Slide 1 – Who Am I?

Slide 2 – Introduction

ooDialog provides GUI capabilities for ooRexx. However, the function provided is necessarily fairly low-level. Thus an application programmer must understand a great deal about GUI programming. The objective of the ooDialog User Guide is twofold:

- From the simplest possible dialog, to build, step-by-step, a simple but working "Sales Order Management" application, discussing and illustrating en route a number of ooDialog capabilities.
- To provide greater simplicity for the application programmer by progressively building a "framework" or "infrastructure" that simplifies the building of multi-dialog apps. (For example, the Framework makes drag/drop much easier than the "raw" ooDialog function.)

Each chapter in the Guide is associated with worked examples (called "exercises"). By the way, running all of the examples (or exercises) in the Guide requires ooDialog 4.2.4, which is not currently distributed in the ooRexx package. So it has to be downloaded separately. If not downloaded, only exercises 2 to 5 will work.

Let's start by introducing the three main kinds of dialogs.

Slide 3 – Main ooDialog Superclasses

ooDialog provides a number of dialog superclasses, but the Guide mainly uses what are arguably the simplest: `RcDialog`, `ResDialog`, and `UserDialog`.

The slide shows examples of dialogs from the Guide – a Customer dialog, a Product Details dialog, and a Sales Order Management dialog (from which Customer, Product and Sales Order details can be launched).

`UserDialog` requires the programmer to programmatically place and size all the controls on a dialog. ("Controls" are the things you see in a dialog window such as entry fields, push buttons, radio buttons, etc.) So one might perhaps use UserDialog for either very simple or quite complex dialogs.

`RcDialog` and `ResDialog` are designed to be used with a `.rc` file that defines the dialog layout. The layout is typically produced using a layout tool such as ResEdit (a freeware tool).

Slide 4 – A Dialog Layout Tool

A `.rc` file is a human-readable (but only just ☺) file produced by a layout tool such as "ResEdit". This slide shows the ResEdit layout tool being used to design the Customer dialog explained in detail in Chapter 4 of the Guide). The resulting `.rc` file …

Slide 5 – The .rc File

… while perhaps technically human-readable … is used directly by an RcDialog (together with the `.h` file also produced). However, for a dialog subclassed from ResDialog, the `.rc` file must be compiled into a `.dll` file. Otherwise there's no difference.

So, Chapter 4 introduces a "Customer" dialog with a `.h` and a `.rc` file, and Chapter 5 introduces a "Product" dialog which has a `.h` file and a `.dll` file compiled from a `.rc` file.

However, enough about layout tools.

Slide 6 – ooDialog Basics

Let's look at how the Guide introduces the ooDialog basics, stating with Chapter 2 (the first chapter is a short "About This Book" chapter).

Chapter 2 is called "Hello ooDialog World", and takes the reader through the creation of the simplest "Hello World" dialog.

The slide shows the complete code for this dialog. First, the HelloWorld class is defined in the `::CLASS` statement. The single required method is `init`. Here we first invoke the superclass (this is an ooDialog requirement), then create an instance of the Hello World dialog with the text "Hello World" as its title, centered in the dialog's title bar.

Btw, note the `::REQUIRES` statement, which is required for all ooDialog programs.

Finally, the two program statements at the top first create the dialog, and then (an ooDialog requirement) executes it. The SHOWTOP argument says show the dialog on top of any others and in the centre of the screen. The second parameter defines the icon placed to the left of the title bar.

That's it – simple – a Hello World dialog.

Slide 7 – "Words of Wisdom"

Chapter 2 then goes on to add function to the dialog: that is, to display, on the press of a button, so-called "words of wisdom" - e.g. "Complex problems have simple, easy-to-understand wrong answers". (Btw, this is an ooDialog version of a very old REXX exec which I used to have fired off when I signed on in the morning back in the days of 3270s).

The Words of Wisdom dialog introduces the basics of "controls" (push-buttons, text fields, etc).

Slide 8 – Separation of Concerns

Chapter 3 of the Guide has the same function as Chapter 2, but introduces a separation of concerns that is used for the remainder of the Guide's worked examples.

In general, what might be called a "component-based" approach is taken. First, `Wow3.rex` separates the code into three classes: `WowView.rex` (the dialog or "view"), "`WowPicker.rex`" (the business logic – such as it is) and `WowData.rex` (access to data, notionally on disk but actually hard-coded in an array).[1]

Following this approach, the three WOW programs make up what might be called a "Business Component", where the "business" is the selection and display of words of wisdom.

In addition, the launching of this simple application is done by an "application startup" program called `startup.rex`. This separates the necessary application launch functions (e.g. the creation of the dialog and the other two classes) from the application itself.

Finally, ***please note*** that this particular approach to structuring the code is a personal preference. ooDialog is very happy with all sorts of other approaches!

Slide 9 – Business Components

Chapter 4 introduces a "Customer" component, and the component approach is maintained throughout the Guide. Thus each subsequent exercise contains copies of or developments of previous exercises, so that what is being built, step-by-step, exercise-by-exercise, is a working Sales Order application.

For example, here is the folder list for Exercise 8. Each folder (other than "Support" and "Extras") contains a "business component". The "Support" folder contains the "framework" (which we'll discuss shortly), and the "Extras" folder contains a trivial "Person" business component used in the Guide as a very simple business component to better explain the essence of the Framework as it looks to the programmer. Finally, separating the various "::REQUIRES" statements into a separate file seems to make sense for a multi-dialog application.

Slide 10 - The "Product" Business Component

As an example, here is the folder containing the Product business component – that is, all and only files concerned with the "Product" concept. Note that the Product Model and Product Data components are grouped in the same file – mainly because the data component is very simple – it subclasses a "generic" file access component used by all business components. File access is read-only so far, read-write being a planned future function, possibly using ooSQL.

[1] Note to Presenter: This is somewhat similar to the MVC (Model-View-Controller) concept, except that the classic "controller" function is provided by ooDialog.

Slide 11 – Opening Another Dialog

Let's now return to ooDialog function.

Chapters 4 and 5 describe in some detail a "Customer" dialog and a "Product" dialog, each introducing various ooDialog function such as a number-only edit field, disabling and enabling controls, defining menus, controlling dialog cancel, and the sequence of events when a dialog is created. At this point, data is hard-coded – a single record only.

Chapter 6 introduces the notion of an "application workplace" – that is, the dialog that represents the application and its various parts.

For example, double-clicking on the "Product List" icon produces a Product List dialog.

In previous chapters, the single dialogs (Words of Wisdom, Customer, and Product) were created using ooDialog's `self~execute` Method. The `execute` method causes the dialog to appear, ***and*** - it also blocks until the "child" dialog is closed. Thus no other dialog can be launched until the child dialog closes.

But the Sales Order Management application requires more than one dialog to be opened and closed without closing the application! Luckily ooDialog provides a non-blocking way of launching dialogs: the `popup` and `popupAsChild` methods (discussed in detail in Appendix B of the Guide).

However, the Guide also seeks to provide a framework that simplifies the task of building a real application. This is mainly done in Chapters 7 and 8.

Slide 12 – The "Infrastructure"

The "Infrastructure" (or "layer" or "framework") is intended to make ooDialog programming much simpler than using "raw" ooDialog function. It provides a number of functions that can make multi-dialog applications a great deal simpler than they otherwise might be. These are:

- The "Component Approach" - aka View-Model-Data separation of concerns
- Class Hierarchy
- Dialog setup - The Model-View Framework
- Debug Tool (aka "Message Sender")
- Drag/Drop (aka Direct Manipulation)
- Event Management

These are described in Chapters 7 and 8 of the Guide, and are illustrated by working examples (aka "exercises") 7 and 8.

Slide 13 – The Component Approach

We have already mentioned the structuring approach taken – what might be called a "component approach".

Slide 14 – Class Hierarchy

Part of the Framework is implemented in "Manager" programs (of which more later), and the rest in superclasses.

The red classes are ooDialog classes. Green classes are Framework classes. The single black class at the top is, of course, the ooRexx "Object" superclass. Blue classes are application classes.

Solid lines indicate normal subclass-superclass relationships, and dotted lines show mixins.

Thus, for example, CustomerView inherits from RcDialog, View, and Component (see class definition at bottom of slide:

```
::CLASS CustomerView SUBCLASS RcDialog PUBLIC INHERIT View Component
```

But why have all these superclasses?

Well, take "Dialog Setup" for example. This is an important part of the Model-View Framework or MVF.

Slide 15 – Dialog Setup – The Model-View Framework

[Chapter 7 of the Guide]

The objective of the Model-View Framework (MVF) is to provide a mechanism whereby application components can read data and display views without needing to be aware of how this is done.

Consider what must happen when, say, a user double-clicks on an item in a Customer List which invokes the showCustomer method of the CustomerList. Before the Customer dialog cab be displayed – with its data – a number of questions have to be answered – as per the top bullet on the slide.

In order to answer all of these questions, first the dialog to be launched (or displayed or maximised) must be identifiable in some way. While there could be a number of ways of doing this, the approach adopted is to require each component instance to have a name. Thus the Customer dialog with the key "AB0784" is defined as having the name or identity: "CustomerView AB0784". In effect, this can be thought of as "ClassName InstanceName".

But what more anonymous dialogs such as Customer List? Well, these are deemed to be "anonymous" components, and have the name "CustomerListViewaboput things

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 dialog. The particular program in the MVF that does this is called the "Object Manager" (`ObjectMgr.rex`), and is located in the "Support" folder. Thus the Object Manager distinguishes between a number of different dialog states, and relieves the programmer from having to code the logic for each component and for each possible state.

Finally, the application code required to surface a dialog is shown on the slide. Thus all that the Customer List need do (excluding error-checking) is:

1. Identify the Customer record in the list view that has been double-clicked
2. Get the Object Manager (`ObjectMgr.rex`) from `.local` (placed there by itself on app startup).
3. Ask the Object Manager to show the component called "CustomerModel – AB1234"

Components are identified by their class name and their instance name. Note that the class name in the sample code is "CustomerModel", and the instance name is the Customer Number read from the List Box.

Slide 16 – The Message Sender – Querying a Customer Instance

The Message Sender is a stand-alone de-bugging tool that sends messages to components. It is invoked through the Help menu of the Sales Order Management dialog. At the top, the prompt "Target" is a pull-down for three objects – the Event Manager (which will be discussed shortly), the Object Manager, and a sample "Person" dialog. Of course, any (valid) component name can be typed in. The slide shows a "query" message being sent to a CustomerModel instance. The response (the data held by the Customer) is shown in the Reply box.

The Message Sender can also send messages to the Object Manager, either to list the components it knows about or, given a model component name, to cause a its view to be surfaced.

Slide 17 – Dubugging – The Messgae Sender – What compnents exist?

Here is the Message Sender asking for a list of instantiated objects from the Object Manager…

Slide 18 – List of Components

… and here is what the console looks like after the "List" request from the Message Sender. Btw, you can have multiple Message Senders at the same time.

Slide 19 – Debugging – The Message Sender

Finally - and this is where the debugging capability starts to pay off – you can use the Message Sender to surface a Component View.

Slide 20 – Drag/Drop

Chapter 8 of the Guide introduces Drag/Drop and Event Management

Drag/drop is supported by ooDialog through its `.Mouse` class. But since it has to deal with all manner of drag/drop designs and situations, the support is quite complex.

In the Guide, however, drag/drop is implemented only for components. Thus support can be simplified, with the complexities delegated to the `View` superclass and a manager program called `DragManager.rex`.

Note that (so far anyway) the only things that can be picked up are a Customer (i.e. a CustomerView dialog) or a Product, and the only thing either can sensibly be dropped on is an Order Form.

There are three phases for drag/drop – (1) setup, (2) drag, and (if accepted) (3) the drop.

Slide 21 – Drag/Drop Setup - Code

As can be seen, the setup code is very simple – just tell your superclass that you can be picked up – or that you can be dropped on – or indeed both.

Specifically, in the `initDialog` method (the method invoked by ooDialog when a dialog is first created), the source dialog sends a `dmSetAsSource` message to itself. This is handled by the `View` superclass, which creates a `.Mouse` object and then connects events to it – including `LButtonDown`" and `LButtonUp`. `View` then sends a message to the Drag Manager (`DragManager.exe`), which adds the "source" dialog to a table that it maintains.

Some time later, or earlier, the target dialog, in its `initDialog` method, sends a `setAsTarget` message to its superclass (also `View`). View forwards this to the `DragManager`, which stores the target dialog in a table.

Slide 22 – Drag/Drop Operation

Briefly:

1. The drag/drop operation starts with the user pressing the left mouse button and holding it while the cursor is over a "source dialog" such as `CustomerView`. `ooDialog` generates a `LButtonDown` event which is caught by the source dialog's `View` superclass, which forwards it to the `DragManager`.
   Note that the source dialog itself plays no part in this – it's all done by its `View` superclass and the `DragManager`.

2. On moving over a potential target, `ooDialog` creates a `Move` event which is captured by the target dialog's `View` superclass. `View` then forwards it to the `DragManager`, which in turn sends the potential target (the Model not the View btw) a `dmQueryDrop` message. The target replies yes or no. If no, the `DragManager` turns the mouse icon into a "No Entry" sign.

3. When the user drops on a valid target, the `LButtonUp` event is caught by the target view's superclass (i.e. `View`), which then sends a `dmDrop` message to the `DragManager` – arguments being the source dialog's id and the source model's id. The `DragManager` then sends a `dmDrop` message to the target Model, so that the target can then "talk" either with the source's View or with its Model.

4. The target has now been properly introduced to the source, as it were, and so can talk with each other in a civilised fashion.

Note that the `dmQueryDrop` message is sent to the target's Model component, not to its View. This is because it's the Model that handles business logic, while the View handles presentation logic. The `dmDrop` message, however, is sent to the View. This is because a Model may have multiple Views. There's possibly a long discussion to be had about this, but not here.

Btw, it may be of interest that, in a single drag/drop operation, around 100 or more `mouseMove` events are generated.

Slide 23 – Event Management Framework (EMF)

Of course, ooDialog handles many events – such as menu selection. However, the events discussed here are application-level events. A good example is `OrderFormView`, which has two "Control Dialogs" (subclassed from ooDialog's `RcControlDialog`) within the main dialog. Before the main dialog is closed, first the Control Dialogs much be closed – otherwise the application hangs. Thus we need a way to signal the Order Form instance when the user closes the application.

The EMF works like this.

1. 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 – `appClosing` - and its object id (i.e. `self`).
2. The message is handled by the `Component` mixin, which forwards it to the `EventManager`, which adds the event to its directory of events. In this directory, each index is the event name, and each item is an array of dialogs (or other things) that have expressed interest in that event.
3. When the user closes the Sales Order Management dialog (the main "application" dialog) it receives a `cancel` message, in which (after checking with the user) it closes its two Control Dialogs, it supers a `triggerEvent("appClosing")` message to `Component`.
4. `Component` forwards the message to the `EventManager`, which then sends a `notify("appClosing")` message to all objects that have registered interest in this event – including `OrderFormView`.
5. `OrderFormView` catches the notify message, closes its Control Dialogs, and returns.

Then the app close is then completed, and the app closes.

Slide 24 – To Do…

Things still to do in further chapters of the Guide include:

- Provide for persistent update of Customer, Product and Order on disk
- ooSQL instead of flat text files
- Drag/Drop within a single dialog (where it makes sense!)
- Re-factor everything into a single "parent" window (this is called Multiple Document Interface or MDI – the usual approach for Windows apps these days).
- Separate component "class" names from ooRexx class names, using a configuration file to assign external names to ooRexx class names
- Move more business logic from View components to Model components.
- Any suggestions??? The End.