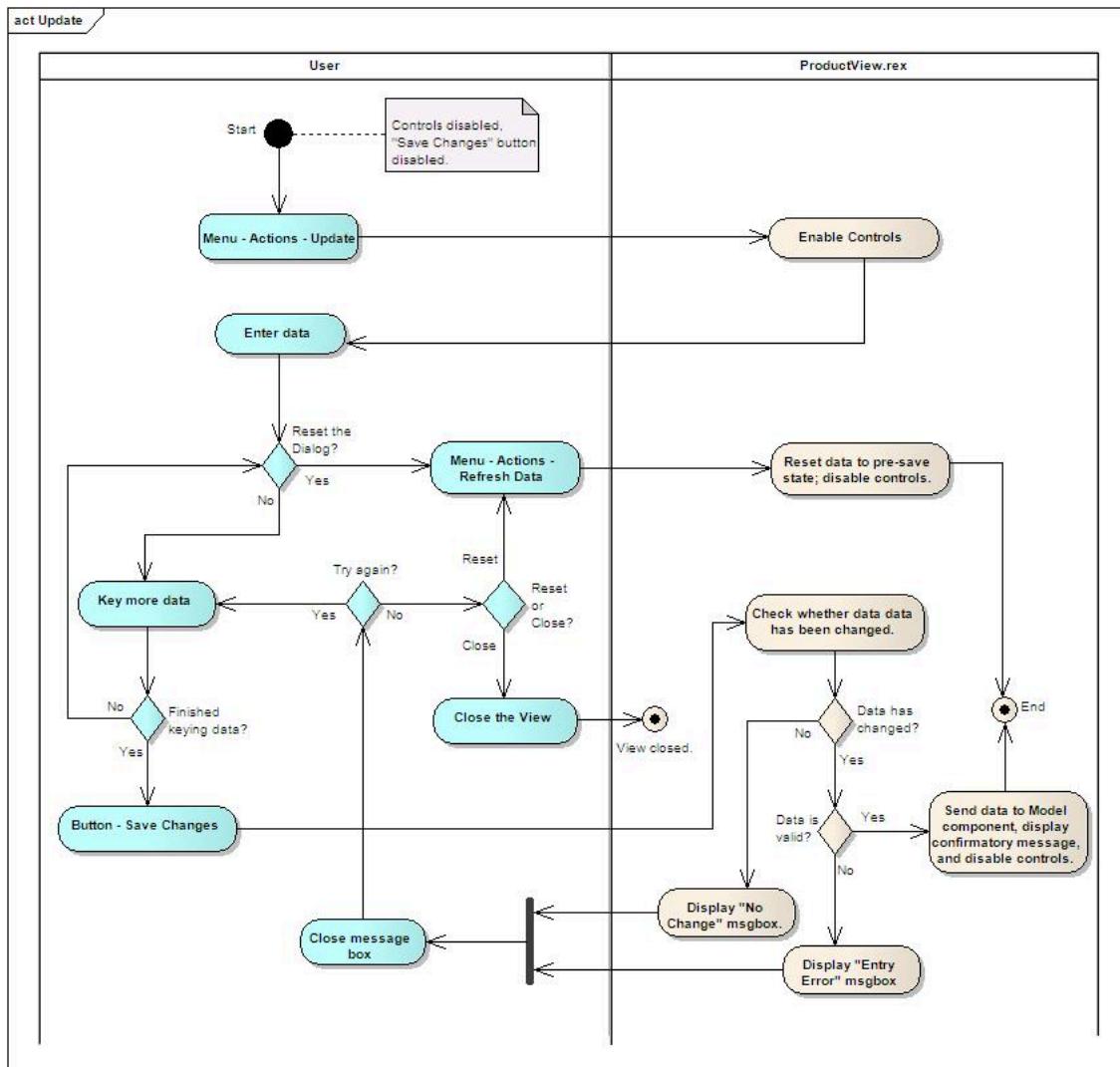
Figure 5.1. ProductView Behavior Diagram

The main thing this diagram illustrates is how the user can go in circles (should he/she wish to) without affecting what the code needs to do. This is helped a great deal by providing a "refresh" function, so that if the user gets mixed up in entering data, s/he can go back to the beginning and start again.

5.6. Controlling Dialog Cancel

Windows provides three ways for the user to cancel a dialog: by pressing the **Esc** key, by clicking on the "close" icon at the extreme top right of the dialog, or by clicking the **close** action on the system menu (click the icon at the extreme top left of the window). All three of these actions result in a "cancel" message being sent to the dialog, and the default superclass behavior is silently to close the dialog. In general, since these default actions are standard for all Windows dialogs, they should not be over-ridden except perhaps to display an "are you sure?" message if, for example, the user is half-way through some unit of work.

The Product View code provides a simple illustration of this in that, depending on the state of the dialog, a modal "are you sure you want to exit" message is displayed. Product View can be said to have three states, as illustrated by the UML state diagram below. In the diagram, ovals are states, and the lines between them are state transitions. A solid circle is the start, and a smaller solid circle with a ring around it is the end.
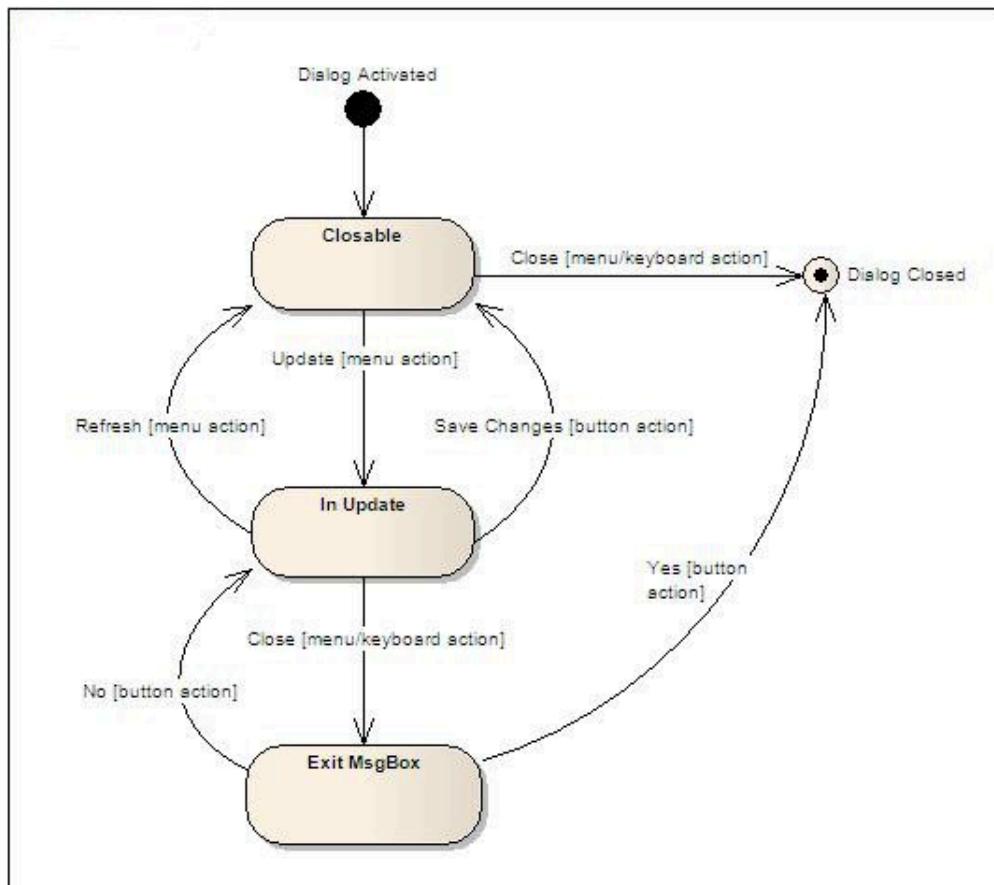
Figure 5.2. ProductView State Diagram

The three states are as follows:

1. The first state is called "closable" - that is, not in the process of being updated. This state is in being when the value of the attribute **dialogState** (defined in the **::attribute** directive immediately after the **::class** directive) is "closable".

2. The second state is called "in update" and is in being when the value of the **dialogState** attribute is "inUpdate". If the user selects any of the close actions in this state, then the third state is entered. The "InUpdate" state is terminated by the user menu selecting **Refresh Data** or pressing the **Save Changes** button - either of which cause the dialog to return to the "Closable" state.

3. The third state is the modal "exit messagebox" state which, depending on the user's choice, either closes the dialog or returns to the "In Update" state. If the user selects "close", then the dialog closes immediately.

An Application Workplace

This chapter introduces the Order Management application, which is designed as a "workplace" for a user handling sales orders. As such, it provides access to the required components - customers, products, orders, and order forms. A common approach for a "workplace" dialog is to provide an icon for each component that the user may wish to use. In the **Exercise06** folder, run **startup.rex**. The **Sales Order Management** dialog opens, and consists mainly of a List View containing four icons. Move the icons around; double-click them and if a Customer, Product, or Order List appears then double-click a list item; re-size the "Order Management" window; check out the menu items and the pushbuttons. As you see, while much of the application function is absent, the data is hard-coded, and there is some redundancy (e.g. every time you double-click on an item in a list view a new Model and Data object is created), the essential parts of the Order Management application mentioned in *Chapter 4, Using Resource Dialogs* are visible. This chapter addresses the following topics in the context of the Order Management application:

- *Program Structure*

- *Popups and Parents*

- *Icons and Lists*

- *Re-sizing Dialogs*

- *Creating Icons*

- *Utility Dialogs*

6.1. Program Structure

6.1.1. Overview

In Exercise 6, each business component has its own folder: Customer, Order, OrderMgr, and Product. Customer and Product are more or less identical to the same components introduced in Exercises 4 and 5 except for the addition of a list view (**CustomerListView.rex** and **ProductListView.rex)**. Placing each business component into a separate folder helps promote high cohesion and low coupling in the software, since the internals of each business component should be opaque to other business components. Thus another application (e.g. Customer Relationship Management) could well be able to make use of the Customer business component without change. The Order Manager (**OrderMgr**) business component is unlikely to be re-used in other applications as it is a kind of "container" business component that "choreographs" the other business components. To the user, creating a new sales order consists of "choreographing" the various business aspects required - creating an Order Form (used to assemble the customer order), searching for and selecting a specific Customer, searching for and selecting one or more Products, recording the quantities ordered, and producing a Sales Order that is the "contract" between supplier and customer. Of course, the **OrderMgr** component could be used by "higher-level" components such as business processes or workflows (for example a fulfilment process). In systems organized according to these principles, invocation of components takes the form of a directed acyclic graph.

Within each business component are a number of components. For example, the Customer business component contains a Customer View and a CustomerList View, as well as CustomerModel and CustomerData components. Also in the folder are the other files (.h, .rc, .bmp and .ico) required. As in Exercise 04, **CustomerView** gets its data from **CustomerModel**, which in turn gets its data from **CustomerData**. However, **CustomerListView** is rather skeletal, and its data is hard-coded. In a later exercise, this will be modified, and will get its data directly from **CustomerData** (there being

no need, in this simple example, for a ListModel component). Note that aside from the list views, the Customer and Product business components are essentially the same as in Exercises 4 and 5 respectively.

The Order Manager business component is implemented by the view class **OrderMgrView**. This contains the code for handling a re-sizable dialog (see *Re-sizing Dialogs*). **OrderMgrView** contains the code specific to the Order Management application. The only reason for splitting the code like this is that it seems to fall happily into these two parts. This reduces the amount of code in any one class or file, and so (arguably) makes for better readability.

Note that the "data" of the Order Manager business component is the set of icons and their associated data. Examples of possible additional components could be a "commodities" component which shows the commodities required to produce a given product; or a credit-check component that links to an external credit-check agency.

The Order business component is extremely skeletal, but will be developed further in a later exercise.

Finally, it's worth noting that the folder structure, while useful for development, is not necessarily the most appropriate structure for a deployed application. A more appropriate deployment structure will be described in a later chapter.

6.1.2. Some Implications

Choosing this folder structure for the development of the application has certain design implications worth mentioning. These are: file paths, the use of **.Application**, and the handling of externalized human-readable strings.

6.1.2.1. File Paths

When an ooRexx program is run, the current (or "home") directory is that from which the program is started. That is, if a program is started on a command prompt like this: **c:\aaa\bbb>myprog.rex**, then the current directory will be **c:\aaa\bbb**. However, if the program is started like this: **c:\aaa>bbb\myprog.rex**, then the current directory will be **c:\aaa**. In Exercise 6, all programs are started from the **Exercise06** folder. Thus any relative paths must be relative to the **Exercise06** folder. However, using ResEdit with relative paths, the path for a resource such as a dialog icon is relative to the folder in which the *.rc file is created. For example, if **CustomerView.rc** is created in the **Customer** folder, then the icon will be specified in **CustomerView.rc** with the path **".\\bmp\\Customer.ico"**. So the path for the icon resource in the *.rc file will be wrong, and the dialog icon will not be shown.

The solution is either to edit the *.rc file and change the icon resource's path, or (better) create the *.rc file in the Exercise06 folder and then move it into the **Customer** folder.

Considering the implication of paths also applies to other parts of the code, such as header files specified in the **.Application~addToConstDir(...)** statement and dialog creation statements such as **dlg = .CustomerView~new(...)**.

In summary, all paths (if not absolute) must be relative to the folder from which the program is started. This is why, when running a stand-alone test such as **startupCustomer.rex**, the program must be started from the **Exercise06** folder, since the design decision was taken to make all paths relative to **Exercise06** folder.

This discussion on paths prompts two thoughts (at least). First, is there not a way to have some support code manage paths, so that each component asks this support code for the path it should

use? While this may be feasible, it's not specifically an ooDialog questions, and so is not pursued here. Second, using the **ResDialog** class instead of **RcDialog** reduces the problem of paths, since resources such as icons and bitmaps are placed in the *.dll file.

6.1.2.2. .Application Usage

The **startup.rex** file applies application-wide defaults through the statement **.Application~setDefaults("O", , .false)** . However, the header file for each view class is included at the beginning of its file. For example, **.Application~addToConstDir("Customer \CustomerView.h")** is the first executable statement in **CustomerView.rex**. For a shipped application that includes multiple classes, it would be much better to provide all the **~addToConstDir()** statements in the startup file after, say, the **~setDefaults()** statement. However, because at this stage the application is still in a pre-deployment state, and each component needs to be able to be unit-tested (see *Appendix B, Testing Popups in Stand-Alone Mode*), it was deemed better to include the *~addToConstDir()* statements in each view file. An alternative was to duplicate them in the unit-test startup programs, but code duplication is generally not the best strategy.

6.1.2.3. Externalized Strings

All components that display information to the user have the displayed strings separated from code either in a class whose name is prefaced "HRS" for "Human-Readable Strings" or in a *.rc file. Each such class name has a suffix - e.g. "HRScv" for the **CustomerView** class. The reason for the suffix is to distinguish the various HRS classes if the various files were later to be placed into a single file for application deployment purposes.

Human-readable strings in *.rc classes are a problem when internationalization is a requirement. "Internationalization" (often referred to as I18N - there are 20 letters in the word) is the term given to providing for translation of human-readable text into other languages. An immediate solution is to display the translated strings from within the program rather than from the *.rc file. The **initDialog** method is a good place to do this. Try inserting the following in **CustomerView.rex**, say just before the statement **self~getData** in the **initDialog** method:

```
custNameLabel = self~newStatic("IDC_CUST_LBL_CUSTNAME")
custNameLabel~setText("Namen:")
```

The Customer View will be displayed with "Namen" (German) instead of "Name" (English).

Of course, this text should come from an HRS class or better from a proper I18N resource file. Such files would be produced using special I18N tools. These tools take account of the many considerations and gotchas of internationalization. For example, in our trivial attempt to change the label from English "Name" to German "Namen", we've lost the colon at the end. This is because the horizontal space given to this particular static text in **CustomerView.rc** is not big enough. Some authorities suggest that 150% of the space required for English is needed to allow for proper translations to other languages. And this is only one of the lesser considerations in the task of internationalization. The following quote from the Wikipedia entry illustrates something of the full complexity of I18N: "It should be noted that 'internationalized' does not necessarily mean that a system can be used absolutely anywhere, since simultaneous support for all possible locales is both practically almost impossible and commercially very hard to justify. In many cases an internationalized system includes full support only for the most spoken languages, plus any others of particular relevance to the application."

6.1.3. Application Function and Naming

One of the first things to notice about Exercise 6 is that there is very little application function. For example, data is all hard-coded, and all Customers have identical data, as do all Products and

all Orders (this will be fixed in the next exercise). The second thing is that while there's an **Order** component to display and change existing sales orders, there's a separate **OrderForm** component for creating new sales orders. But there is no way to create new Customers or Products. This will not be fixed in the next exercise, mainly because providing this function would not exercise any new ooDialog features. However, in the next exercise data will be able to be read from a data file.

The reason for having a different dialog for creating as opposed to viewing and updating is that in real applications, creating a Customer, or Product, or Order generally requires a more complex process than simply updating. For example, creating a Customer often cannot be done without a credit check and establishing the customer's bank details in the accounting system. In our Order Management application, only the **Order** component has a separate Order Creation dialog, in order to exercise, in a later chapter, more ooDialog capabilities. So, for present purposes, we assume that Customer and Product creation takes place outside of the sample Order Management application.

Although perhaps not immediately apparent, a specific naming convention has been used. This convention is useful to differentiate between the various parts of the application. Thus "X Management" is the name given to the application as a whole (in our case "X" is "Sales Order"). Generally, an application constructed from a number of relatively autonomous components has one or more "coordinator" or "process" components that arrange for the "choreographing" of other components. These are often called "Managers" - hence the "Order Manager" component that provides the framework for the application. Finally, entities such as Customer are given the entity name - "Customer", "Product", "OrderList", etc. - followed by the suffix "View", "Model", or "Data" as discussed in *Chapter 3, Re-Structuring the Code*.

Finally, starting an application that can make changes to a business is generally guarded by some form of security. When starting the sample application, this is trivially represented by a password dialog, which is visible if you start the application (in the **Exercise06** folder) with the command **startup enterPW** instead of just **startup**. Yes, this is the wrong way round, but its purpose is to illustrate the code required for a password prompt. It is very simple, and uses one of the many and useful ooDialog built-in dialogs - **PasswordBox(...)** - as follows:

```
parse arg pwOption
if pwOption = "enterPW" then do
  pwd = PasswordBox("Please enter your password","Sign In")
  if pwd \= "Password" then exit
end
```

6.2. Popups and Parents

This section addresses how the various dialogs in the Order Management application are launched. First, the way in which dialogs are started is addressed. Second, the use of the **interpret** statement in displaying dialogs by double-clicking an icon in the "Order Management" window is discussed.

6.2.1. Starting a Popup Dialog

In previous chapters, dialogs have been started using the statement **self~execute(...)**. The **~execute** method makes the dialogs "modal", that is, access to other dialogs is blocked until the dialog is closed. A good example of a modal dialog is the Help-About dialog in Exercise 5. While this is open, the Product View dialog is blocked.

The dialogs in Exercise 6 are not modal; they are "amodal" or "modeless". Any of them can be accessed by the user at any time. A modeless dialog is created by using the **~popup** or **~popupAsChild** methods in place of **~execute**. The difference between the two is as follows.

- **~popup()** - If dialog A pops up dialog B, then B exists independently of A. Either can be accessed by the user at any time. Either can be closed without affecting the other. The application ends only when both are closed.

- **~popupAsChild(parentDlg)** - If dialog A pops up dialog B as its child, then B's existence depends on A's. If A is closed, so does B. However, as with **~popup** either can be accessed by the user at any time. Note that the only required argument for **~popupAsChild(parentDlg)** is the parent dialog.

For the Order Management application, **~popupAsChild** is used. Thus while **OrderMgrView** is started with **~execute**, all other dialogs in the Order Management application are started with **~popupAsChild(rootDlg)** where the "root" (or parent) dialog is always **OrderMgrView**. Thus all dialogs are modeless and independent of each other, except that when **OrderMgrView** is closed, everything else closes and the application ends. (Note that **ProductView**'s "About" dialog is still modal: it blocks access to the specific instance of **ProductView** from which it is launched; other instances of **ProductView** are unaffected, as are other dialogs.)

So, for a dialog to be "popped up as child", there has to be a parent dialog that was surfaced with either *~popup* or *~execute*. This presents a problem for stand-alone testing. The solution adopted in this exercise is illustrated by the following code fragment, taken from **CustomerListView**'s **activate** method (which is called from its **newInstance** class method):

```
::METHOD activate UNGUARDED
  expose rootDlg
  use arg rootDlg
  if rootDlg = "SA" then do   -- If standalone operation required
    rootDlg = self     -- To pass on to children
    self~execute("SHOWTOP","IDI_CUSTLIST_DLGICON")
  end
  else self~popupAsChild(rootDlg, "SHOWTOP", ,"IDI_CUSTLIST_DLGICON")
  return
```

This code illustrates the two ways of starting a dialog. For stand-alone testing (see *Appendix B, Testing Popups in Stand-Alone Mode*), the dialog is started using **self~execute()**. In normal operation, however, it is started by **self~popupAsChild(...)**. Notice that the first parameter of **~popupAsChild(rootDlg, ...)** is the **OrderMgrView** dialog, which is passed to the **newInstance** class method and thence as the parameter **rootDlg** to the **activate** method. Thus **CustomerListView** is both a child of **OrderMgrView** and parent of **CustomerView**. Later in **CustomerListView**, a Customer is displayed by the user double-clicking on an item in the List View. The event handler method (**showCustomer**) that surfaces the Customer is as follows:

```
::METHOD showCustomer UNGUARDED
  expose lvCustomers rootDlg
  item = lvCustomers~selected
  if item = -1 then do  -- if no item selected.
    ret = MessageDialog(.HRSclv~nilSelected, self~hwnd, title, 'WARNING')
    return
  end
  info=.Directory~new
  if lvCustomers~getItemInfo(item, info) then do
    .local~my.idCustomerData  = .CustomerData~new    -- create Customer Data instance
    .local~my.idCustomerModel = .CustomerModel~new   -- create Customer Model instance
    .local~my.idCustomerData~activate
    .local~my.idCustomerModel~activate
    .CustomerView~newInstance(rootDlg,"CU003")
    self~disableControl("IDC_CUSTLIST_SHOWCUST")
  end
  else do
```

```
        say "CustomerListView-showCustomer-04: ~getItemInfo returned .false."
     end
```

The list of customers is shown in a ListView control (see *Icons and Lists* below). The **showCustomer** method is invoked when the user double-clicks on an item in the list. This item is identified by the statement **item = lvCustomers~selected**, the proxy object for the list control being **lvCustomers** (an item is automatically selected when it is double-clicked). If no item is selected, an error message is displayed, and the method returns. The data in the selected row is then placed in a directory (with an error check in case **~getItemInfo** returns **.false**). The next statements (**.local~my...**) create instances of the **CustomerModel** and **CustomerData** classes. Then an instance of **CustomerView** is created by the statement **.CustomerView~newInstance(rootDlg,"CU003")**. The second parameter is the Customer Number, which is ignored in Exercise 6 (but which will be used in a later exercise). Finally, the **Show Customer** pushbutton is disabled.

The approach to establishing the model and data objects shown here is not ideal. Indeed, the above code merely satisfies the requirement for a **CustomerView** object to have access to a **CustomerModel** instance which in turn needs access to an instance of **CustomerData**. And, in this exercise, the data is all hard-coded. The next exercise will illustrate a much better way of doing this, with data being read from a disk file (a notional "data base").

6.2.2. Offsetting Dialogs

When creating a resource file for a dialog, it is unusual to define the position of the dialog on the screen. Instead, the option to center the dialog in the screen is often used. This is the option applied in Exercise 6. However, when a number of different dialogs are all surfaced in the same place they tend to overlap each other, so making things difficult for the user who has to continually move dialogs away from the center. A better approach is to offset newly-surfaced dialogs from existing ones such that the new dialog pops up in the best place from a user point of view. This is possible with ooDialog, but is not simple.

However, ooDialog also provides a half-way house, where simple code produces a useful result. This simpler code is discussed in *Section B.2, "Visual Offsetting"* in *Appendix B, Testing Popups in Stand-Alone Mode*. The following code illustrates the key functions:

```
    -- In 'parent' dialog:
    ::METHOD getPopupPos
      popupPos = self~getRealPos
      popupPos~incr(100,100)
      return popupPos

    -- In 'child' dialog:
    ::METHOD offset
      use arg popupPos
      self~moveTo(popupPos, 'SHOWWINDOW')
      self~ensureVisible()
```

The "parent" dialog finds its own position on the screen with **parentPos=self~getRealPos>** (where **parentPos** is an instance of the **Point** class). It then increments the point's **x** and **y** coordinates using the point's **incr** method. The result is the child dialog's desired position. When the parent dialog pops up a "child" dialog, it passes this desired position to the child dialog. From the child dialog's **initDialog** method, either in-line or with a method call, the instruction **self~moveTo(popupPos,...)** moves the child dialog to the desired position. Finally, the instruction **self~ensureVisible()** ensures that the child dialog is wholly on the screen and not partly invisible.

6.2.3. Use of 'Interpret'

When an icon in the "Order Management" dialog is double-clicked, a child dialog is surfaced. This is handled by two methods in the **OrderMgrView** class. First, the event-handling method **onDoubleClick** catches the double-click, works out which icon (or "record" - see *Icons and Lists* below) was double-clicked, and then calls the **showModel** method. This method uses an **interpret** instruction to launch a view of the component represented by chosen icon, as follows:

```
use arg record
className = record~ID
viewClassName = className||"View"
interpret "."||viewClassName||"~newInstance(self)"
```

Thus in principle icons for additional components can be added without changing the code. An alternative to using **interpret** would be to use the ooRexx **Message** class. Then again, an arguably better approach could have been to hold the class object in the record, and to invoke **newInstance** directly on the class object. However, in the next exercise, the mechanics of invoking the various components (given a class name and an "instance name" such as a customer number) will be moved to support classes.

Finally, a separate file - **RequiresList.rex** - contains the set of **::requires** statements for the components that might be surfaced. This is why the first executable statement in the file **OrderMgrView.rex** is **call "OrderMgr\RequiresList.rex"**. While these statements could have been included in **OrderMgrView.rex**, they were separated as they can be thought of as "configuration", and it's arguably better to keep configuration separate from code.

6.3. Icons and Lists

A **ListView** should not be confused with a **ListBox**. A ListView is a souped-up ListBox with lots of additional features. In particular:

- An item in a ListView can be a complex structure or "record" containing multiple fields. One of these fields is termed the "label" of the item.

- ListView items can be displayed in four different styles (or modes):
  - Icon view - each item appears as a full-sized icon with a label below it. Items can be dragged around the ListView.

  - Small-icon view - each item appears as a small icon with a label to its right. Items can be dragged around the ListView.

  - List view - each item appears as a label with an optional small icon to its left.

  - Report view - each item appears as a row in a table with an optional small icon to its left.

  The four different modes are well illustrated by the sample program **oodListViews.rex** located in the **ooRexx\samples\oodialog** folder.

In the Order Management application, a ListView control in the "Icon" style provides the main area of the **OrderMgrView** dialog where draggable icons represent the various components of the application. (The ListView control in the "Report" style is used to provide the tabular lists for the **CustomerListView**, **ProductListView**, and **OrderListView** dialogs.)

6.3.1. The Icon View

The Order Manager dialog is provided by two classes: **OrderMgrBaseView** and **OrderMgrView**. The former handles re-sizing, and to do this it needs to know about the ListView control. But the latter also needs to know about the ListView control. To provide for both requirements, the proxy for the ListView control is stored in **OrderMgrBaseView** as a private attribute named *lv*.

Five things are needed to produce an icon view: first, create (or obtain) some icons; second, specify the **ICON** style for the ListView control; third, create an ImageList from the icons (required by the ListView control); fourth, create a set of records (one record per icon) to be loaded into the ListView; and fifth, load the icons and records into the ListView.

1. **Produce the Icons**

   The large "icons" in the ListView are actually bitmaps. Icons and bitmaps have different formats, and different uses, and there are a number of differences between them. The bitmaps themselves are in the folders of the relevant business components, so the "icon" for the Customer List, for example, is **Exercise06\Customer\bmp\CustList.bmp** (the *.ico files are the dialog icons). See *Creating Icons* for further information.

2. **Specify the ICON Style**

   The icon style for a ListView control is specified either in the *.rc file as the **LVS_ICON** (in ResEdit, set the "View" property to "Icon"), or in a UserDialog, by creating the ListView control in the **initDialog** method using the **ICON** style - e.g.: **self~createListView(IDC_ORDMGMT_ICONS, ... "ICON")** where the first parameter is the ID for the ListView control.

3. **Create an ImageList**

   The ListView documentation provides several ways to load icons. Probably the easiest is to create an instance of the **ImageList** class which is then loaded into the ListView. In **OrderMgrView**, this is done in the **createIconList** method (invoked from the **init** method) as follows:

```
        ::METHOD createIconList PRIVATE
          expose iconList
          imgCustList  = .Image~getImage("customer\bmp\CustList.bmp")
          imgProdList  = .Image~getImage("product\res\ProdList.bmp")
          imgOrderList = .Image~getImage("order\bmp\OrderList.bmp")
          imgOrderForm = .Image~getImage("order\bmp\OrderForm.bmp")
          -- Boldly assume no errors in creating the Image List or in the ~getImage
statements.
          iconList = .ImageList~create(.Size~new(64, 64), .Image~toID(ILC_COLOR4), 4,
0)
          iconList~add(imgCustList)   -- item 0 in the list
          iconList~add(imgProdList)   -- item 1 in the list
          iconList~add(imgOrderList)  -- item 2 in the list
          iconList~add(imgOrderForm)  -- item 3 in the list
          imgCustList~release
          imgProdList~release
          imgOrderList~release
          imgOrderForm~release
          return
```

   For each icon, only two statements are required: create an Image from file, and then copy it to the ImageList (and a third, if you're a polite programmer, clean up afterwards by releasing the image).

4. **Create Records**

Records are typically created in the **init** method (or in a method invoked from there). In **OrderMgrView** the records are created in the **initRecords** method which is invoked from **init**. Each record has two fields: the class name of the dialog to be surfaced when a user double-clicks on an icon, and the text to appear beneath the icon. The design choice for these records is that each record is a directory, and each directory is stored in an array. The array index of a record is equivalent to the position of its icon in the ImageList (remembering that arrays are 1-based while ImageLists are 0-based). The code for creating the record array is as follows (showing only the Sales Orders item for brevity):

```
::METHOD initRecords PRIVATE
  expose records
  records = .array~new()
  ...
  rec = .directory~new
  rec~ID = "OrderList"          -- Class Name
  rec~name = "Sales Orders"     -- Text to display under the icon
  records[3] = rec
  ...
  return records
```

5. **Load the ImageList and the Records**

Loading icon images and records into the ListView is done in **OrderMgrView**'s **initDialog** method:

```
::METHOD initDialog
  expose records iconList
  self~initDialog:super
  self~lv~setImageList(iconList, .Image~toID(LVSIL_NORMAL))
  do i=1 to records~items
    self~lv~addRow(, i-1, records[i]~name)
  end
```

The icons in the ImageList are all applied to the ListView control in the single statement, **self~lv~setImageList(...)**. The second parameter of the **setImageList** method specifies the size of the icons by invoking the **toID** method of the Image class with the parameter **LVSIL_NORMAL**. This is the flag for the icon view as opposed to the list, report, or small icon views. The Image class is used to work with and manipulate images. The icons having been set, the records are then added using the ListView's **addRow** method. The first parameter is the index of the list item (if omitted, the record is added after the last). The second parameter is the index of the icon to be used with this record, and the last parameter is the label for the list item - the string "Customer List" in the case of the first item added.

6.3.2. The Report View

Three of the icons in the Sales Order Management dialog surface a list when double-clicked - the Customer List, Product List, and Order List. These three components are technically very similar - so that a "list superclass" could perhaps be useful. However, in Exercise 6 this is not done, and each list is quite separate. Nevertheless, their similarity means that discussing one list - the Customer List - effectively addresses all three.

A list view with the "Report View" style provides for a variable number of columns, each item appearing on a separate line with information arranged in columns. Each line may have a small icon at the left of each line. Note that the fields in a ListView must be defined in code, since a Windows resource file does not support the definition of columns within the list view.

The following code fragment from the **CustomerListView** class shows how the List View (without small icons) is defined:

```
::METHOD initDialog
  expose menuBar lvCustomers btnShowCustomer
  ...
  lvCustomers = self~newListView("IDC_CUSTLIST_LIST");
  lvCustomers~addExtendedStyle(GRIDLINES FULLROWSELECT)
  lvCustomers~insertColumnPX(0,"Number",60,"LEFT")
  lvCustomers~insertColumnPX(1,"Name",220,"LEFT")
  lvCustomers~insertColumnPX(2,"Zip",80,"LEFT")
  self~connectListViewEvent("IDC_CUSTLIST_LIST","CLICK",itemSelected)  -- Single click
  self~connectListViewEvent("IDC_CUSTLIST_LIST","ACTIVATE",openItem)   -- Double-click
  self~connectButtonEvent("IDC_CUSTLIST_SHOWCUST","CLICKED",showCustomer)
  self~loadList
```

First, a proxy for the ListView control, **lvCustomers**, is created. Then, in the second statement, the list view is formatted using "extended styles" (of which there are around twenty). Extended styles are defined by Microsoft, and can only be added after the underlying Windows control has been created - that is, (normally) in the **initDialog** method. In the above code, only two extended styles are applied: GRIDLINES and FULLROWSELECT. Both apply only to the Report View. The former draws gridlines around all items; the latter defines that, when a row is selected by the user, the whole row is highlighted rather than just the first column. Then there are three **~insertColumnPX** statements, each adding a column to the list view - "Number", "Name", and "Zip". Following these are two **~connectListViewEvent** statements that define event handler methods for single click and a double-click - **itemSelected** and **openItem**. The latter merely invokes the **showCustomer** method, as does the second-to-last statement **~connectButtonEvent** which defines the event handler method for the pushbutton.

The last statement in the above invokes the **loadList** method, which loads the list view with data, as follows:

```
::METHOD loadList
  expose lvCustomers
  lvCustomers~addRow( , ,"CU001", "ABC Inc.",    "TX 20152")
  lvCustomers~addRow( , ,"CU002", "Frith Inc.", "CA 30543")
  lvCustomers~addRow( , ,"CU003", "LMN & Co",    "NY 47290-1201")
  lvCustomers~addRow( , ,"CU005", "EJ Smith",    "NJ 12345")
  lvCustomers~addRow( , ,"CU010", "Red-On Inc.","AZ 12345")
  lvCustomers~addRow( , ,"AB15784", "Joe Bloggs & Co Ltd","LB7 4EJ")
  lvCustomers~setColumnWidth(1)
```

The **~addRow** method adds a row of data into the list view. As can be seen, the data in the list is hard-coded (but this will be fixed in the next exercise). The first parameter is the 0-based index of the item, and defaults to the index of the last item added plus 1 (if no items already in the list view, this defaults to 0). Note however that when the user creates the dialog, the last item appears first not last. This is because the *.rc file specifies the style **LVS_SORTASCENDING**. The second parameter is the index (in an ImageList) of the item's icon should that be required. Finally, the last statement sets the width of the second column to that of the longest text entry. Note that loading the list view with data could have been done in the **initDialog** method. However, the separation of concerns principle points strongly to separating the formatting of the list view from loading data into the list view.

Surfacing a Customer from the Customer List is done in one of two ways: either double click on an item, or select the item and then press the **Show Customer** button. Both invoke the **showCustomer>** method. These two approaches are implemented by the following code (error-handling code omitted):

```
-- 1. Double-Click:

::METHOD initDialog
```

```
        ...
        self~connectListViewEvent("IDC_CUSTLIST_LIST","ACTIVATE",openItem)    -- Double-click
        ...


    ::METHOD openItem UNGUARDED
      self~showCustomer



    -- 2. Select (single click) then press button:

    ::METHOD initDialog
      ...
      self~connectListViewEvent("IDC_CUSTLIST_LIST","CLICK",itemSelected)  -- Single click
      self~connectButtonEvent("IDC_CUSTLIST_SHOWCUST","CLICKED",showCustomer)
      ...

    ::METHOD itemSelected UNGUARDED
      use arg id, itemIndex, columnIndex, keyState
      if itemIndex > -1 then self~enableControl("IDC_CUSTLIST_SHOWCUST")
      else self~disableControl("IDC_CUSTLIST_SHOWCUST")
```

In the first approach, If the user double-clicks on a row, and the row is empty, the second click of the double-click is ignored, else the double-click method (**openItem**) is invoked. This is turn invokes **showCustomer**. In the second approach, the **itemSelected** method is fired when the user clicks on a row in the ListView. If the user clicks on an empty row, then **itemIndex** is set to -1, else it is set to the 0-based row number. As can be seen, both approaches invoke the **showCustomer** method, which is as follows (where **lvCustomers** is the proxy for the List View control):

```
    ::METHOD showCustomer UNGUARDED
      expose lvCustomers rootDlg
      item = lvCustomers~selected
      info=.Directory~new
      lvCustomers~getItemInfo(item, info)
      .local~my.idCustomerData  = .CustomerData~new   -- create CustomerData instance
      .local~my.idCustomerModel = .CustomerModel~new -- create CustomerModel instance
      .local~my.idCustomerData~activate
      .local~my.idCustomerModel~activate
      .CustomerView~newInstance(rootDlg,"CU003")
      self~disableControl("IDC_CUSTLIST_SHOWCUST")
```

First, the relevant row (**item**)is found using the **~selected** method of the List View. Then a directory is created, and the data from the selected row is placed into the directory by the List View's **getItemInfo** method. Thirdly, the Customer Data and Model objects are instantiated, and then the CustomerView is instantiated (**CustomerView** depends on **CustomerModel** being available). As can be seen, in this version of **CustomerListView** the data from the ListView is ignored, and the same Customer is surfaced regardless. This is also true for the other List Views. In the next exercise this will be fixed so that instantiation of the Model and Data objects will be handled elsewhere, and the correct instance will be shown.

Finally, two items about ListViews. First, to change the font for the data in a ListView, use the **createFont** method of the dialog (actually a method in ooDialog's "WindowsExtensions" mixin). For example, try inserting the following in **CustomerListView**'s **initDialog** method, immediately before the statement **self~loadList**:

```
    font = self~createFontEx("Ariel", 10)
    lvCustomers~setFont(font)
```

Save and run. You should see the data in the ListView displayed using the Ariel 10-pitch font.

The second item concerns the appearance of the ListView control. If you place the mouse over one of the headers, its appearance changes and if you click it, it acts rather like a pushbutton. But

nothing happens, although you might expect it to sort the list according to values in the clicked column). The reason it does not is that **CustomerListView.rc** does not include the style **LVS_NOSORTHEADER**. Try adding this to the resource file (change "... LVS_SORTASCENDING, ..." to "... LVS_SORTASCENDING | LVS_NOSORTHEADER, ..."), then re-run. On the other hand, should you wish to change things so that the data is sorted, then check out the **sortItems** method in the "List View Controls" chapter in the ooDialog reference.

6.4. Re-sizing Dialogs

If you haven't already done so, try re-sizing the Order Management dialog. The ListView containing the icons expands to match the new window size, and the two push-buttons both move (and change in size) in proportion. The re-sizing functionality is provided by the ooDialog-provided mixin class **ResizingAdmin**. All that is required is to specify this mixin in the **::Class** statement, and the dialog will be re-sizeable. Note that together with the dialog, controls are also re-sized. Often, however, it is necessary for some controls *not* to be resized, or, for example, only to be re-sized in one dimension. This can be specified, for each control, by using the **controlSizing** method of the **ResizingAdmin** class. Essentially, this method provides for "pinning" individual controls in relation either to the dialog, or to other controls.

Re-sizing is also provided by the **DlgAreaU** class. This ingenious class was written some time ago when ooDialog capabilities were much less developed than they are today. It works by parsing the source code of a subclass' **defineDialog** method at run-time. However, this constrains it to being used only with **UserDialog**, where the dialog template is created through explicit control creation statements. In addition, since the source code is required at run-time, it will not work if the source code is tokenized using **rexxc**. For information about using this approach to re-sizing dialogs, see the documentation for the **DlgAreaU** class in the ooDialog Reference manual or the copious comments in **dlgAreaUDemoThree.rex** (which can be found in the ooRexx samples folder in **oodialog \resizableDialogs\DialogAreaU**).

6.5. Creating Icons

This section discusses first the creation of icons and bitmaps, and secondly how the icons in the **OrderMgrView** dialog are loaded into its icon-style List view.

Various questions arise when creating icons for the first time - especially since the whole area of images in Windows is not, at first glance, simple. This section lists some of the main points about creating icons.

First, it's important to establish whether what's required is an icon (file type *.ico) or a bitmap (file type *.bmp). The "icons" in **OrderMgrView** dialog are actually bitmaps. But a "dialog icon" (the icon displayed in the left hand corner of the title bar of a dialog) is an icon, not a bitmap. A number of tools are available for creating and editing images, icons, bitmaps etc., some of them providing conversion and re-sizing capabilities. One such is GIMP (GNU Image Manipulation Program), a freely distributed piece of software, from http://www.gimp.org.

Second, the size of a dialog icon is variable. That is, an icon larger than the space available will be shrunk to fit. The dialog icons in this exercise are all 64x64 in size, and are automatically shrunk to fit. For resource dialogs, the dialog icon is specified in the resource file, and its ID in the resource file is specified in the **self~execute(...)** method. For UserDialog dialogs, the dialog icon is loaded by the **addIconResource** method. The two arguments to this method are a resource ID and the file name of the icon, for example: **dlg~addIconResource(105,"MyPicture.ico")**. The resource ID is then specified in the **dlg~execute("SHOWTOP, 105)** statement.

Finally, the "icons" in the "Order Management" dialog are bitmaps of size 64x64. These are not shrunk; a smaller icon will look smaller. These bitmaps are loaded into the ListView programmatically. The code that loads the bitmaps into the ListView is as follows (with repetitive statements removed):

```
::METHOD createIconList PRIVATE
  expose iconList
  imgCustList  = .Image~getImage("customer\bmp\CustList.bmp")
  ...
  iconList = .ImageList~create(.Size~new(64, 64), .Image~toID(ILC_COLOR4), 4, 0)
  iconList~add(imgCustList)
  ...
  imgCustList~release
  ...
  return

::METHOD initRecords PRIVATE
  -- Called from init - This method simulates getting the "data" for the OrderMgr view.
  expose records
  records = .array~new()
  ...
  rec = .directory~new
  rec~ID = "ProductList"
  rec~name = "Product List"
  records[2] = rec
  ...
  return records

::METHOD initDialog
  expose records iconList
  self~initDialog:super
  self~lv~setImageList(iconList, .Image~toID(LVSIL_NORMAL))
  do i=1 to records~items
    self~lv~addRow(, i-1, records[i]~name)
  end
```

The icon view requires icons to be loaded from an "image list" - that is, an instance of the **ImageList** class. It is the function of the **createIconList** method (invoked from **init**) to produce such an image list. To build the image list - called **iconList** in the above - each bitmap is first loaded from disk into an instance of the **.Image** class using the **getImage** method. Then the statement **iconList=.ImageList~create(...)** creates an empty image list, into which each of the four images is loaded using the **add** method. Finally, each separate image is released. By the end of this method, an image list has been created, but has not yet been loaded into the list view.

As with the "Record View" used for **CustomerListView**, items in an "Icon View" are loaded as "records". In this case, each record consists of an icon and a text label for that icon. The **initRecords** method does just that - sets up the records in an array called **records**. The record id is used to hold the class name (e.g. "ProductList") of the component to be launched when the user double-clicks an icon.

Finally, in the **initDialog** method, the image list (**lv**) is first set into (added to) the list view, following which the records are added. It is a user responsibility to make sure the sequence of icons in the icon list matches the sequence of text data in the records array.

To complete the behavior of the **OrderMgrView** component, there remains the task, when the user double-clicks on an icon, of surfacing the required component. This is done by the following code in **OrderMgrView**:

```
::METHOD onDoubleClick UNGUARDED
  expose records
  index = self~lv~focused  -- lv is an attribute of the superclass.
  record = records[index+1]
```

```
      self~showModel(record)

::METHOD showModel UNGUARDED
  use arg record
  className = record~ID
  viewClassName = className||"View"
  interpret "."||viewClassName||"~newInstance(self)"
  say "OrderMgrView-showModel-02:"
```

The **onDoubleClick** method is the event handler method defined for the list view. The first statement (after **expose records**) finds which icon has focus - that is, which one was double-clicked. The second retrieves the corresponding record, and then **showModel** is invoked with the appropriate record. In **showModel** an appropriate view is created and surfaced using the **interpet** instruction in much the same way that CustomerList did for individual Customer views (see *Section 6.2.3, "Use of 'Interpret'"*).

It remains only to mention that the above code will allow as many lists to be created and surfaced as the user wishes. This may or may not be what's required. It is of course possible to arrange things so that only a single list for each of Customers, Products and Orders is allowed. In such a case, when the user double-clicks on an icon, the appropriate list would be "surfaced" in the proper sense of the word - that is, created and shown as the top-level dialog, or, if already created, would have focus put on it so that, if hidden under other dialogs, it will pop to the "surface" - that is, become the topmost window on the screen.

6.6. Utility Dialogs

A subject not yet mentioned is the use of ooDialog utility classes and routines that can be used in any ooRexx program. The routines are very simple, and are often one-liners. As an example, in this exercise the startup program provides for entry of a password using the (one-line) **PasswordBox** routine. Invoking **startup enterPW** produces a password box that will accept the password "Password". If you get the password wrong, the startup routine will silently end. The code is as follows:

```
parse arg pwOption
if pwOption = "enterPW" then do
  pwd = PasswordBox("Please enter your password","Sign In")
  if pwd \= "Password" then exit
end
.OrderMgmtView~newInstance
::REQUIRES "OrderMgmt\OrderMgmtView.rex"
```

Check out the ooDialog Reference for the whole set of classes and routines.

Towards A Working Application

7.1. Introduction

This chapter, and the accompanying Exercise, provides much of the infrastructure for an application that uses model-view-data component concepts. The infrastructure implements a pattern called the "Model-View Framework" which removes from the application developer most of the work involved in instantiating View, Model and Data components, reading data from disk, and providing that data to a dialog.

Open the **Exercise07** folder and start the Order Management application by double-clicking on **startup.rex**. Try it out. Explore the function which, while not complete, is much more so than in the previous exercise. In particular, note that application data is now read from files. For example, the Customer data is read from the file **CustomerFile.txt**. However, although data can be changed in some dialogs, the changed data does not (in this exercise) update the files.

Also, note that the **Help** menu on the main Sales Order Management dialog now includes an option **Message Sender** (discussed in *Message Sender*). Click this option and a "Message Sender" dialog opens. This sends messages to (invokes methods on) the various components, and is a useful debugging tool that replaces the "stand-alone" function used in Exercise 6. For example, try sending a "query" message to the Customer whose key is BA0314. To do this, key "CustomerModel BA0314" (without the quotes) in the **Target** field , "query" in the **Method** field, and then press **Send**. The customer's data is returned in the **Reply** field as a name-value string.

Now try using the Message Sender to surface a Product dialog - say the view for Product CU003. To do this, select **ObjectMgr The** in the **Target** combo-box pull-down, **showModel** in the **Method** field, and type **ProductModel CU003** in the **Data** field. Now press **Send**. The Product dialog for instance CU003 appears.

The "Object Manager" (which could also be called "ComponentManager" since it manages application components as opposed to any old ooRexx object) is a support class that instantiates a component by invoking its **newInstance** class method. It also keeps track of which components have been instantiated. For more detail see *The Object Manager*

Consider now what has to happen to display a Product dialog. First, a **ProductData** instance must be created, and its data is read from disk. Then the appropriate **ProductModel** component must be instantiated and its data retrieved from the **ProductData** component. Finally, an instance of **ProductView** has to be created and its data retrieved from the **ProductModel** component by invoking its **query** method.

This sequence is a pattern - the "Model-View Framework" pattern - that can be applied to most business components, and hence can be handled by superclasses instead of by duplicated code in application components.

The next section in this chapter introduces the Model-View Framework - its objectives, a brief overview of its function, an example of use, and the classes that implement it. Then *Components and Data* discusses first the different "kinds" of application component and also the data formats returned by data components. The fourth section provides additional detail on the *Message Sender*, and then *Dialog Re-sizing* for the Order Manager dialog is re-visited. Finally the use of Control Dialogs for the *Order Form* is described.

7.2. The Model-View Framework

This section presents the externals of the MVF. For some additional detail, see *Appendix D, The Model-View Framework*.

7.2.1. MVF Objective

The objective of the Model-View Framework (MVF) is to provide a mechanism whereby application components can read and write data and display views without needing to be aware of *how* this is done. Thus the MVF supports view-model-data separation of concerns in application components. The MVF comprises three superclasses for application components called **Model**, **xxView** (where "xx" is "Rc", "Res" or "Ud") [1], and **GenericFile** (for data components), plus two "manager" objects: **ObjectMgr** and **ViewMgr**.

When a user double-clicks on an item in say the Customer List dialog, the confident expectation is that a Customer dialog that shows the data associated with the list item will be displayed. Now consider what has to happen to make that Customer dialog appear:

1. Create the appropriate data component, which...

2. ... opens the correct file and reads the data.

3. Create the model component and provide it with its "key" (e.g. customer number).

4. In the model component, get a reference to the data component.

5. Invoke a method on the data component to retrieve the data associated with the model's key.

6. Create a view component.

7. Provide the view component with the model's data.

8. Make the dialog (including data) visible.

This sequence set of actions assumes that none of the component instances involved are yet activated. However, the MVF must also work when some or all are activated. For example, if a Customer dialog exists but is minimized, and the user double-clicks on that customer in a Customer List dialog, then the MVF need only surface the Customer dialog. Thus the MVF distinguishes between a number of different states, and relieves the programmer from having to code the logic for each component and for each possible state.

Consider, for example, the code in **OrderMgrView** that launches a List View. On the left is the Exercise 6 code, and on the right is the Exercise 7 code (equivalent or identical statements have been placed on the same line for comparison).

```
                              OrderMgrView
        Exercise 6                                          Exercise 7
        ----------                                          ----------
 ::METHOD showModel UNGUARDED                  ::METHOD showModel UNGUARDED
                                                 expose idObjectMgr
  use arg record                                 use arg record
  className = record~ID                          className = record~ID
  viewClassName = className||"View"
  interpret "."||viewClassName||-                r = idObjectMgr~showModel(classname, -
     "~newInstance(self)"                              "a", self)
```

The key difference is that in Exercise 6 the Customer List is launched without any concern for the data - because the data is hard-coded in the List View. In Exercise 7, on the other hand, the data is read from disk and provided to the List View. This is done by the **showModel** method of the Object Manager (**idObjectMgr**), its id having been retrieved from **.local** in the **init** method.

---

[1] The three "xxView" classes are identical except for their superclasses - RcDialog, ResDialog and UserDialog respectively (in the next exercise, it is planned that these will be merged into a single mixin class).

In the Exercise 7 code, the second parameter in the **showModel** method is "a". This indicates that the List View instance is "anonymous". Instance names and "kinds" of component are discussed in *Section 7.3.1, "Kinds of Component"*.

Now that the Customer List View has been created, consider what happens when the user double-clicks on an item in the list. In Exercise 6, the list data was hard-coded. In Exercise 7 it is read from disk by the **CustomerData** class. In both exercises, launching a Customer View from the list (when the user double-clicks on a list item) is done by the **showCustomer** method as follows (excluding code common to both):

```
                        CustomerListView
  Exercise 6                                        Exercise 7
  ----------                                        ----------
::METHOD showCustomer UNGUARDED              ::METHOD showCustomer UNGUARDED
  ...                                          ...
  .local~my.idCustomerData  = .CustomerData~new
  .local~my.idCustomerModel = .CustomerModel~new
  .local~my.idCustomerData~activate
  .local~my.idCustomerModel~activate
                                               objectMgr = .local~my.ObjectMgr
  .CustomerView~newInstance(rootDlg,"CU003")   objectMgr~showModel("CustomerModel", -
                                                 info~text, rootDlg)
  ...                                          ...
```

In Exercise 6, before the CustomerView is instantiated, the CustomerData and CustomerModel components must be instantiated, and their object IDs stored in **.local** for later access by the CustomerView and CustomerModel objects. This is needed because, although no data is actually read from disk, the CustomerView invokes a method on CustomerModel which invokes CustomerData.

In Exercise 7, on the other hand, the value of **info~text** (the index read from the selected ListView row) is the Customer Number, and is used as the Customer object's instance name. The Object Manager's **showModel** method then manages or "choreographs" the sequence of method invocations required to surface the Customer View dialog, and (if necessary) create the appropriate model and data objects. This choreography, using function in the **Model**, **xxView** and **GenericData** superclasses, results in the Customer View being surfaced with its data read from disk and displayed. See *Section D.3, "MVF Operations"* for a full description of the way in which this is done.

Finally, note that in Exercise 7, the names of the possible List Model classes (e.g. **CustomerListModel**) are hard-coded in the **initRecords** method of **OrderMgrView** (although these would arguably be better placed in some configuration file).

7.2.2. MVF Overview

The MVF consists of five classes: **ObjectMgr**, **ViewMgr**, **Model**, **xxView** (i.e. RcView, ResView, and UdView), and **GenericFile** (see *MVF Classes* for more detail). These are all located in the folder **ooRexx\samples\oodialog\userGuide\exercises\Exercise07\Support**. Together, they provide for the three different types of application component - view, model, and data. Each model gets its data from its data component, and each model has a single view. (A production-strength MVF could support multiple views of the same model by providing an "Open as..." option for a given icon or list item. This would display a selection of views, similar to the "Open with..." function provided by a button-2 click on an item in Windows Explorer.)

MVF requires that each component has a text name, the name being the class name of the main class (such as "CustomerModel", "ProductView" or "OrderData") together with an instance name. For components with a "key" such as Customer Number, the instance name is the key (e.g. "CustomerModel BA0314"). For components that are "singletons" - that is, there can logically be only one instance, the name is "The". An example of a singleton is a data component (e.g. "CustomerData

The"). Finally, some components - such as lists - are anonymous: when an anonymous component is instantiated, its instance name is given as "a" or "A", and its real instance name - such as a number as in "CustomerList 3" - is assigned by the **Model** super-class (part of the MVF) during instantiation. The instance name is important to MVF since its internal logic differs slightly depending on which kind of instance name is used - a "key" name, a singleton name, or an anonymous name. (Note that this naming convention could be relaxed if components were named in a configuration file; however, the distinctions between the different kinds of component would remain.) See *Section 7.3.1, "Kinds of Component"* for further discussion.

7.2.3. An Example - The 'Person' Component

A key question is, what does the MVF look like to the programmer? This section answers this question using the Person component (which has minimal function) as an example. But first, using the Message Sender, try sending a **showModel** message to **ObjectMgr The** with the data **PersonModel PA150**. The Person dialog appears, and the MVF has handled the task of ensuring that both the Data and Model components are active before the View is launched. First, MVF checks to see if they are already activated; if not, it instantiates them (the Data instance first, then the Model). Second, on instantiation, **PersonModel**'s superclass asks the **PersonData** instance for its data. Third, **PersonView**'s superclass asks **PersonModel** for its data. Finally, the dialog for Person PA150 appears.

The code required to conform with the MVF is shown in the Person component in **ooRexx\samples \oodialog\userGuide\exercises\Exercise07\Extras\Person**, and those requirements are as follows:

- **A Data Component** (a subclass of **GenericFile**)

```
::METHOD newInstance CLASS PUBLIC
  ...
  -- Check if an instance has already been created; if so, return .false.
  ...
  idData = self~new()
  return idData

::METHOD init PRIVATE
  ...
  records = self~init:super(fileName, columns)
  ...
```

Data components such as **PersonData** are required to provide a **newInstance** class method, which is invoked by the MVF. No parameters are provided. This method first checks if an instance has already been created. If not, it is created, and its object ID is returned to the MVF (i.e. to the caller).

In the **init** method, the superclass' **init** method is invoked with the filename and the number of columns in the file as parameters. Invocation of super with these parameters is an MVF requirement.

- **A Model Component** (a subclass of **Model**)

```
::METHOD newInstance CLASS PUBLIC
  use strict arg instanceName
    forward class (super) continue
    modelId = RESULT
    return modelId

 ::METHOD init
```

```
    use strict arg myData
```

Model components such as **PersonModel** are required to provide a **newInstance** class method with one required argument - the model's instance name. The method must be forwarded to the **Model** superclass, which first retrieves the intended instance's data from its data component, and then creates an instance of itself with the instance data as a parameter. The new instance must then be returned.

- **A View Component** (a subclass of **RcView**, **ResView**, or **UdView**)

```
::METHOD newInstance CLASS PUBLIC
  use strict arg modelId, rootDlg
  -- create dialog, e.g. "dlg = .PersonView~new(...)"
  dlg~activate(modelId, rootDlg)
  return dlg

::METHOD activate UNGUARDED
  use strict arg modelId, rootDlg
  forward class (super) continue
  personData = RESULT
```

View components such as **PersonView** must provide a **newInstance** class method and an **activate** method. The **newInstance** method is invoked by MVF with the view's model id (and also the root dialog - that is, the Order Manager dialog). After the dialog is created, MVF requires that **activate**, with the model's id as the first argument, be invoked on the new dialog. In the **activate** method, the superclass must be invoked using **forward**. The superclass returns the Model's data in **RESULT**. Finally, the id of the new dialog must be returned.

The above is how a "named" component uses MVF. By "named" is meant a component whose identity is a combination of its class and a specific "key" such as a Customer Number, or Product Number. However, there are three other kinds of component: a "singleton" such as a data component, a "form" such as the Order Form, and "anonymous" such as a Customer List. These are discussed below in *Section 7.3.1, "Kinds of Component"*.

7.2.4. MVF Classes

This section describes each of the classes that comprise the MVF. They are, the Object Manager, the View Manager, plus three superclasses, one for each of Model, View, and Data components.

7.2.4.1. The Object Manager

The Object Manager (**ObjectMgr.rex** in the **userGuide\exercises\Support** folder) is a "singleton" class (there can only logically be one of them) and has the external name "ObjectMgr The". It maintains a table (called the "Object Bag") of all instantiated components. The public methods of the Object Manager are:

- **getComponentId** (with parameters *className* and *instanceName*) - Returns the id of the requested component. If the id is not in the Object Bag, then it sends **newInstance** to the class object *className*. If the class does not not exist, **.false** is returned.

- **list** - Lists the contents of the ObjectBag on the Command Prompt. The following shows the results of sending **list** to the Object Bag after (a) the application was started, then (b) the **Customer List** icon was double-clicked, (c) a Customer in the list was double-clicked, and (d) the *Message Sender* was used to send a **query** message to **ProductModel LM400**:

```
Object Bag List:
```

```
--------------------------------------------------------------------------
Class-Instance             Model Id                 ViewClass-Inst

-----------------------    ----------------------   ----------------------
   CUSTOMERLISTVIEW-26701042 a CUSTOMERLISTVIEW        .nil
   PRODUCTMODEL-LM400        a PRODUCTMODEL            .nil
   CUSTOMERVIEW-267047652    a CUSTOMERVIEW            .nil
   CUSTOMERLISTMODEL-1       a CUSTOMERLISTMODEL       CUSTOMERLISTVIEW-26701042
   CUSTOMERDATA-THE          a CUSTOMERDATA            .nil
   PRODUCTDATA-THE           a PRODUCTDATA             .nil
   CUSTOMERMODEL-BA0314      a CUSTOMERMODEL           CUSTOMERVIEW-267047652

   -------------------------------------------------------------------------
```

The instance name for a View component is derived by invoking **identityHash** on its id.

- **showModel**(className, instanceName) - Shows the View for the specified Model. If the View exists, then it is surfaced. If not, then if the Data component is not already instantiated, it is instantiated. If the Model component is not already instantiated, it is instantiated. Then the View is instantiated. All instantiations use the *newInstance* method. All dialogs except the "application" dialog (that is, Order Manager) and the Message Sender are created and/or surfaced using the **showModel** method.

For further information on how the **showModel** method of the Object Manager works, see *Section D.3, "MVF Operations"*.

7.2.4.2. The 'Model' Superclass

Model is the MVF superclass for all model components, and provides key methods for subclasses as follows:

- **newInstance** - invoked by the Object Manager (which ensures that the required data component is instantiated) with an instance name as the single parameter. The id of the data component is retrieved from Object Manager, after which **getRecord** (or **getFile** for a "list" component) is invoked on the data component. Then **self~new** is invoked with model's data as a parameter. **Model** also provides an instance attribute **myData** that contains the instance data returned from the data component.

- **getInstanceName** is invoked by the Object Manager's **showModel** method for anonymous components only (such as a **CustomerListModel**). It adds 1 to a class variable and returns it. (However, for **OrderFormModel**, the method is over-ridden and **OrderFormModel** itself returns a new order number (see *Order Form*).

- **query** - A component framework generally requires that components provide specific methods defined by the framework. Aside from instance creation methods, a "well-known" method is required for MVF to access a model component's data. This method has the name "query", and it must conform to a specific protocol as follows:
  - If a component's **query** method is invoked with no parameters, then it must return a directory containing all the "public" data it has. The directory indexes are the labels for the data as defined in the "database" (although this is not usually the case for real production-strength systems, where the data dictionary for application-level components often differs from the column names in an SQL database).

    For example, use the Message Sender to send *query* to *PersonModel PA150*. A directory is returned by **PersonModel**, and the Message Sender presents the directory in name-value form in its **Reply** field as follows:

    ```
    dob: 751513; baseSalary: 38000; number: PA150; jobDescr: Packer;
    ```

```
familyName: James; firstName: Alfred;
```

- If a component's `query` method is invoked with one parameter, and when that parameter is a directory, an array, or a string, then only those fields specified by name are returned.

  For example, use the Message Sender to query the first name and family name for "PersonModel PA150". To do this, specify the fields by name (case-sensitive) in the **Data** edit field, as follows: ***firstName familyName*** (note - field names are case-sensitive). On pressing the **Send** button, the data ***firstName: Alfred; familyName: James;*** is returned as a directory which Message Sender unpacks and presents as a string in the ***Reply*** field.

  For debugging purposes, you can use MessageSender to send a set of data names as a string (as in the example just given), a directory, or an array. To send as a directory, enclose each name in square brackets, e.g.: **[firstName] [familyName]**. To send as an array, place a vertical bar before each name, e.g: **| firstName | familyName**.

7.2.4.3. The 'View' Superclass

There are three View superclasses in the **Support** folder: `RcView`, `ResView`, and `UdView`. They are identical except for their superclasses - `RcDialog`, `ResDialog` and `UserDialog` respectively (their function could be provided by a single mixin class, and this is planned for a later version of the User Guide exercises). Important methods are:

- `activate` - This is invoked by the subclass, which must provide the view's model id as the single argument. A `query` message is invoked on the model, which returns the model's data as a directory. In addition, this method saves information used to tidy up the view in the `leaving` method.

- `leaving` - This method is automatically invoked by ooDialog when a dialog closes. Its only function is to inform the Object Manager that the view is about to close. The Object Manager then removes the view from the Object Bag (otherwise it might later try to surface a non-existant view!).

- `offset` - This method offsets dialogs from the Order Management dialog when first opened. Although not used elsewhere in this exercise, the effect can be seen using the "Person" component in the `Exercise07\Extras\Person` folder. First, use the Message Sender to launch a Person dialog (for example send ***showModel*** to ***ObjectMgr The*** with the data ***PersonModel PA150***). The Person dialog appears in the center of the screen. Now un-comment the last line (`--self~offset:super`) in the `initDialog` method of `PersonView`, save, re-start the application, and launch the Person dialog as before. Note that the dialog "flickers" when opened - it seems to open for a fraction of a second in the centre of the screen, then re-appears offset from Message Sender. The flicker results from the .rc file containing the dialog property WS_VISIBLE (in ResEdit the behavior property "visible" is set to true). First it appears in the center of the screen, then moves to the offset position. Now remove "`| WS_VISIBLE`" from the .rc file, save, and re-run. The Person dialog appears without a flicker but offset (from the Message Sender).

7.2.4.4. The 'GenericFile' Superclass

The data superclass is called "GenericFile" since it acts on any file having the defined format. Open any of the .txt data files (e.g. `PersonFile.txt` to see the format. Essentially, the first line in the file must be the column names (or labels). A label must not contain spaces. Lines 2 to *n* are the data values, each separated by a vertical bar character. The main methods are:

- **getRecord** - Invoked with a record key (e.g. a Customer Number) as its single argument, reads the record from the file defined in the subclass, and returns a "record directory".

- **getFile** - Returns the file in "file as directory" format.

- **list** - Lists the file on the console.

See *GenericFile Data Formats* for a description of the "record directory" and "file as directory" formats.

7.3. Components and Data

7.3.1. Kinds of Component

There are four "kinds" of components in Exercise07: "named", "singleton", "anonymous", and "form".

- A "**named**" component instance is identified by a unique name derived from the instance's data (analogous to a database key). An example is **CustomerModel**, where each instance is identified by its Customer Number. The external name for such an instance is of the form model class name, model instance name - e.g. "CustomerModel AB0784". Note that a "Form" component such as the Order Form is of the named component kind, since although it starts out without a name, it is (and must be) given its name (such as an order number) when first instantiated. This is done when the Object Manager invokes its **getInstanceName** class method. A View component is named by its object reference number (that is, the number returned by invoking **identityHash** on the view instance).

- "**Singleton**" instances are those for which there can logically only be a single instance - for example, data components such as **CustomerData**, or the Order Manager (which in Exercise07 is a view-only component). Their instance name is always "The".

- An "**anonymous**" component is one for which there can logically be more than one instance, but which do not have any obvious distinguishing name. Thus they are initially given the instance name "A". Examples are list components such as **CustomerListModel**. Instances of an anonymous component are provided with a system-generated number. For example, the name of a Customer List Model is a unique number generated by its superclass, starting at '1'. For example, to create an instance of **CustomerListModel**, the message **getComponentId("CustomerListModel","A")** is sent to **ObjectMgr** which, on seeing instance name "a" or "A", invokes **getInstanceName** on the ListModel class object, which is handled by the **Model** superclass. **Model** returns a number starting at "1".

- A "**form**" instance such as a Sales Order Form or a Purchase Request form is a special kind of component. Initially it is anonymous, and although when created there is no database record of it, there will be after it's completed and the user hits OK. A new number (e.g. an order number) is assigned to a form when it is created, and this number is used as the database key when, after completion, the form is committed to the database. For example, the Order Form component assumes that this will happen, and so its instance name is a unique Sales Order number. This is created by the **getInstanceName** class method of **OrderFormModel** which over-rides the same method in its superclass (**Model**).

The above classification covers almost all the kinds of dialog found in a typical UI environment handling business systems. To test this, consider the "Words of Wisdom" business component in Exercise03, which was implemented as a view, a model, and a data component. The **...Exercise07\Extras\Wow4** folder contains the same set of components, but modified to use the MVF. Code no longer required is commented out with the comment **v01-00**; methods or statements added for MVF use are commented with **MVF**; modified statements are commented **v01-00-->MVF**;

and unchanged statements are commented **v01-00**. (Actually, Wow is somewhat schizophrenic, in that it can be launched either as a singleton or as anonymous - that is, with an instance name of "The" or "A". If "The", then only one instance is allowed. If "A", then multiple instances can be created.)

As mentioned above, this whole approach of having component names define the type of component is not particularly scalable. A better approach - certainly for production-strength apps - is to provide a configuration file that names the classes and states what type they are. Such a file might look something like this, and would remove the need for using class names as the basis for managing instances:

```
<modelClass name="Product",     type="named", dataClass="ProductDB"/>
<modelClass name="NewOrder",    type="form"/>
<modelClass name="CustomerList",type="anonymous"/>
<modelClass name="SalesOrder",  type="named"><viewClass name="MySpecialView"/></modelClass>
<modelClass name="Wow",         type="singleton"
   <viewClass name="WowView">
   <dataClass name="WowData" source="sql">
</modelClass>
```

7.3.2. GenericFile Data Formats

Fields in a file record are separated by a vertical bar character, and field names are defined in the first line of the file. All data files have the file extension ".txt". **GenericFile**'s **getRecord** method returns a single record (for example a Customer record) in a "record directory", whose format is as follows:

```
    "Record Directory" format (using sample data values):

            Indexes      Items
          +- - - - - - - - - - - - - -+
          | CustNo   | BA0314        |
          +- - - - - + - - - - - - - -+
          | CustName | LMN & Partners |
          +- - - - - + - - - - - - - -+
          | Zip      | 84394         |
          +- - - - - + - - - - - - - -+
          + ...      | ...           |
          +- - - - - - - - - - - - - -+
```

A complete file (for example the Customer File retrieved and displayed by the CustomerList component) is returned by **GenericFile**'s **getFile** method in "File as Directory" format, as follows:

```
    "File Directory" format (using sample data values):

            Indexes      Items
    +- - - - - - - - - - - - - - - - - - - - - - - +
    |Headers | CustNo  | CustName           | .... |   1D array
    |- - - - + - - - - + - - - - - - - - - - -+ - - -|
    |Records | AB0784  | ABC Enterprises Inc. | .... |   2D array
    |        | - - - - + - - - - - - - - - - -+ - - -|
    |        | AC0027  | Frith Motors Inc.   | .... |
    |        | - - - - + - - - - - - - - - - -+ - - -|
    |        |   ...   + ...                + ... |
    |- - - - + - - - - - - - - - - - - - - -+ - - -+
    |Count   | n |                                    Integer
    +- - - - + - +
```

The data format for an Order is a combination of data from three files - OrderData, CustomerData, and ProductData - and is described in *Compound Data*

7.3.3. Compound Data

"Compound data" is data assembled from two or more files. In Relational Data Base terms, this is a "join". The one example of compound data in Exercise 7 is the **OrderData** class, where the **init** method uses the superclass's **getFile** method to read each of the two Order files (OrderHeader.txt and OrderDetail.txt). This data is stored in two attributes, **dirOrderHeaders** and **dirOrderLines**. Then the **addCustomerInfo** method is called to get selected Customer data (e.g. Customer Addresses) from the **CustomerData** component, and this data is then added to the **dirOrderHeaders** attribute. Finally, the **addProductInfo** method accesses the **ProductData** component to get selected Product data (e.g. Product Names) which is added to the **dirOrderLines** attribute.

The **dirOrderHeaders** format is as follows:

```
 Indexes  Items
+----------------------------------------------------------------------+
|Headers|OrderNo|CustNo|Date  |Disc|Cmtd|Cust |Cust|CustAddr | Zip    |   <a 1D array>
|       |       |      |      |    |    |Name |Disc|         |        |
|- - - -+- - - -+ - - -+ - - -+- - +- - +- - -+- - +- - - - -+ - - - -|
|Records|SO-1234|AB0784|120821| 2  | N  |ABC..| B1 |2154 En..|FL 37043|   <a 2D array>
+- - - -+- - - -+ - - -+ - - -+- - +- - +- - -+- - +- - - - -+- - - - |
        |SO-2345|BA0314|110815| 1.5| Y  |LMN..| C2 |116 Hig..|NV 84394|
        |- - - -+ - - -+ - - -+- - +- - +- - -+- - +- - - - -+- - - - |
        |SO-3456|BA0314|120527| 0  | Y  |LMN..| C2 |116 Hig..+NV 84394|
        |- - - -+- - - +- - - +- - +- - +- - -+- - - - - -|- - - - |
        |SO-4567|CU0003|120630| 0  | Y  |Red..| A1 |43 Main..|AR 48231|
        |- - - -+ - - -+ - - -+- - +- - +- - -+- - +- - - - -+- - - - |
        |SO-4569|AC0027|120824| 5  | N  |Fri..| B1 |124 Fre..|TX 78254|
        +----------------------------------------------------------+
```

And the format of the **dirOrderLines** attribute is as follows:

```
     Indexes    Items
    +------------------+
    | Count   | 12     |
    |- - - - -+- - - - -+- - - - -+- - -+- - - - - -+
    | Headers | OrderNo | ProdNo  | Qty | ProdName   |    <a 1D array>
    |- - - - -+- - - - -+- - - - -+- - -+- - - - - -|
    | Records | SO-1234 | AB100/W |   5 | Baffle     |    <a 2D array>
    +- - - - -+- - - - -+- - - - -+- - -+- - - - - -|
            | SO-1234 | CF300/X |   6 | Widget Box |
            |- - - - -+- - - - -+- - -+- - - - - -|
            | SO-1234 | EF500/W | 15 | Slodget    |
            |- - - - -+- - - - -+- - -+- - - - - -|
               ...       ...       ...    ...
            |- - - - -+- - - - -+- - -+- - - - - -|
            | SO-4569 | XY200   | 12 | Blad Anchor |
            +- - - - -+- - - - -+- - -+- - - - - -+
```

Since the Order data is held in two attributes of **OrderData**, the **getRecord** and **getFile** methods are over-ridden, and handled entirely by the **OrderData** class. While **getFile** is very simple - merely returning **dirOrderHeaders** (which is sufficient for the Order List component), the **getRecord** method needs to build the data for a single Order from both **dirOrderHeaders** and **dirOrderLines**. Thus it also over-rides its superclass method, and builds a directory called "dirOrderRecord" whose format is as follows:

```
   "dirOrderRecord" format (using data values from Order No SO-1234):

           Indexes        Items
         +- - - - - - - - - - - - - - - - - - - - - - - +
         | OrderNo      | SO-1234                       |
```

```
          +- - - - - - - -+- - - - - - - - - - - - - - - - +
          | CustNo       | AB0784                         |
          +- - - - - - - -+- - - - - - - - - - - - - - - - +
          | CustName     | ABC Enterprises Inc.           |
          +- - - - - - - -+- - - - - - - - - - - - - - - - +
          | CustAddr     | 2145 Engle Blvd,Hardtown,FL    |
          +- - - - - - - -+- - - - - - - - - - - - - - - - +
          | Zip          | 37043                          |
          +- - - - - - - -+- - - - - - - - - - - - - - - - +
          | Cmtd         | N                              |
          +- - - - - - - -+- - - - - - - - - - - - - - - - +
          | CustDisc     | B1                             |
          +- - - - - - - -+- - - - - - - - - - - - - - - - +
          | Disc         | 2                              |
          +- - - - - - - -+- - - - - - - - - - - - - - - - +
          | Date         | 120821                         |
          +- - - - - - - -+- - - - - - - - - - - - - - - - +
          | OrderLineHdrs | OrderNo  ProdNo Qty ProdName  |     <a 1D array>
          +- - - - - - - -+- - - - - - - - - - - - - - - - +
          | OrderLines   | SO-1234  AB100/W  5 Baffle     |     <a 2D array>
          |              | SO-1234  CF300/X  6 Widget Box |
          |              | SO-1234  EF500/W 15 Slodget    |
          +- - - - - - - - - - - - - - - - - - - - - - - - +
```

Finally, a **listOrders** method is provided, since the **list** method of **GenericFile** cannot list data from more than one file.

7.4. The Message Sender

The Message Sender is launched from the **Help** menu of the **Sales Order Management** dialog and is used to "send messages to" (aka "invoke methods on") components. It illustrates a useful kind of debugging aid, and replaces the special component-specific "startup" scripts provided in Exercise 6. While a useful aid, it is provided as merely a demonstration of the kind of debugging aid that can be deployed when using a component-based architecture with a Model-View Framework. Thus it does not pretend to be all-encompassing, and the results of sending some messages may be unpredictable. In addition, its display of data returned is limited. For example, a "query" message sent to a List component only displays the directory indexes and items as follows:

```
   RECORDS: an Array; COUNT: 5; HEADERS: an Array;
```

Some commonly-used target objects and messages are provided in the two combo boxes **Target** and **Method**. For repetitive testing of a given component, additional targets and messages can be temporarily "stored" in the combo boxes, so saving in typing time. However, such additions to the combo boxes are thrown away when the Message Sender is closed.

The "Data" section can be used to send data to a component. The data formats sent must be either a string, an array, or a directory. To send an array, place a "|" (vertical bar) character before each array element. To send a directory, use square brackets for indexes- for example, [Name] Jim Brooks [Age] 34.

The Message Sender is located in the **Support** folder. It is implemented as two ooRexx classes and a routine. The classes are a main dialog class and a separate data-only class for visible strings. The routine sends a message by constructing an instance of the ooRexx **Message** class, then invoking its **send** method. Multiple copies of the Message Sender can be launched concurrently.

7.5. Revisiting Re-sizing

In Exercise 6, the Order Manager is a re-sizeable dialog, However, when re-sized, all controls were also enlarged (or shrunk). Normally, to have all controls change size is not a requirement; rather controls such as push buttons and edit fields should usually not change their size.

In Exercise 7, only the "container" for the icons is required to re-size. Other controls should not change. This is achieved by "pinning" the other controls so that they do not move or expand/contract. For example, the "Reset Icons" button is pinned to the left side and to the bottom side of the dialog, so preventing the button from moving away from the bottom-left of the dialog. In addition, to prevent the button changing its size, the top side of the button is pinned to the bottom side, and the right side is pinned to the left side. Code to define such constraints must be placed in a **defineSizing** method which is automatically invoked by ooDialog before the underlying dialog is created. If nothing is defined, the default is to do nothing. Note that in this method, no other method that requires the underlying dialog to exist can be invoked. Note also that this method must return **.false**.

Specifying how controls behave when the dialog is re-sized is done by invoking the **controlSizing** method (a method of ooDialog's **ResizingAdmin** class) on 'self'. As an example, the following code defines the resizing behavior of the "Reset Icons" button on the Order Management dialog:

```
::METHOD defineSizing
  ...
  self~controlSizing(IDC_ORDMGR_RESET, -
                     .array~of('STATIONARY', 'LEFT'  ), -
                     .array~of('STATIONARY', 'BOTTOM'), -
                     .array~of('MYLEFT',     'LEFT'  ), -
                     .array~of('MYTOP',      'TOP'   )  -
                    )
```

The **controlSizing** method takes five arguments: a control's resource ID (e.g. the Reset button) and four arrays. The first array addresses the left side of the control, the second the top, the third the right, and the fourth the bottom. So: left, top, right, bottom. For each side there is an array, and each array has three items. First the type of pin. Second the edge of the "other" window (remembering that each dialog and each control is actually a "window"). Third the id of the other window to which this control is pinned, the default being the resource id of the main dialog (in our case the Order Management dialog.

So, taking the second parameter as an example, "STATIONARY" means that the left side of the Reset button must not move away (or towards) the second parameter, i.e. the "LEFT" side of the dialog.

Consider the last parameter - the array 'MYTOP', 'TOP'. The keyword 'MYTOP' is a special keyword that can only be used for the bottom edge of a control. It pins the bottom edge of the control to its top edge. This has the effect of keeping the height of the control constant. Similarly, 'MYLEFT' pins the right side of the control to the left side, so keeping the width of the control constant. Note that in this case the second parameter is ignored, although it must be a valid sizing parameter.

The result of all this is that the Reset button does not move from the bottom left corner of the dialog, and its size remains constant.

For further information about the ResizingAdmin class, see the ooDialog Reference. In addition, the folder **Program Files\ooRexx\samples\oodialog** contains examples of re-sizeable dialogs in **resizableDialogs\ResizingAdmin**. Here, the program **augmentedResize.rex** has copious and excellent comments on the various aspects of re-sizing. Worth a read!

7.6. The Order Form

Open an Order Form from the icon in the Order Management dialog. Although still not functional, the format of the dialog *looks* much more like a usable sales order form than in the previous exercise.

The main part of the form is a Tab Control with two tabs - one for entering customer details, the other for product details.   ooDialog supports two approaches to embedding content in a Tab Control: a Property Sheet Dialog with a Property Sheet Page for each tab, and a Resource Dialog with a Control Dialog for each tab.[2] One important difference is that while the Property Sheet Dialog approach is simpler (more is handled by the operating system), it does not allow for interesting controls to be placed on the main dialog outside the Tab Control. Thus the alternate approach - Control Dialogs - is used for the Order Form.

The instance name for the new Order Form is the Order Number. When the icon on the Order Management dialog is double-clicked, the MVF is used to surface the OrderForm dialog. The sequence of operation is as follows (with detail irrelevant to the Order Form omitted):

- **OrderMgrView** sends **showModel("OrderFormModel", "a")** to the Object Manager.

- The Object Manager sees that the instance name is "anonymous", and so sends **getInstanceName** to the class object (**.OrderFormModel** in this case).

- **OrderFormModel** provides this class method, and returns the next Order Number.

- The Object Manager uses the new Order Number as the instance name for the Order Form, and...

- ... sends **newInstance(instanceName,...)** to **.OrderFormModel**, which then instantiates an **OrderModel** instance.

- The Object Manager then sends **newInstance** to **.OrderFormView**, which then creates an instance of the **OrderFormView** dialog.

The Order Form consists of three dialogs - the main Resource File Dialog (an **RcDialog**) plus two Control Dialogs (**RcControlDialog**) in a Tab Control. The two Control Dialogs - one for Customer Details and one for details of Products ordered, are created in the **activate** method as follows:

```
cd1 = .CustDetailsDlg~new("Order\OrderFormView.rc", IDD_ORDFORM_CUST_DIALOG)
cd2 = .OrderLinesDlg~new("Order\OrderFormView.rc", IDD_ORDFORM_ORDLINES_DIALOG)
tabContent = .array~of(cd1, cd2)
cd1~ownerDialog = self
self~prep(tabContent)
```

The **prep** method is then called to set up the tabs:

```
::METHOD prep
  expose tabContent lastSelected havePositioned
  use strict arg tabContent
  havePositioned = .array~of(.false, .false)
  lastSelected = 0
  self~connectTabEvent(IDC_ORDFORM_TABS, SELCHANGE, onNewTab)
```

The 'havePositioned' array is used to determine if the page dialogs have been positioned, and both are marked as not positioned. Then "no tab yet selected" is set. Finally, an event handling method is connected to the event **onNewTab** (which is invoked when the user clicks on a tab).

Next, in the **initDialog** method, the tabs are added to the tab control, their size is calculated, and the control dialog used for the first page is positioned and shown:

```
::METHOD initDialog
```

---

[2]By "resource dialog" is meant an RcDialog, a ResDialog or a UserDialog. The two former have resources defined in a resource file, whereas the latter's resources are defined programmatically. See the ooDialog Reference for a full description.

```
    expose ... tabContent lastSelected ...
    ...
    tabControl = self~newTab(IDC_ORDFORM_TABS)
    tabControl~addSequence("Customer Details", "Order Lines")
    ...
    self~calculateDisplayArea
    self~positionAndShow(1)
```

The two methods **calculateDisplayArea** and **positionAndShow** are well-commented, and are copied from the ooDialog sample program **oodListViews.rex** - one of the excellent samples in the **samples\oodialog\propertySheet.tabControls** folder, which is in the **Program Files \ooRexx** folder.

Note that it's essential to properly close the control dialogs in the **cancel** and/or **ok** methods. This must be done using the **endExecution** method.

Finally, the Order Form illustrates one use of the **DateTimePicker** control.   This is a very fully-featured control, providing many ways of displaying and manipulating date and time. For the Order Form it allows the user easily to specify the Order Date. It's partly configured in the .rc file (in ResEdit, **Use Spin Control** is set to **false** and **Format** is set to **Short Date**), and partly in code, as follows:

```
::METHOD initDialog
  ...
  orderDate = self~newDateTimePicker(IDC_ORDFORM_DATE);
  orderDate~setFormat("MMM dd',' yyyy")
  today = .DateTime~today
  maxOrderDate = today~addYears(1)
  orderDate~setRange(.array~of(today,maxOrderDate))
```

First the format of the edit field is set. Then the ooRexx **DateTime** class is used to set the allowable range of dates that the user can enter: the range is from "today" (i.e. the day on which the dialog is used) to a year from "today".

7.7. Completing the Application

At this point, there is more to do to complete the application. For example, generic function such as that found in the List components could be moved to a superclass; the three view superclasses could be made into a single mixin class; the Order Form needs to provide for data to be entered and stored; SQL could be used for data-on-disk; updates could properly implemented, and it would be nice if the user could use drag-and-drop to enter data into the Order Form. It is planned that some or all of these functions will be addressed in the next version of this Guide.

Dialog-to-Dialog Drag-Drop

8.1. Introduction

Exercise 8 introduces drag-drop - sometimes known as "direct manipulation" - between dialog components. Try opening an Order Form, then open a Customer dialog. Drag the Customer to an Order Form dialog and drop. The customer details are entered automatically. Drag a Product dialog to the OrderForm dialog and the product number is entered. Drag-drop as it appears to the application developer is discussed in the section *Direct Manipulation*, and further detail is provided in *Appendix E, Direct Manipulation*.

However, Exercise 8 also introduces a number of other things. Although not apparent to the user, substantial changes have been made to the MVF class structure. In particular, the unfortunate duplication of code in the three View superclasses of previous exercises is removed. The new structure is discussed in *Refactoring the MVF*, while the consequential changes at the application level are described in *Using the MVF*.

You may have noticed in Exercise 7 that the process hangs when you close the Order Management dialog without first closing all Order Form dialogs. This problem, and its solution, is addressed in *Event Management*.

Also, some enhancements to the *Section 8.6, "The Order Form"* function are briefly described.

Finally, the last section - *Section 8.7, "To Be Continued"* gives a brief preview of some of the function planned for the next version of this Guide.

8.2. Direct Manipulation

Direct manipulation (or drag-drop) is the use of the mouse to drag a small mouse icon representing a dialog around the screen and drop it on other dialogs, resulting in sopme form of comunication between the two dialogs. To try this out, first surface a Customer dialog (e.g. by double-clicking on an item in the Customer List dialog). Second, create an Order Form dialog by double-clicking on the OrderForm icon in the Order Manager dialog. Now place the mouse cursor over the Customer dialog then press and hold the "primary" button (usually the left mouse button). [1] Start dragging - you'll see the mouse cursor change to a no-entry (or "no-drop") symbol. Drag it over the OrderForm. The mouse cursor changes to a Customer symbol, meaning a drop is allowed. Now release the mouse button - that it, "drop" the Customer onto the Order Form. The customer details appear on the Order Form.

Behind this operation there is some quite complex code (discussed in *Appendix E, Direct Manipulation*), implemented by a Drag-Drop Framework consisting of code within the `View` superclass and a new "manager" object called the "Drag Manager" (`DragMgr.rex` in the `Exercise08\Support`) folder). The objective of the Drag-Drop Framework is to make drag/drop as simple as possible fort he application developer. Thus the code within application components is very simple. In the drag-drop context, there are two kinds of component: a "source" dialog that is dragged, and a "target" dialog that is dragged over and/or dropped upon (dragging more than one component is not supported in Exercise 8). Note that, if it makes sense, a dialog may be both a source and a target. The code required to participate in drag-drop is as follows:

- *A Source Dialog*

---

[1]The "primary" button defaults to being the left-hand button, but this can be changed in the Windows Mouse Properties so that it's the right-hand button.

To enable support for dragging from a source dialog, simply provide the following superclass invocation when the dialog is started (typically in the **initDialog** method):

```
r = self~dmSetAsSource:super("<cursor file name>")
```

The cursor file (with file extension "cur") must be given with its path. For example, the cursor file name in **CustomerView** is **Customer\bmp\Customer.cur**. A cursor file can be created using an appropriate tool.[2]

- *A Target Dialog*

  To enable a dialog to accept a drop, simply provide the following superclass invocation when the dialog is started (typically in the **initDialog** method):

  ```
  r = self~dmSetAsTarget:super(dropArea)
  ```

  The single optional parameter defines the area within the dialog onto which a drop is allowed. If omitted, the drop area defaults to the dialog window less 10 on each side.

  In a target dialog, two things have to be done. First, when being dragged over, decide whether a drop is acceptable, so that the mouse cursor can be set (by the Drag/Drop Framework) as either no-drop or ok-to-drop. Second, if acceptable, accept the drop if it happens (the user might cancel the operation by pressing the Escape key, or may carry on dragging over one dialog to another).

  - **Drop acceptable?**

    When the Drag/Drop Framework detects that the mouse is over a dialog other than the source dialog, it checks whether a drop is acceptable by sending a **dmQueryDrop** message to the class object of the target's model component. This message has a single parameter - the name of the source model's class. For example, a drop on an Order Form sends the **dmQueryDrop** message to **.OrderFormModel**, which accepts a drop of a Customer as follows:

    ```
    ::METHOD dmQueryDrop CLASS PUBLIC
       use arg sourceClassName
       if sourceClassName = "CUSTOMERMODEL" then return .true
       else return .false
    ```

    Returning **.true** changes the mouse cursor to the "drop ok" icon; returning **.false** changes the mouse cursor to the "no entry" or "no drop allowed" cursor (the code for doing this is in the Drag/Drop Framework).

    Why ask the class object and not the View instance if a drop is OK? Well, in the general case, a user might want to drop on some visible representation of a target such as an item in a ListView. To get to the right item, the mouse cursor could be dragged over many other items. If each is asked if a drop is allowed, then each must first be instantiated, and this could mean many database accesses. Much better to check with an object that's already instantiated - such as the class object (class objects are instantiated when the program starts). Note, however, that in Exercise 8 only instantiated and visible dialogs can be drag-drop targets.

  - **Drop happens**

---

[2]One such tool is *GConvert* [http://www.gdgsoft.com/gconvert/index.aspx]. However, there are others, and mention of this particular product should not be interpreted as a preference.

If a drop is acceptable, and if the user then releases the mouse button, then the view instance receives a **dmDrop** message from the Drag/Drop Framework. (In **OrderFormView.rex** it is the main dialog rather than one of the control dialogs that receives this message.) This message has two parameters: the source dialog's model, and the source dialog. The target's **dmDrop** method typically asks the source's model object for data. For example, the drop method in **OrderFormView** is as follows:

```
::METHOD dmDrop PUBLIC
   expose cd1 cd2
   use strict arg sourceModel, sourceDlg
   parse var sourceModel . modelName
   select
     when modelName =  "CUSTOMERMODEL" then do
       cd1~getCustomer(sourceModel); return .true; end
     when modelName = "PRODUCTMODEL" then do
       cd2~getProduct(sourceModel); return .true; end
   end
   return .false
```

Note that the variables **cd1** and **cd2** are the Customer Details and the Product Details control dialogs respectively (see *Section 8.6, "The Order Form"*). The method **getCustomerData** in the Customer Details control dialog invokes **query** on the source's model instance - in this case **CustomerModel**. The customer details part of the Order Form is then populated with the customer data received. Similarly for product details.

8.3. Refactoring the MVF

Exercise 8 introduces a new superclass for components called **Component**, which becomes part of the MVF. **Component** handles event management on behalf of subclasses. In addition, the duplicate View superclasses (**RcView**, **ResView**, and **UdView**) are replaced by a single **View** superclass. Ideally, and using parts of the Customer business component as an example, the desired class structure would be as follows (where a blue border indicates an application component, red an ooDialog class, green an MVF class, and black an ooRexx-provided class):
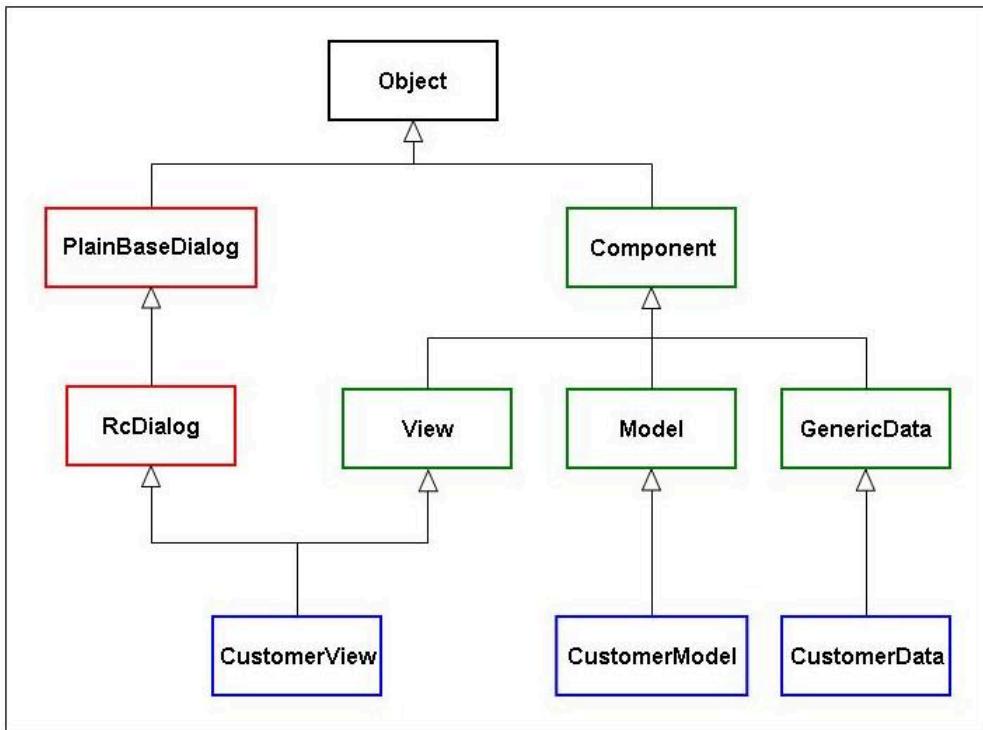
Figure 8.1. Exercise08 Notional Class Structure

However, this structure requires multiple inheritance: each "view" component (such as **CustomerView**) subclasses two superclasses - **RcDialog** and **View**. But this form of multiple inheritance is not supported by ooRexx. However, ooRexx *does* provide "mixin" classes. A mixin class is a class that is "mixed into" the single-inheritance class hierarchy. It's a very useful way to provide more than one effective superclass. Technically, making a class a mixin adds the ability to use the class in the "inherit" option of another class lower down the class hierarchy. In all other respects the mixin is a normal class.

In Exercise 8, the ooRexx mixin facility is used to provide a single "View" mixin class. However, we also want **View** effectively to be a subclass of **Component**. The solution used here is to make **Component** a mixin as well (but note that there may well be other ways of achieving the objective). As long as **View** and **Component** do not have the same method names, then all will be well. The resulting class hierarchy is as follows (with **ProductView** added to illustrate an additional ooDialog superclass):
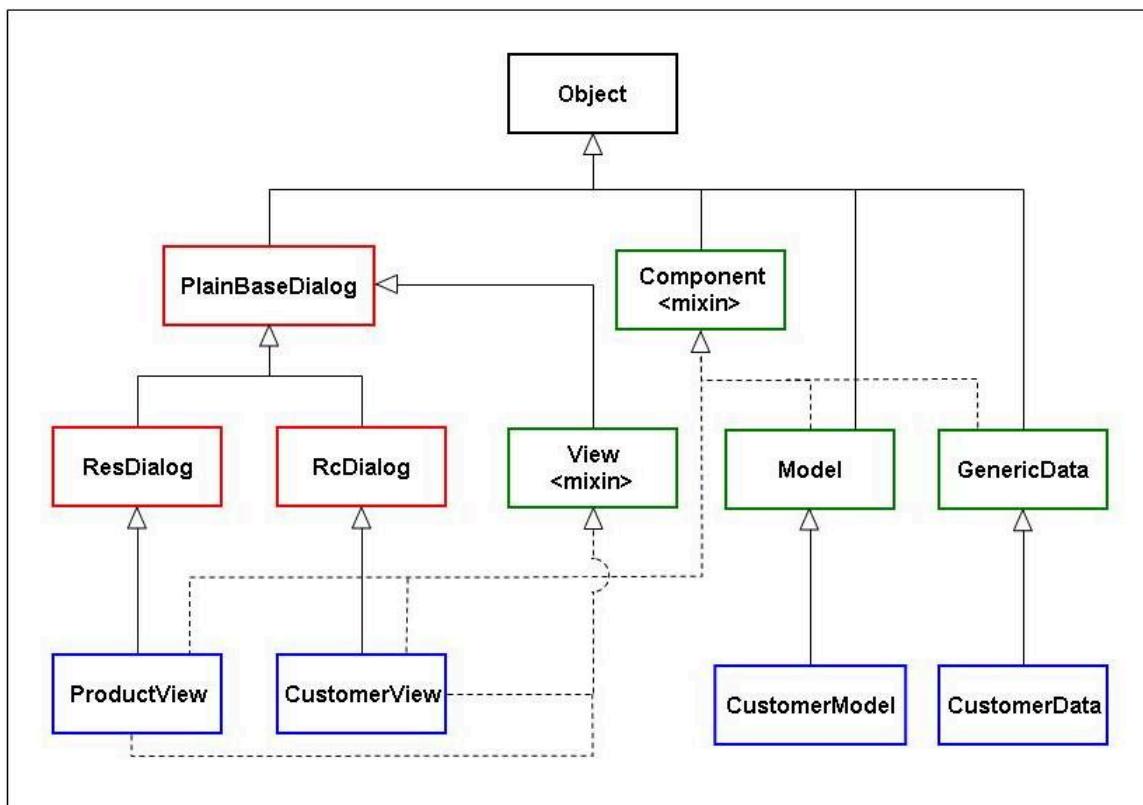
Figure 8.2. Exercise08 Class Structure

Solid lines are normal "subclass" lines, while dotted lines are mixin "inherits". The ooRexx class definitions required for the **Customer** business component to operate with the two mixins is as follows (with unnatural spacing to show differences more clearly):

```
::CLASS Component                              PUBLIC MIXINCLASS Object

::CLASS View                                   PUBLIC MIXINCLASS PlainBaseDialog

::CLASS Model          SUBCLASS Object     PUBLIC INHERIT Component

::CLASS GenericFile    SUBCLASS Object     PUBLIC INHERIT Component

::CLASS CustomerView   SUBCLASS RcDialog   PUBLIC INHERIT Component View

::CLASS CustomerModel SUBCLASS Model       PUBLIC

::CLASS CustomerData  SUBCLASS GenericFile PUBLIC
```

In summary, to comply with MVF, a View application component "inherits" both **View** and **Component**, while a Model or Data component inherits only **Component**.

8.4. Using the MVF

For the application developer, using the revised MVF is very simple, as follows.

For *View* components such as **CustomerView.rex**:

- Add to the start of the *.rex file:

```
::REQUIRES "Support\View.rex"
 ::REQUIRES "Support\Component.rex"
```

- Sublass from **RcView**, **ResView**, or **UdView** as appropriate.

- Add to the end of the **::CLASS** statement "**INHERIT View Component**".

- In the **init** method, immediately after **forward class (super) continue**", add **self~initView**. This ensures that the View mixin is properly initialized (just supering **init** results in a run-time error).

For *Model* and *Data* components, there's no change; merely ensure there's a

```
::REQUIRES "Support\Model.rex"
```

at the start of any Model program file, and a

```
::REQUIRES "Support\GenericFile.rex"
```

at the start of any Data program file.

8.5. Event Management

If an Order Form is open when the Order Management application is closed, then the process hangs. The reason for this is that the OrderForm's control dialogs must be cleaned up explicitly when the Order Form dialog is closed - else the control dialogs are left in limbo, and the process cannot close. The solution is to tell all Order Form dialogs to close when the user closes the application (that is, when the Order Management dialog is closed). But how should this be done? Somehow, all active Order Form dialogs must be told about the "application close" event so they can clean up the two control dialogs.

One solution might be for the Order Management dialog to tell each Order Form that the application is about to close. Perhaps the Order Management dialog could inquire of the Object Manager as to which Order Form dialogs exist, and then send each of them a "close now" message. However, a more elegant solution - and one that has much wider potential use - is to provide an event management framework.

A simple event management framework (EMF) is introduced in Exercise 8. The object that keeps track of events and who's interested in them is the Event Manager (the file **EventMgr.rex** in the **Exercise08\Support** folder). The code that "talks" directly to the Event Manager is in the **Component** mixin. Application-level components merely super event requests, and **Component** does the rest.

Note that events handled by the Event Manager must not be confused with events generated by ooDialog, such as menu selection events.

The EMF works like this. First, a dialog component (such as the Order Form) decides that it's interested in some event - say "AppClosing". When this event occurs, it wants to be sent a message so that it can take appropriate action. To express its interest, the component supers (e.g. in its **activate** method) a **registerInterest** message. This has two parameters - the event in which it's interested and its object id (i.e. **self**). For example, the "register interest" invocation in **OrderFormView** is:

```
self~registerInterest("appClosing", self)
```

This message is handled by the **Component** mixin, which forwards the message to the Event Manager, which then adds the event to its directory of events. In this directory, each index is the event name, and each item is an array of objects that have expressed interest in that event. Some time later, when the Order Manager (**OrderMgrView** - which in Exercise 8 mixes-in and so inherits from both **ResizingAdmin** *and* **Component**) is closed, it asks for the event "appClosing" to be triggered:

```
self~triggerEvent("appClosing")
```

The **Component** mixin sends this to the Event Manager which then sends a **notify** message to each component that has registered interest in this event. The Order Form provides a method to capture this event and close its control dialogs:

```
::METHOD notify PUBLIC
    use strict arg event
    if event = "AppClosing" then self~closeControlDialogs
```

Thus each Order Form that's open when the app closes receives a notification so that it can close its control dialogs.

Note that this simple event management framework is fairly generic, and can be used for any event by any component. The only constraint is that event names should be unique across an application. For example, triggering an "AppClosing" event to signal that a dialog is opening is likely to have unfortunate consequences (although the Event Manager will be quite happy).

Finally, note that the Event Manager has a **list** method which lists all registered events on the console.

8.6. The Order Form

The Order Form dialog is now partly semi-functional. Aside from the enhancements already mentioned (drag/drop and closing properly on app close), several new capabilities are added:

1. Customer details can be entered using the keyboard (as well as with drag/drop). Type a customer number (e.g. "CU0003") into the "Customer Number" field on the "Customer Details" tab. The "Find Customer" button is enabled. Click on the button, and the Customer details are entered on the form. Accessing customer details is done by the following code (excluding error-handling code):

    ```
    ::METHOD findCustomer UNGUARDED
       expose ecCustNum objectMgr
       custNo = ecCustNum~getLine(1)
       idCust = objectMgr~getComponentId("CustomerModel",custNo)
       dirCustData = idCust~query
       self~setCustomer(dirCustData)
    ```

2. Products can also be entered using the keyboard. Click on the "Order Lines" tab, and enter a product number (e.g. "CU003") into the "Product No." field. Then enter a quantity in the "Quantity" field. Finally, click the "Add Order Line" button. A line item for that product is entered into the Order Details list view.

3. Third, totals are added up and discounts applied. However, removing a line item does not, in this version, decrement the order totals (a serious omission in a real system!).

4. Finally, try double-clicking on an Order Line. A product dialog for the line item is surfaced. This uses the same code (mutatis mutandis) as that for surfacing a Customer dilaog.

Capabilities missing in this exercise include saving the Order to disk when it's complete, allowing use of the enter key for the "Find Customer" and "Add Order Line" buttons, and ensuring that the totals are always correct. In addition, the calculation of totals should be done in the Model component, not in the View. This is because a major role of a model component is to implement business rules. The results of applying such rules is then made visible by the View component. In summary, the Model should do the "business" part while the View should make the business visible and provide for new info to be entered.

8.7. To Be Continued

In the next version of this Guide, the development of the Order Management application will be continued. In particular, it is planned to move the text data files to an *SQLite*[3] database, which will provide for data to be updated as well as read. SQLite will be accessed using *ooSQLite*[4] - the ooRexx language interface to SQLite.

A second function planned is drag/drop of a component from a list view such as **CustomerListView** to an Order Form. The Order Form will see this as a drop of a specific Customer rather than of a Customer List.

You may have noticed that most of the code in the List View components (e.g. **CustomerListView.rex**) is identical aside from some variable names. This cries out for a ListView superclass. Exercise 9 will introduce just such a superclass, and the individual list views will merely configure the view and provide a call-back method for any data needing specific attention - for example formatting a date field according to country-specific usage.

Finally, aside from other enhancements, it is planned to show how an application consisting of multiple dialogs can be "packaged" so that the many folders and **.rex** files are simplified into a much smaller number of files.

---

[3] http://www.sqlite.org/index.html
[4] http://sourceforge.net/projects/oorexx/files/ooSQLite/

Appendix A. Dialog Attributes and AutoDetection

Since the early days of ooDialog, a dialog's controls (listboxes, edit controls, radio buttons, etc.) have been treated as attributes of the dialog object. An ooRexx program could set the attributes' data values in an ooRexx compound symbol (aka stem variable or compound variable) before the dialog was created. Then, when the dialog was created, those values were automatically passed to the controls in the underlying Windows dialog and so were visible to the user. Data could then be entered or modified by the user. When the user closed the dialog, the data (whether changed or unchanged) was automatically communicated from the underlying Windows dialog to the ooRexx dialog and placed in the compound symbol, after which the dialog closed and returned control to the next ooRexx statement in the program. The data was then available to the ooRexx programmer in the same compound symbol.

This function is still supported by ooDialog. The compound symbol is often referred to as "dialog data", and the process of automatically moving data between the ooRexx dialog and the underlying Windows constructs is called "automatic data detection" or "auto detection" for short. The aim of this appendix is to illustrate, through a simple example, how automatic data detection is coded. The example, **ASimpleDialog.rex**, can be found in the **Exercise04\Extras\DlgData** folder. When executed, the dialog looks like this:

Figure A.1. A Simple Dialog

The program does the following:
- Sets the value "It's a fine day today." in a dialog attribute value in an edit control.

- Creates and then displays the dialog.

- When the dialog is closed, retrieves the attribute value as modified (or not) by the user.

- Sets up an appropriate message (this is straight ooRexx and forms the bulk of the program!).

- Displays the message in a message box.

The ooDialog content of the program is very simple, and is as follows:

```
-- (1) Set text in the edit control:
statement = "It's a fine day today."
dlgData.IDC_EDIT1 = statement

-- (2a) Create the dialog defined by the .rc file:
dlg = .ASimpleDialog~new("ASimpleDialog.rc", IDD_DIALOG1, dlgData., "ASimpleDialog.h" )
```

```
-- (2b) Display the dialog:
ret = dlg~execute("SHOWTOP", IDI_DLG_OOREXX)

-- (3) When the dialog is closed, and if the user pressed OK, then retrieve
--      the data provided by the user:
if ret == 1 then do   -- if the user pressed OK
  statement2 = dlgData.1002  -- get data from the edit control
  agree = dlgData.IDC_RADIO1  -- get the state of the radio buttons:
  disagree = dlgData.1004

-- (4) Set up the appropriate message to display:

      /* a number of lines of ooRexx code */

-- (5) Display a message to respond to the user's choices:
ret = MessageDialog(msg, 0, title, 'OK', icon, 'TOPMOST')

::requires "ooDialog.cls"

::CLASS ASimpleDialog SUBCLASS RcDialog
```

As illustrated by section (1) of the code, data is first set up in the stem *dlgData.* by the statement: **dlgData.IDC_EDIT1 = statement**. Note that the name *dlgData.* is not reserved - it could be any name. When the dialog is created, the text "It's a fine day today." is automatically placed in the edit control identified by the symbolic ID **IDC_EDIT1**.

Section (2) creates and displays the dialog. Only two statements are required to do this:

```
      dlg = .ASimpleDialog~new("ASimpleDialog.rc", IDD_DIALOG1, dlgData.,"ASimpleDialog.h" )
      ret = dlg~execute("SHOWTOP", IDI_DLG_OOREXX)
```

The first parameter of the dialog creation statement is the name of the .rc file, and the second the symbolic ID of the dialog as defined in that file. The third parameter - *dlgData.* - is the stem variable that contains the data - that is the attribute values - to be placed in the dialog's controls. Finally, the fourth parameter is the header file, which can be omitted if the statement **.application~useGlobalConstDir("O", "ASimpleDialog.h")** is placed at the start of the program.

The class **ASimpleDialog** is defined at the end of the program. Note the extreme simplicity of coding a simple form-filling dialog class. No methods are needed - the dialog consists of a single **::CLASS** statement. The superclass, **RcDialog**, provides all the function needed.

Section (3) of the code shows how data is retrieved from the controls via *dlgData* after the dialog has been closed by the user. Note that a mixture of numeric and symbolic IDs can be used, as illustrated by statements such as *disagree = dlgData.1004*. Indeed, a control can be referenced by its symbolic ID in one place and by its numeric ID in another, as illustrated by the use of both **IDC_EDIT1** and **1002** to refer to the edit control. However, from a program comprehensibility point of view, it is not good practice to mix symbolic and numeric resource IDs in the same program. Further, it is generally held that using only symbolic IDs is best practice.

Section (4) of the code analyzes the user's input. Finally, section (5) displays a message box to inform the user of the results.

In order to illustrate the same function using **.ResDialog** the program **ASimpleDialog2.rex** is included in the **Exercise04\Extras\DlgData** folder together with its *.dll file.

Finally, when desired, there are two ways to turn auto detection off (by default it is turned on). First, by the Application Manager (see the ooDialog Reference), and second programmatically by intercepting the *initAutoDetection* message. This is automatically sent to the dialog by the ooDialog framework

when the dialog is instantiated. To turn auto detection off, just invoke *noAutoDetection* on *self*. Try adding the following method to **ASimpleDialog**

```
::CLASS ASimpleDialog SUBCLASS RcDialog
  ::method initAutoDetection
    self~noAutoDetection
```

When the program is run, the edit control is blank. On pressing "OK", the dialog closes and an error is reported on the console. The error occurs because the radio button "data" (i.e. a boolean) is not returned, and so an "if" statement fails because it's expecting a boolean value to be tested.

Appendix B. Testing Popups in Stand-Alone Mode

This appendix discusses two separate aspects of "popup" dialogs. The first aspect is testing popped-up dialogs in stand-alone mode - that is, without having to run the "parent" dialog from which the popped-up dialog is launched. The second aspect is the issue of how a popped-up dialog can be visually offset from its "parent" so that it does not obscure the parent.

B.1. Stand-Alone Testing

Consider four dialogs called Parent, Child, Grandchild, and GreatGrandChild. Parent is the "application" - the dialog that opens first, and from which other dialogs are directly or indirectly surfaced. Parent can thus be called the "root" dialog, and is designed to run in "standalone" mode - that is, it is not surfaced by some other dialog. It pops up Child dialogs, each of which may pop-up Grandchild dialogs, which in turn may pop-up GreatGrandchild dialogs, and so on.

When testing an application, there is often a need to test an individual dialog which, in the application, is a "child" that's invoked by a "parent" dialog which issues *self~popUpAsChild(...)* - rather than *self~execute(...)* - to surface the child dialog. In addition, popping-up requires the parent dialog to be specified as a method argument: *self~popUpAsChild(parent,...)*.

In the Order Management application, the **OrderMgrView** class is the parent for all child dialogs. The reason for using popupAsChild is so that, for example, a CustomerList can be closed without automatically closing any Customer dialogs that might have been opened from it.

Now, using the parent dialog (Order Manager in our case) as some sort of test-harness that will eventually surface the child dialog to be tested can be time-consuming and irritating. However, if a child dialog is started without the parent first being run, it must still be able to invoke subordinate dialogs in the same way as if it were running as part of the full application. Thus there is a need to enable individual child dialogs to be tested in "stand-alone" mode, without using the parent dialog just to surface them, but invoking other "subordinate" dialogs as if it was not being run stand-alone. In addition, a stand-alone test of what is normally a "child" dialog will require the child to act as the parent of any "grandchild" dialogs that it invokes, which means that the child must pass its own id to the grandchild instead of passing the parent's.

One approach to resolving this problem is to have two versions of each child dialog - one using *~execute(...)* and one using *~popupAsChild(...)*. This results in two code bases for each dialog - which can quickly get out of sync. Not the best idea.

An arguably better solution is to build each dialog so it can be run either individually (stand-alone) or within the application. The file **Popups.rex** in the folder **Exercise06\Extras\Popups** shows a way of doing this, using dialogs that are as simple as possible.

The rules illustrated by the code in **Popups.rex** are as follows, assuming an application consisting of a single Parent dialog that invokes one or more Child dialogs, each of which may invoke one or more Grandchild dialogs, each of which may invoke one or more GreatGrandChild dialogs. The child and grandchild dialogs are "intermediate" dialogs. The GreatGrandChild dialog is a "leaf" dialog - that is, it does not invoke any other dialog (except of course those integral to its own functioning such as an About box or a data entry sub-dialog).

```
The Parent Dialog:

  (1) Is invoked from a Startup script with:
        .ParentDialog~newInstance
```

```
    (2) Provides an event handling method that sufaces a Child dialog:

          ::METHOD popup UNGUARDED
            ...
            .ChildDialog~newInstance(self)
            ...

 An Intermediate Dialog:

   (1) For stand-alone testing is invoked from a Startup script with:
          .AnIntermediateDlg~newInstance("SA")

   (2) Provides the following methods (among others):

        ::METHOD newInstance CLASS
          use arg rootDlg
          ...
          dlg = self~new
          dlg~activate(rootDlg)

        ::METHOD activate UNGUARDED
          expose rootDlg
          use arg rootDlg
          ...
          if rootDlg = "SA" then do  -- If standalone operation required
            rootDlg = self    -- To pass on to subordinates
            self~execute("SHOWTOP")
          end
          else self~popupAsChild(rootDlg, "SHOWTOP")

        ::METHOD popup UNGUARDED      -- An event handler method
          expose rootDlg
          .ASubordinateDlg~newInstance(rootDlg)

 A Leaf Dialog:

   (1) For standalone testing is invoked from a Startup script with:
          .ALeafDlg~newInstance("SA")

   (2) Provides the following methods (among others):

        ::METHOD newInstance CLASS
          use arg rootDlg
          ...
          dlg = self~new
          dlg~activate(rootDlg)

        ::METHOD activate
          use arg rootDlg
          if rootDlg = "SA" then self~execute("SHOWTOP")
          else self~popupAsChild(rootDlg, "SHOWTOP")
```

Try running the **Popups.rex** program without any parameters. Note that as each "junior" dialog is created (by pressing the pushbutton in the "senior" dialog) it completely obscures its parent. This is because all dialogs are coded to surface in the center of the screen (by the style "**CENTER** in the *create* method), and second all dialogs are the same size. The next section illustrates a useful way to offset the subordinate dialogs so that at least some part of the senior dialog is still visible.

B.2. Visual Offsetting

The program **OffsetPopups.rex** in the **exercises\Exercise06\Extras\Popups** folder is a copy of **Popups.rex** with added code to handle dialog offsetting (comments show where statements have been added or modified). If no parameters are provided, the offset defaults to zero and the

behavior is identical to that of **Popups.rex**. Try running the program with an offset of 100 by entering **OffsetPopups 100** on a command prompt. You'll see that popped-up dialogs are offset from the dialogs from which they're popped up, and do not now obscure them. Entering **OffsetPopups ?** provides help.

In **OffsetPopups.rex**, all classes are subclassed from a **View** class (itself subclassed from **UserDialog**) which has one class attribute and two methods, *getPopupPos* and *offset*, as follows:

```
::CLASS View SUBCLASS UserDialog

  ::ATTRIBUTE offsetAmount CLASS PUBLIC

  ::METHOD getPopupPos
    popupPos = self~getRealpos
    offset = .View~offsetAmount
    popupPos~incr(offset,offset)
    return popupPos

  ::METHOD offset
    use arg dlgPos
    self~moveTo(dlgPos, 'SHOWWINDOW')
    self~ensureVisible()
```

The class attribute *offsetAmount* is set at the start of the program, and defines the amount of space by which to offset a junior dialog. The term "junior dialog" in this section refers to a dialog that is popped up by a "senior dialog", and in the sample code refers to any of the classes **Child**, **GrandChild**, and **GreatGrandChild**. "Senior dialog" refers to the dialog that pops up a junior dialog, and in the sample code refers to any of the classes **Parent**, **Child**, and **GrandChild**

In brief, the method *getPopupPos* is used by a senior dialog to establish where it wants a junior dialog to pop up. The junior dialog then uses the *offset* method to (a) move itself to the desired position, and (b) to ensure that it is wholly visible on the screen and not partly off the screen.

In detail:
- ***getPopupPos*** - This method is used by the senior dialog to establish where on the screen the junior dialog is to appear (relative to the senior dialog). The first statement *popupPos = self~getRealPos* gets the position of the senior dialog as a point object (see ooDialog Reference) whose attributes are the point's *x* and *y* screen coordinates (that is, the top-left corner of the dialog). The point object is assigned to *popupPos*.

- The second statement, *offset = .View~offsetAmount* gets the offset amount stored in the class attribute.

- Then the statement *popupPos~incr(offset,offset)* increments each coordinate of the *popupPos* object by the amount defined by *offset*. That is, *popupPos* is now the desired new position of the junior dialog.

- Finally, the desired junior dialog's position is returned, and the senior dialog then passes it to the junior dialog when the latter is created via its *newInstance* method.

- ***offset*** - This method is invoked from the the junior dialog's *initDialog* method in order to move itself to the position (*dlgPos*) defined by the senior dialog. The first statement (*self~moveTo(dlgPos,'SHOWWINDOW')*) moves the dialog. However, if the senior dialog is near the bottom or right-hand edge of the screen, the junior dialog could surface half-off the screen in the correct offset position. But half-off the screen is not particularly friendly.So...

- ... the last statement (*self~ensureVisible()*) ensures that the current dialog is wholly visible. However, because it's so fast, you don't see this re-positioning. To see the re-positioning, insert

call sysSleep(2) just before the last statement, run the program, and move the parent dialog to the bottom of the screen. Then popup the child dialog. It appears half-off the screen, then after two seconds it snaps up to a wholly-visible position. Neat.

Appendix C. Dialog Creation Methods

This appendix provides a programmer's aide-memoire for the methods required to create and set up a dialog using one of the more usual superclasses - **UserDialog**, **RcDialog** or **ResDialog**. Menu creation is included even though this is technically quite separate from dialog creation, and does not have to be done in the *init* method.

The following table shows, for each of the three main dialog types, the method invocations that the programmer must code and the methods (invoked by ooDialog-provided superclasses as part of the dialog creation framework) that the programmer must provide.

Table C.1. Dialog Creation - Method Sequences

Methods / ~Invocations	UserDialog	RcDialog	ResDialog	Comment
.Dlg~new(...)	Yes.	Yes.	Yes.	Class Method
init	Yes.	Yes. Create Menubar *.ScriptMenuBar~new*	Yes.	Must be passed to the superclass using *forward class (super) continue*
defineDialog	Yes. Programmer creates the Dialog Template using *self~create...(...)*	Optional. Dialog Template is defined by the *.rc file, but additional controls (or menu items) can be added here.	No. Not invoked - Dialog Template is defined by the *.dll file - so controls or menu items cannot be added.	Called by super's *init* method (but not for **ResDialog**s) Purpose: create the Dialog Template.
dlg~execute	Yes.	Yes.	Yes.	Creates the Underlying Dialog based on the Dialog Template
initDialog	Create "proxies" for controls and initialize them using *ctl=self~new...(...)* Create Menubar *.BinaryMenuBar ~new()*	Create "proxies" for controls and initialize them using *ctl=self~new...(...)* Attach Menubar using *~attachTo(self)*	Create "proxies" for controls and initialize them using *ctl=self~new...(...)* Create Menubar *.BinaryMenuBar ~new()*	Called automatically after *~execute* is invoked.

First, as for all ooRexx objects, the *~new* method creates an instance of a dialog class, and the *init* method is then invoked on the new instance, which must invoke the superclass' *init* using *forward class (super) continue*. For **RcDialog** subclasses, a menubar could be created in this method, but could also be created later.

The *defineDialog* method (see the ooDialog Reference) is invoked automatically by the superclass' *init* method. This method provides for the creation of the "Dialog Template" (see the ooDialog Reference) - that is, the layout of controls on the dialog. For a **UserDialog** the dialog template must be created using self~create(...) instructions. For an **RcDialog** the dialog template is normally fully-defined by the *.rc file, but can optionally be enhanced here. However, in the case of **ResDialog**, the dialog template is fully-defined by the *.dll file, and cannot be changed programmatically. Therefore, a *defineDialog* message is not sent to a **ResDialog**.

On exit from the *defineDialog* method, the dialog template is established.

The "underlying dialog" (see the ooDialog Reference) is then created and surfaced (made visible to the user) by invoking the superclass' *execute* method.

The last method in the table - *initDialog* - is provided for the programmer to initialize the various controls, e.g. setting an edit control to its initial data value, or pre-selecting a radio button. This is done by creating "proxies" for those controls that need to be manipulated within the program, typically using *proxy = self~new...(...)* statements. It's worth remembering that the "init" in *initDialog* means "initialize" - not to be confused with the ooRexx *init* method.

Finally, although the creation of a menubar is mentioned in the table, technically it is not part of dialog creation. A menubar can be created any time. However, it can only be attached to a dialog after the underlying dialog is created. Thus the first opportunity to attach a menubar to the dialog is in the *initDialog* method; but it can be done later.

Appendix D. The Model-View Framework

A description of the Model-View Framework (MVF) is given in *MVF Overview*. This appendix provides some further detail on certain aspects of the MVF. By the way, the MVF should really be called "Model-View-Data Framework", since it also encompasses data components; however, historically such frameworks have omitted the term "data" in their names, and here we lazily conform with precedence. The following areas are addressed:

- A brief review of what is meant by "component"

- The classes that make up the MVF

- An "under-the-covers" example of a common MVF operation.

D.1. Components, Files, and Folders

A "component" is one of three kinds: a Model, View, or Data component. A component may have more than one class. For example, the "ProductView" component (in the **Exercise07\Product** folder) consists of three classes: **ProductView**, **AboutDialog**, and **HRSpv**. The first of these is the "main" class in the component (sometimes called the "focus class"), the other two being "subsidiary" classes (sometimes called "support" classes).   The **ProductView** component, together with the **ProductModel**, **ProductListModel**, **ProductListView**, and **ProductData** components make up the "Product Business Component". Thus a "business component" consists of one or more components, each of which has a "main" class and naught or more subsidiary classes. Thus (by design) a Business Component is a collection of components that implements all and only a significant business concept.[1] Finally, a business component is packaged within a folder that bears its name, the individual components being packaged in .rex files. Thus the Product business component is the folder **Product** which contains a number of files, most of them executables such as **ProductModelsData.rex** which contains the **ProductModel**, **ProductListModel**, and **ProductData** components.

D.2. MVF Classes

The Model-View Framework (MVF) manages the components that are assembled to make up an application. There are two main parts to the MVF:
- Two management classes called the "Object Manager" (**ObjectMgr.rex**) and the "View Manager" (**ViewMgr.rex**)

- A set of superclasses - **Model**, **GenericFile**, and three View superclasses - **RcView.rex**, **ResView.rex**, and **UdView.rex**. The three View superclasses are identical except for their superclasses - **RcDialog**, **ResDialog**, and **UserDialog** respectively.[2]

All MVF classes are in the **Exercises/Support** folder.

---

[1]The definition of a "significant business concept" in the context of components has been addressed in a number of sources, including "Enterprise Service-Oriented Architectures", McGovern, Sims, Jain & Little, Springer 2006.
[2]The three View superclasses should properly be mixins rather than different files, and a future version of the Guide may take this approach.

D.2.1. Management Classes

D.2.1.1. The Object Manager

The Object Manager is at the heart of the Model-View Framework. Its public methods are described in *The Object Manager*. Briefly, the function of the Object Manager is to relieve the programmer from having to be concerned with the various management aspects in a multi-dialog application. Thus the Object Manager, in conjunction with other MVF classes, implements common patterns such as the "get component id" pattern, and the "show a view of a model" pattern. An example of how the **getComponentId** method works is shown below in *Section D.3, "MVF Operations"*.

D.2.2. The View Manager

The View Manager manages common function to do with views (or dialogs) that are not appropriate (from a design point of view) for the View superclasses.

In Exercise 7, View Manager is used only for offsetting dialogs, which is demonstrated by **PersonView** (in the **Exercise07\Extras\Person** folder). First, un-comment the statement "**self~offset:super** in the **initDialog** method. Then launch a Person Model instance using Message Sender to send a **showModel** message with the data, *PersonModel PA150* to the Object Manager. The person dialog appears offset from the Message Sender.

The offsetting is done by the two dialogs involved: the "parent" dialog from which offsetting is to be done, and the "offset" dialog that appears offset from the parent. The sequence of operation is as follows:

```
A. Setup:
   1. ViewMgr sets its 'dlgOffset' attribute in its 'init' method (set to 200
      in Exercise 7).

B. Execution:
   1. The parent dialog (e.g. MessageSender) sets itself as the 'parent' (from which
      offsetting is to be done) by setting ViewMgr's 'parentOffsetDialog' attribute.
   2. The parent launches the 'child' offset dialog by sending 'showModel' to the
      Object Manager.
   3. The offset dialog (e.g. Person View) issues 'self~offset:super' in its
      'initDialog' method.
   4. This is handled by the superclass (RcView for PersonView), which first asks
      ViewMgr for the offset amount (attribute 'dlgOffset'), and then moves the new
      dialog by the offset amount.
```

Note that application components only do steps 1, 2, and 3, the code for which is as follows:

```
1. Parent - <some method>:  .local~my.ViewMgr~parentOffsetDlg = self
2.                          r = .local~my.objectMgr~showModel(childModel, -
                                childInstance, rootDlg)
3. Child -  initDialog:     self~offset:super
```

In summary, using the MVF, it takes very little effort by the application programmer to perform dialog offsetting.

D.2.3. Component Superclasses

Superclasses provided for application components are: **Model**, **GenericFile**, and a set of view (or dialog) superclasses: **RcView**, **ResView**, and **UdView**. These are described in *MVF Overview*.

D.3. MVF Operations

The following shows the sequence of operations that takes place when a `getComponentId` operation is invoked on the Object Manager. For this example, it is assumed that no instance exists at the start of the operation.

```
(1)   X --> invokes "getComponentId('PersonModel','PA150') on ObjectMgr.
 (2)     <-- ObjectMgr, if the object ref for "PersonModel PA150" exists, returns it to X.
 (3)         Else ObjMgr invokes "newInstance" (with the instance name "PA150" as an
              argument) on .PersonModel.
 (4)         --> .PersonModel forwards to its superclass (.Model) which, in order to get
                 this instance's data, invokes "getComponentId('PersonData','The')" on
                 ObjectMgr.
 (5)             --> ObjectMgr, if "PersonData The" exists,
 (6)             <-- returns its object ref to .Model.
 (7)                 Else ObjectMgr invokes "newInstance", with the instance name "The",
                     on .PersonData.
 (8)                 --> .PersonData creates an instance of itself ("Person Data The") and
                         the instance's 'init' method uses its superclass to read its file
                         from disk.
 (9)                 <-- The object ref for "PersonData The" is returned to ObjectMgr.
(10)             <-- ObjectMgr stores the object ref (and name) for "PersonData The",
                     and returns it to .Model.
(11)             .Model then invokes "query", with the instance name ("PA150"),
                 on "PersonData The".
(12)             --> "PersonData The" uses its superclass ("GenericFile") to read
                     the data from disk, and
(13)             <-- returns the instance data (as a directory) to .Model.
(14)             .Model issues "self~new" with the instance data as a parameter,
                 so creating a PersonModel instance.
(15)             --> PersonModel's init method saves the data and
(16)             <-- returns to .Model
(17)         <-- .Model returns the id to ObjectMgr.
(18) X <-- ObjectMgr returns the id to X.
```

The Object Manager maintains a table of all active component instances. They can be listed on the console by using the Message sender to send a **list** message to `ObjectMgr The`. See *The Object Manager* for a description of the Object Manager's public methods.

D.4. Class Naming Constraints

The software architecture used for the Order Management application defines several different *kinds* of component. These are "Named", "Singleton", "Anonymous", and "Form", and are fully described in *Section 7.3.1, "Kinds of Component"*. The MVF must be aware of these differences, since the fine detail of its operations sometimes depend on the kind of class being dealt with. For example, one difference between showing a view of a Customer and showing a view of a Customer List is that the former requires one record to be returned from the Customer Data component, while the latter requires the complete file to be returned. In Exercise07, the MVF discovers what kind of component is being dealt with by specific words in the class name ("List", "Form") or by the instance name being "A" or "a". A better way of specifying the kinds of component could be through configuration, where the folder for each component would contain a configuration file which the MVF would read.

D.5. The Requires List

An oorexx class that is instantiated from another file must be specified in an ooRexx "requires" statement. However, in the MVF, component instantiation (or, to be precise, instantiation of the main class in a component) is handled by `ObjectMgr`. This mean that `ObjectMgr` must provide a `::requires` statement for each component. However, to avoid changing the `ObjectMgr.rex`

file, a separate file (**RequiresList.rex** in the **Exercise07** folder) containing only the desired set of **::requires** statements is used. **ObjectMgr.rex** calls this file as a routine (**call "RequiresList.rex"**) as its first executable statement. In this way, additional components can be added to the application merely by adding their file names to this file. Of course, if a configuration file approach were to be used, then the MVF could generate the appropriate RequiresList file.

Appendix E. Direct Manipulation

This appendix provides an overview of the internals of the direct manipulation or "drag/drop" function provided from Exercise08 onwards. Drag/drop is handled by a framework that relieves the application developer from almost all of the necessary complexities of drag/drop. This framework is implemented by a cooperation between the **View** superclass and a new "manager" class, the Drag Manager (**DragMgr.rex**) both of which are located in the **Exercise08\Support** folder. A description of the application developer's view of drag/drop is provided in Chapter 8, *Direct Manipulation*.

This Appendix discusses the following aspects of the "mechanics" of drag/drop:

- *The Mouse Class*

- *Factoring the Drag/Drop Code*

- *Enabling Drag/Drop*

- *Pickup and Drag*

- *Drop on Target*

E.1. The Mouse Class

Drag/drop is enabled through ooDialog's **Mouse** class. When a window (a dialog or a control such as a List View) is to be the source of a drag operation, then an instance of the **Mouse** class must be instantiated (typically in the dialog's **initDialog** method) and associated with the window. For example:

```
mouse = .Mouse~new(self)
```

Mouse events are then connected to the mouse instance. For example, here's the code for connecting a left-button-down (aka "start drag" or "pickup") event:

```
mouse~connectEvent('LBUTTONDOWN', dmOnLBdown)
```

This defines the method **dmOnLBdown** as the event handler for a pickup. Other drag/drop events include mouse movement (dragging with the left button down) and a drop (left button up).

When the user presses and holds the left mouse button then a pickup event is detected, and the "Left Button Down" event handler method is invoked. In this method, the mouse instance is "captured" as follows:

```
if mouse~dragDetect(...) then do
    ...
    mouse~capture()
    ...
  end
```

The **dragDetect** method returns **.true** if the user has indeed started dragging. Note that all drag/drop events are sent to the object in which the mouse was instantiated - and each mouse instance is affiliated with a specific window object. For clarity, in the above code various statements to do with setting the drag cursor and establishing the initial state of the drag operation (e.g. the "we're not over a target" state) are omitted.

E.2. Factoring the Drag/Drop Code

Drag/drop code is not particularly simple. So the question is, can the code be factored so that much of the detail can be delegated so that the application developer finds drag/drop easy to implement? To answer this, we start by defining the things that *must* be done by the application developer; everything else can then be delegated to a superclass or other "manager" object. This minimum set, for simple drag/drop operations between dialogs, is as follows:

- Define the dialog window (or control window within a dialog) that is to be the drag/drop "source" (i.e. that it can be picked up). This includes:
  - The area within the window from which a pickup is valid.

  - The cursor icon to be used in dragging from a source window.

- Define the dialog window (or a control window within a dialog) that will accept a drop. This includes the area within the window in which a drop is valid.

Definition of both source and target dialogs is handled by an invocation of the superclass in an application dialog's **initDialog** method (as discussed in the next section). Everything else is handled by the **View** superclass (the superclass of all "view" components and located in the **Support** folder).

Finally, **View** delegates most of the detailed handling of a drag/drop operation to the Drag Manager (**DragMgr.rex** - also located in the **Support** folder).

E.3. Enabling Drag/Drop

In Exercise08, all mouse events are registered in the **View** superclass. However, only an application-level dialog can define itself as a potential source for a drag operation. It does this by supering (from its **initDialog** method) a **dmSetAsSource** message (as shown at the top of *Figure E.1, "Drag/Drop Setup"*). A mouse instance is created in the superclass's **dmSetAsSource** method and the pickup, drag, and drop events are connected to **View** methods. Finally, **View** sends a **setSource** message to the Drag Manager informing it that a new source dialog is present (the dialog's mouse instance being included as one of the arguments). The Drag Manager stores this information in a table. When a drag is started, it is the Drag Manager that manages the fine detail of the operation, finding the right mouse instance from its table of source and target windows.

Now, if mouse events are fired during drag/drop, then what object instance receives these events? The answer is the object that instantiated the mouse - that is, the application-level view component - although handling these events is done in the **View** superclass, from where they're passed to the Drag Manager.

The diagram "*Figure E.1, "Drag/Drop Setup"*" shows how components are enabled for either dragging or dropping.
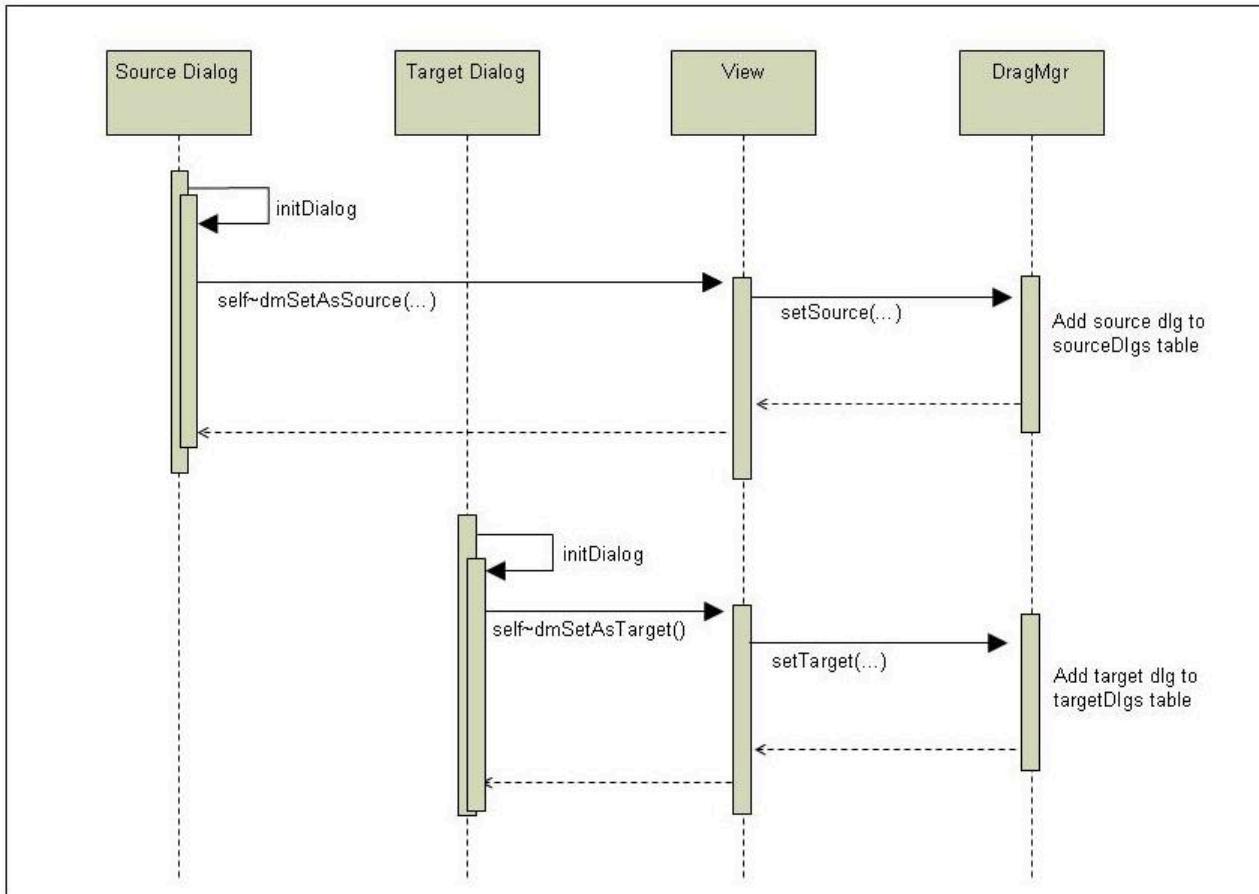

Figure E.1. Drag/Drop Setup

In the diagram, a component view that can be "picked up and dragged" is called a *source dialog*.
 It defines itself as a "source" by supering to **View** a **dmSetAsSource** message (typically from its **initDialog** method). There are three parameters to this message: first (obligatory) is the file name of the cursor to show when dragging (a ***.cur** file), and second (optional) the area of the source dialog from which it's valid to "pick up" - i.e. to start dragging, and the third (optional) is a control window when the pickup is from a specific control in the dialog. If the second parameter is not provided, then the superclass defaults to the dialog area less 10 on each side. If the third argum,ent is not provided, it default to the dialog window.

In its **dmSetAsSource** method, **View** first creates an instance of the **.Mouse** class, and registers three event handling methods - **LBUTTONDOWN**), **MOUSEMOVE** and **LBUTTONUP**. If these events are not defined then drag/drop operations are ignored. It then invokes the **setSource** method of the Drag Manager with five arguments: the source window, the mouse instance, the cursor filename, the source's valid pickup area, and the source dialog. The Drag Manager then adds the source window to its table of sources. Note that if a **dmSetAsSource** message is not supered by a dialog, then no mouse events are handled, and so nothing happens when mouse button 1 is pressed and held.

A view component that will accept a drop is called a *target dialog*. It defines itself as a target by supering a **dmSetAsTarget** message (typically from its **initDialog** method) to its **View** superclass. The only argument is the area in the dialog in which a drop is allowed. If omitted, the default is the dialog client area less 10 on each side. **View** then invokes the Drag Manager's **setTarget** method with three parameters: the dialog id, the dialog's window handle, and area within the dialog within which a drop is allowed.

Finally, note that a view component can be both a source and a target.

E.4. Pickup and Drag

The top part of the following figure shows the pickup and drag operation. The lower half shows a drop.
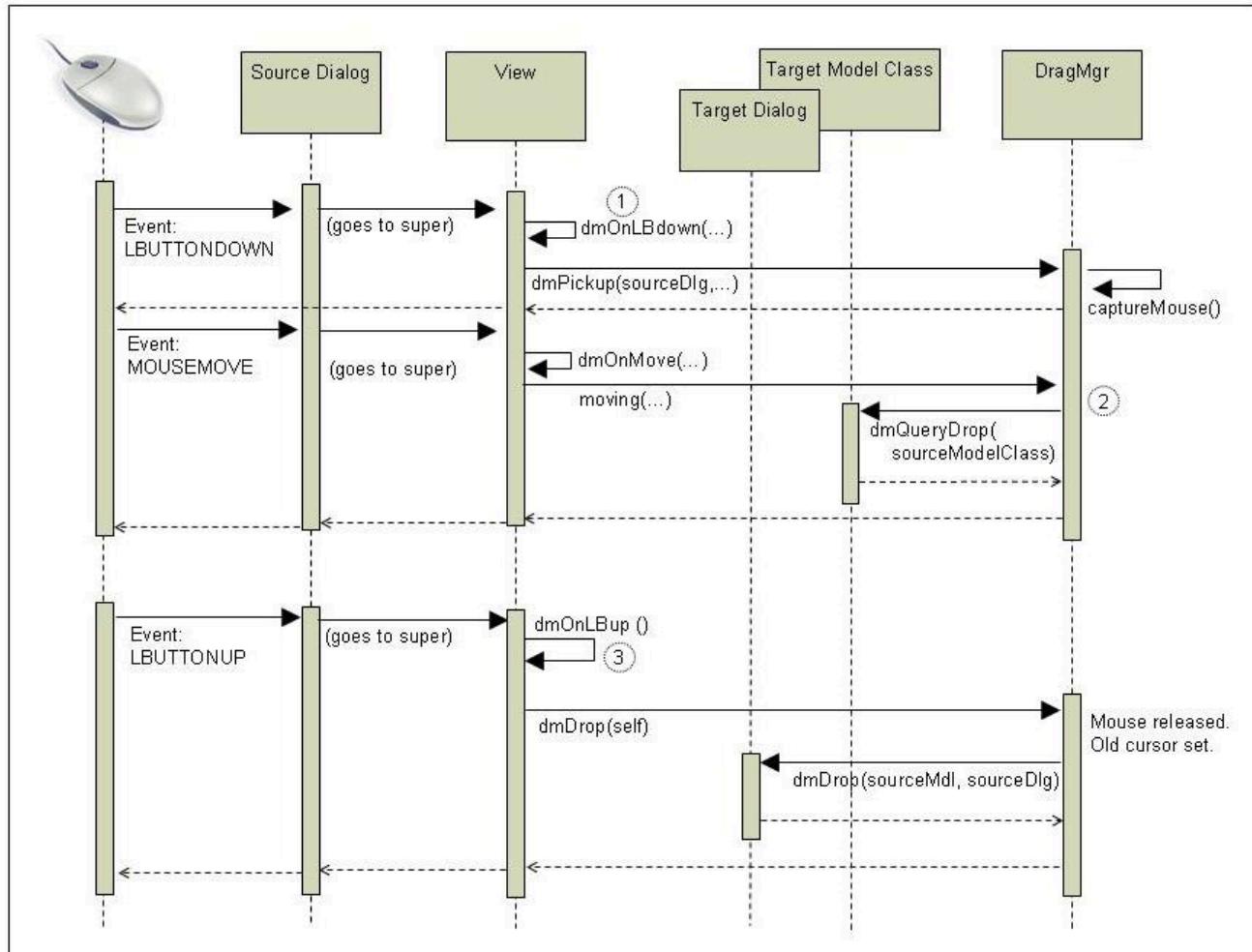

Figure E.2. Drag/Drop Operation

The operation is as follows:

1. The operation starts with the user pressing and holding the "primary" button (usually the left button) of the mouse while over a "source" dialog. This results in the event **LBUTTONDOWN** being signaled to the source dialog. This is handled by the **View** superclass's **dmOnLBdown** method, which in turn invokes the Drag Manager's **dmPickup** method. The Drag Manager then "captures" the mouse instance (using the Mouse class's **capture** method). This results in all mouse messages being sent to the source dialog where they are handled by the **View** superclass.

2. As the mouse is dragged, multiple **MOUSEMOVE** events are handled by the **View** superclass, which delegates them to the Darg Manager. For each such message, the Drag Manager checks if the mouse is over a target that's registered in its Target Table. If no, then nop. If yes, then a possible target has been found, at which point the Drag Manager:

    *   Sends a **dmQueryDrop** message to the class object of the target view's model with source's model class name as the single parameter. Why the class object? And why that of the target view's model? The reason is that, in the general case, the user may drag the mouse across a large number of possible targets such as multiple entries in a list view. And it could well be that

each item in the list represents a model and view instance (e.g. a list of Customers). Further, it may well be that none of them are instantiated (which could well involve accessing a data base of some sort). Now all that's required for the drag operation is a simple yes/no answer: can the source be dropped or not. Thus in order to avoid excessive and pointless processing, the model's class object is invoked. The target object returns a simple yes or no as to whether as drop on an instance of the target class is permissible.

- If the potential target returns true, then the Drag Manager set the cursor to the "drop OK" cursor - that is, the cursor provided by the source dialog when it told its superclass that it was a potential drag source.

E.5. Drop on a Target

The drop is illustrated diagrammatically in the bottom half of *Drag/Drop Operation*. A dialog (or a control within a dialog) sets itself as a potential drop target by supering a **dmSetAsTarget** message. The **View** superclass then sends the Drag Manager a **setTarget** message with three parameters: the dialog id, the dialog's window id, and the drop area. The Drag Manager stores this information in a table. A drop occurs when the user releases the left button. When this happens, the event handling method **dmOnLBup** in the **View** superclass is invoked. **View** then sends the Drag Manager a **dmDrop** message. This has three arguments: the dialog id, the key state, and the mouse position. The Drag Manager first releases the mouse and resets the mouse cursor. Then it sends a **dmDrop** message to the target view's model object, with two arguments: the model id of the source object, and the source dialog's id.

The target object then does whatever it wishes to do. Often target's action is to send a **query** message to the source model in order to get its data. The drag-drop operation is now complete.

Appendix F. Notices

Any reference to a non-open source product, program, or service is not intended to state or imply that only non-open source product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any Rexx Language Association (RexxLA) intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-open source product, program, or service.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurement may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-open source products was obtained from the suppliers of those products, their published announcements or other publicly available sources. RexxLA has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-RexxLA packages. Questions on the capabilities of non-RexxLA packages should be addressed to the suppliers of those products.

All statements regarding RexxLA's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

F.1. Trademarks

Open Object Rexx™ and ooRexx™ are trademarks of the Rexx Language Association.

The following terms are trademarks of the IBM Corporation in the United States, other countries, or both:

1-2-3
AIX
IBM
Lotus
OS/2
S/390
VisualAge

AMD is a trademark of Advance Micro Devices, Inc.

Intel, Intel Inside (logos), MMX and Pentium are trademarks of Intel Corporation in the United States, other countries, or both.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the Unites States, other countries, or both.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.

F.2. Source Code For This Document

The source code for this document is available under the terms of the Common Public License v1.0 which accompanies this distribution and is available in the appendix *Appendix G, Common Public License Version 1.0*. The source code is available at *https://sourceforge.net/p/oorexx/code-0/HEAD/ tree/docs/*.

The source code for this document is maintained in DocBook SGML/XML format.

The railroad diagrams were generated with the help of "Railroad Diagram Generator" located at *http:// bottlecaps.de/rr/ui*. Special thanks to Gunther Rademacher for creating and maintaining this tool.

Appendix G. Common Public License Version 1.0

THE ACCOMPANYING PROGRAM IS PROVIDED UNDER THE TERMS OF THIS COMMON PUBLIC LICENSE ("AGREEMENT"). ANY USE, REPRODUCTION OR DISTRIBUTION OF THE PROGRAM CONSTITUTES RECIPIENT'S ACCEPTANCE OF THIS AGREEMENT.

G.1. Definitions

"Contribution" means:

1. in the case of the initial Contributor, the initial code and documentation distributed under this Agreement, and

2. in the case of each subsequent Contributor:
   a. changes to the Program, and

   b. additions to the Program;

where such changes and/or additions to the Program originate from and are distributed by that particular Contributor. A Contribution 'originates' from a Contributor if it was added to the Program by such Contributor itself or anyone acting on such Contributor's behalf. Contributions do not include additions to the Program which: (i) are separate modules of software distributed in conjunction with the Program under their own license agreement, and (ii) are not derivative works of the Program.

"Contributor" means any person or entity that distributes the Program.

"Licensed Patents " mean patent claims licensable by a Contributor which are necessarily infringed by the use or sale of its Contribution alone or when combined with the Program.

"Program" means the Contributions distributed in accordance with this Agreement.

"Recipient" means anyone who receives the Program under this Agreement, including all Contributors.

G.2. Grant of Rights

1. Subject to the terms of this Agreement, each Contributor hereby grants Recipient a non-exclusive, worldwide, royalty-free copyright license to reproduce, prepare derivative works of, publicly display, publicly perform, distribute and sublicense the Contribution of such Contributor, if any, and such derivative works, in source code and object code form.

2. Subject to the terms of this Agreement, each Contributor hereby grants Recipient a non-exclusive, worldwide, royalty-free patent license under Licensed Patents to make, use, sell, offer to sell, import and otherwise transfer the Contribution of such Contributor, if any, in source code and object code form. This patent license shall apply to the combination of the Contribution and the Program if, at the time the Contribution is added by the Contributor, such addition of the Contribution causes such combination to be covered by the Licensed Patents. The patent license shall not apply to any other combinations which include the Contribution. No hardware per se is licensed hereunder.

3. Recipient understands that although each Contributor grants the licenses to its Contributions set forth herein, no assurances are provided by any Contributor that the Program does not infringe the patent or other intellectual property rights of any other entity. Each Contributor disclaims any liability to Recipient for claims brought by any other entity based on infringement

of intellectual property rights or otherwise. As a condition to exercising the rights and licenses granted hereunder, each Recipient hereby assumes sole responsibility to secure any other intellectual property rights needed, if any. For example, if a third party patent license is required to allow Recipient to distribute the Program, it is Recipient's responsibility to acquire that license before distributing the Program.

4. Each Contributor represents that to its knowledge it has sufficient copyright rights in its Contribution, if any, to grant the copyright license set forth in this Agreement.

G.3. Requirements

A Contributor may choose to distribute the Program in object code form under its own license agreement, provided that:

1. it complies with the terms and conditions of this Agreement; and

2. its license agreement:

    a. effectively disclaims on behalf of all Contributors all warranties and conditions, express and implied, including warranties or conditions of title and non-infringement, and implied warranties or conditions of merchantability and fitness for a particular purpose;

    b. effectively excludes on behalf of all Contributors all liability for damages, including direct, indirect, special, incidental and consequential damages, such as lost profits;

    c. states that any provisions which differ from this Agreement are offered by that Contributor alone and not by any other party; and

    d. states that source code for the Program is available from such Contributor, and informs licensees how to obtain it in a reasonable manner on or through a medium customarily used for software exchange.

When the Program is made available in source code form:

1. it must be made available under this Agreement; and

2. a copy of this Agreement must be included with each copy of the Program.

Contributors may not remove or alter any copyright notices contained within the Program.

Each Contributor must identify itself as the originator of its Contribution, if any, in a manner that reasonably allows subsequent Recipients to identify the originator of the Contribution.

G.4. Commercial Distribution

Commercial distributors of software may accept certain responsibilities with respect to end users, business partners and the like. While this license is intended to facilitate the commercial use of the Program, the Contributor who includes the Program in a commercial product offering should do so in a manner which does not create potential liability for other Contributors. Therefore, if a Contributor includes the Program in a commercial product offering, such Contributor ("Commercial Contributor") hereby agrees to defend and indemnify every other Contributor ("Indemnified Contributor") against any losses, damages and costs (collectively "Losses") arising from claims, lawsuits and other legal actions brought by a third party against the Indemnified Contributor to the extent caused by the acts or omissions of such Commercial Contributor in connection with its distribution of the Program in a commercial product offering. The obligations in this section do not apply to any claims or Losses relating to any actual or alleged intellectual property infringement. In order to qualify, an Indemnified

Contributor must: a) promptly notify the Commercial Contributor in writing of such claim, and b) allow the Commercial Contributor to control, and cooperate with the Commercial Contributor in, the defense and any related settlement negotiations. The Indemnified Contributor may participate in any such claim at its own expense.

For example, a Contributor might include the Program in a commercial product offering, Product X. That Contributor is then a Commercial Contributor. If that Commercial Contributor then makes performance claims, or offers warranties related to Product X, those performance claims and warranties are such Commercial Contributor's responsibility alone. Under this section, the Commercial Contributor would have to defend claims against the other Contributors related to those performance claims and warranties, and if a court requires any other Contributor to pay any damages as a result, the Commercial Contributor must pay those damages.

G.5. No Warranty

EXCEPT AS EXPRESSLY SET FORTH IN THIS AGREEMENT, THE PROGRAM IS PROVIDED ON AN "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, EITHER EXPRESS OR IMPLIED INCLUDING, WITHOUT LIMITATION, ANY WARRANTIES OR CONDITIONS OF TITLE, NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Each Recipient is solely responsible for determining the appropriateness of using and distributing the Program and assumes all risks associated with its exercise of rights under this Agreement, including but not limited to the risks and costs of program errors, compliance with applicable laws, damage to or loss of data, programs or equipment, and unavailability or interruption of operations.

G.6. Disclaimer of Liability

EXCEPT AS EXPRESSLY SET FORTH IN THIS AGREEMENT, NEITHER RECIPIENT NOR ANY CONTRIBUTORS SHALL HAVE ANY LIABILITY FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING WITHOUT LIMITATION LOST PROFITS), HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OR DISTRIBUTION OF THE PROGRAM OR THE EXERCISE OF ANY RIGHTS GRANTED HEREUNDER, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

G.7. General

If any provision of this Agreement is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this Agreement, and without further action by the parties hereto, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.

If Recipient institutes patent litigation against a Contributor with respect to a patent applicable to software (including a cross-claim or counterclaim in a lawsuit), then any patent licenses granted by that Contributor to such Recipient under this Agreement shall terminate as of the date such litigation is filed. In addition, if Recipient institutes patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Program itself (excluding combinations of the Program with other software or hardware) infringes such Recipient's patent(s), then such Recipient's rights granted under Section 2(b) shall terminate as of the date such litigation is filed.

All Recipient's rights under this Agreement shall terminate if it fails to comply with any of the material terms or conditions of this Agreement and does not cure such failure in a reasonable period of time after becoming aware of such noncompliance. If all Recipient's rights under this Agreement terminate, Recipient agrees to cease use and distribution of the Program as soon as reasonably practicable.

However, Recipient's obligations under this Agreement and any licenses granted by Recipient relating to the Program shall continue and survive.

Everyone is permitted to copy and distribute copies of this Agreement, but in order to avoid inconsistency the Agreement is copyrighted and may only be modified in the following manner. The Agreement Steward reserves the right to publish new versions (including revisions) of this Agreement from time to time. No one other than the Agreement Steward has the right to modify this Agreement. IBM is the initial Agreement Steward. IBM may assign the responsibility to serve as the Agreement Steward to a suitable separate entity. Each new version of the Agreement will be given a distinguishing version number. The Program (including Contributions) may always be distributed subject to the version of the Agreement under which it was received. In addition, after a new version of the Agreement is published, Contributor may elect to distribute the Program (including its Contributions) under the new version. Except as expressly stated in Sections 2(a) and 2(b) above, Recipient receives no rights or licenses to the intellectual property of any Contributor under this Agreement, whether expressly, by implication, estoppel or otherwise. All rights in the Program not expressly granted under this Agreement are reserved.

This Agreement is governed by the laws of the State of New York and the intellectual property laws of the United States of America. No party to this Agreement will bring a legal action under this Agreement more than one year after the cause of action arose. Each party waives its rights to a jury trial in any resulting litigation.

Appendix H. Revision History

Revision 1-0   Fri Jan 4 2013   Oliver Sims
   First version, exercises 02-06.

Revision 1-1   Fri Mar 22 2013   Oliver Sims
   Second version, exercises 02-07.

Revision 1-2   Fri Aug 31 2013   Oliver Sims
   Third version, exercises 02-08.

Index

Symbols

.Application
    and Unit-Testing, 35
    setDefaults, 16

A

Accelerator key
    in menus, 26
activate method
    in View superclass, 53
Attributes, of a dialog, 69
Autoconnection, 19
AutoDetection
    turning off, 70
Automatic data detection, 69

B

Binary Resource Dialogs, 23
Binary resource files, 23
Bitmap Editor, 44
Blocking
    concurrency issue, 10
Business Component, 7, 78

C

Class methods
    newInstance, 23
Code structure, 7
Common Public License, 89
Compilation
    DLLs, 23
Compiling a resource file, 24
Component, 9
    Kinds of, 54
    Name, 49
    Person, 50
Component concepts, 7
    Reference, 7
Component names
    Naming convention, 36
Concurrency issue, 10
Control
    Re-sizing, 57
Control Dialog, 59, 59
Controls
    ListView, 39
    RadioButton, 25
    Styles, 27
Controls Library, 27
Convention

    Naming, 13
Coupling, 9
CPL, 89
Creating an Image, 27
CustomerView component, 12

D

Data Types, 28
DateTime class, 60
DateTimePicker, 60
Debugging, 53
defineSizing method, 58
Design of a dialog, 30
Design, of dialogs, 12
Dialog
    Creation, 76
    Design, 30
    Re-sizing, 57
Dialog Attributes, 69
Dialog Data, 69
Dialog template, 12
Direct manipulation, 61, 82
    Source dialog, 84
    Target dialog, 84
dlgdata, 69
DLL
    Compiling, 23
Drag-Drop, 61
Drag-drop, 82

E

Edit control
    Numeric-only, 25
Event handler, 19
    Public method, 19
Exercise location, 2

F

File path, 34
Font
    in a ListView, 43

G

Generic File, 53
getFile method, 54
getInstanceName, 52
getRecord method, 54
globalConstDir, 16

I

Icon editor, 44
Icons
    in a ListView, 39