

Chapter 1: An example

In which we learn how to build a program using object oriented techniques

In the first part of this book, we will discuss how to build an interesting application in REALbasic using Object Oriented Programming (*OOP*) techniques. The features of this program are chosen to be particularly amenable to OOP techniques, and painful without these techniques. This is because OOP is all about *connections*.

A text processing program

Our example program processes text. It lets its user choose any of a variety of sources of text (the clipboard; text typed into a box; or text read from a file), process the text in any of a variety of ways (counting the words; finding lines containing certain text; or extracting a column), and provide the results in any of a variety of forms (onto the clipboard; to a file; or displayed in either a ListBox or an EditField):

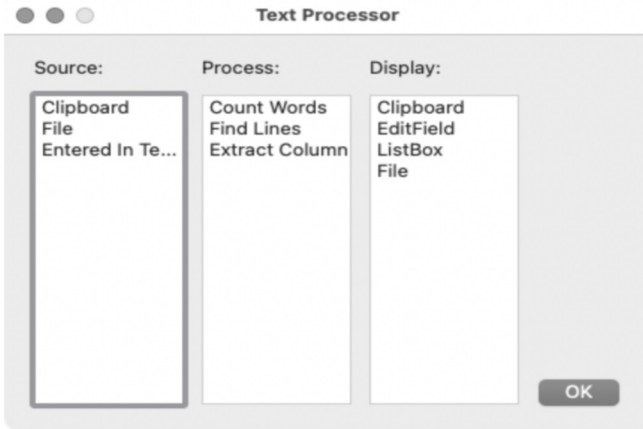


Figure 2: A text-processing application

We want the design to be very open—it should be as easy as possible to add new sources of text, new processes, and new ways of providing the results.

A non-OOP solution

I’m really glad that REALbasic doesn’t force us to use OOP (as opposed to, say, *Java*). In fact, you can write any kind of program that REALbasic is capable of creating without using any significant OOP features at all.

This is a good thing because for smaller programs, or for simpler parts of more complex programs, OOP is overkill. It imposes “engineering requirements” — declaring classes, initializing them and so on — that just get in the way of solving your problem.

But for any program or part of a program that involves lots of parts connecting up with lots of other parts, a little of that OOP engineering can *radically* simplify the design.

In order to see that, let's start by designing our example without using any significant OOP.

An obvious step is to create methods in the main window for the text processing (such as counting words), because the code for those will generally be non-trivial. On the other hand, reading the text in and displaying it is fairly simple, so we won't bother to put those steps in separate methods.

Once the user has chosen an option from each ListBox and clicked *OK*, we have to connect the source of text to the processing method and to the display. Without OOP, we need to do something like this, in the OK button's *Action* handler:

```

Dim sourceText As String
Select Case SourceBox.SelectedRow
Case 1
    Dim c As New Clipboard
    If c.TextAvailable Then
        sourceText = c.Text
    End If
Case 2
    //etc
End Select

Dim resultText As String
Select Case ProcessBox.SelectedRow
Case 1
    resultText = countWords(sourceText)
Case 2
    resultText = findLines(sourceText) //Displays dialog requesting search str
Case 3
    resultText = extractColumn(sourceText)//Displays dialog
End Select

Select Case DisplayBox.SelectedRow
Case 1
    Dim c As New Clipboard
    c.Text = resultText
Case 2
    EFWindow.Show
    EFWindow.EditField1.Text = resultText
Case 3
    //etc
End Select...

```

Figure 3: Example of non-OOP OK button Action Event Handler

Notice that apart from the code that does the actual *work* we want (getting the text, processing the text, displaying the result), we've got a whole lot of other code, that:

- Processes the user interface (work out what is meant by the selection in each list box);
- Connects the various parts together; and
- Converts the results from one step to the form wanted by the next.

Alright, so we don't have any of the conversion work. Yet. But if we kept extending this program, eventually we would run into things like a bar graph results display that needed to work out what to do with the output from the “find the lines containing this phrase” process.

This *Action* Event Handler is an example of the kind of busy work I always hate to write. A whole bunch of different forms of the same thing, over and over: work out which line of the ListBox was clicked on, and do the corresponding thing. And make sure you carry over the result from one stage to the next.

Even worse than writing all those *Case whatever* clauses, I don't know about you, but I'm really horrible at remembering where I need to go to maintain this kind of thing. User interfaces are particularly bad: when the user chooses an option in one place, I've got to think of all the places I've got to hide and show things, disable and enable things, or whatever.

The connections and conversions and whatnot are pretty easy to maintain in this artificial example. But this tiny maintenance headache in this tiny program is a proxy for the big, complex maintenance headaches you have in big, complex programs. *Whenever you change this, you've got to remember to change that.*

Better organization

The simplest way OOP contributes to reducing this problem is that it offers a good way to organize the various parts of your project—it lets you keep the data and controls and other objects, and the code that accesses them, in one place (this is called *encapsulation*). We call this place a *Class* (or it might be a *Window*).

So rather than having to say:

```
EFWindow.Show
EFWindow.EditField1.Text = resultText
```

We can add a method to *EFFWindow*, and then do:

```
EFFWindow.ShowResult resultText
```

So we're moving the logic for showing text with the window into the window. And this lets us wrap up code that accesses multiple controls or other objects or values along with those things, so everything is more neatly organized. We'll see much more interesting examples of organizing our project using classes as we proceed.

Polymorphism: A better way to connect

The more interesting thing we can do using OOP is to radically alter the way the parts of this project relate to each other. At the moment, all the code to connect the various parts of our project together is outside of those parts themselves.

But what is interesting to note is that all of the connecting-up steps have basically the same form. For example, all the steps that get text from somewhere do something that produces text. All the steps that process text take text from the first step and produce some more text. And so on.

The really important feature in OOP, *Polymorphism*, lets other parts of the program treat the same any objects that “look the same”, without regard to what they actually do on the inside. And “look the same” means “have the same public methods”.

So you only need to write any *kind* of connecting-up once. Those big case statements just disappear into a single line. This eliminates a lot of code that you don’t have to debug and maintain.

That’s the essential advantage of OOP. What happens once you understand how much more easily you can connect things together in an OOP program, you can use quite different designs with more and richer types of connections. In the end, you can write more interesting and powerful and extensible programs. The first step is to lodge this whole approach to connections firmly in your head. That’s what

this book is intended to do. Longer-term, you need to get some experience working with these richer connections and learning some of the standard patterns based on OOP for solving problems. This book will give you a start on that.

OOP in our example

So in our example program, we can have a whole bunch of different text source objects, doing very different things internally, and any of our text processor objects can use any of them through the same connections. Another way of saying all that is that our text source objects can all play the same *role* for our text processor objects.

Let's look at how this idea works in our example program.

Using OOP, the connecting-up of the code that formerly occurred in the [Select Case](#) statements is now almost entirely inside a set of classes we define to represent the sources, the processes and the displays. The result is that our OK button action handler will now look something like:

```
Dim t As TextSource = TextSource(SourceList.CellTag(SourceList.SelectedRow, 0))
Dim p As TextProcessor = TextProcessor(ProcessList.CellTag(ProcessList.SelectedRow, 0))
Dim d As ProcessDisplay = ProcessDisplay(DisplayList.CellTag(DisplayList.SelectedRow, 0))

Try
    p.Process t
    d.Display p.GetResult
Catch e As Exception
    e.DisplayError
Catch e As GeneralException
    //Do nothing
End Try
```

Figure 4: The OOP OK Button Action Event Handler

We take care of the User Interface processing by using the `ListBox`'s *CellTag* property. Every call in the `ListBox` has associated with it a variant called `CellTag`. It will be very common in OOP to put an object in the `CellTag` that takes responsibility for the cell, or is the “content” of the cell.

In this case, we pull the *CellTags* out for the selected rows (casting them to the appropriate type as we go), and just connect them together. We ask the process to do its thing with the source of text, and then we ask the display to do its thing with the process. Any details of making them work with each other is now inside of the classes.

To make that *OK* button's code work, we define types called *TextSource*, *TextProcessor* and *ProcessDisplay*, and then set up our classes so that they assume the *role* of *TextSources*, *TextProcessors* or *ProcessDisplays*. This lets them all work together without needing to know any more details than necessary about each other.

For example, the *purpose* of any *TextProcessor* can be written entirely in terms of the external features (the *Public* Method calls and Properties) common to all *TextSources*. This lets us add any number of wildly different *TextSources*, and as long as they all provide equivalent external features, they will all work with all the *TextProcessors*.

OOP Text Input

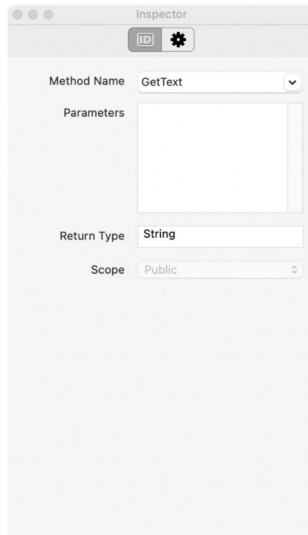
In our example, the *TextProcessor* has a method called *Process*, which takes a *TextSource* as its argument.

When execution gets inside that *Process* call, the *TextProcessor* will want to get text from the thing passed to it, so all of our *TextSource* objects will have a *GetText* function that returns a string.

So how do we define a *TextSource*? You are almost certainly thinking “Define a class”, and that would indeed be a way to define the *TextSource* type. But a much better solution is to define this particular type by creating a *Class Interface*.

You create a Class Interface by going to the *Insert* menu, and choosing *Class Interface*. Then add a *GetText* method with no parameters and a return type of *String*.

Here is what our *TextSource* Class Interface will look like:



*Figure 5: The
TextSource Class
Interface.*

Notice that there is no code. You *can't* add code. A Class Interface just defines what a class—potentially a whole lot of classes—must look like from the *outside*, in order to be treated as being one of the class interface's type. In this case, the *TextSource* class interface says something must have a *GetText* method, taking no arguments and returning a string, in order to be treated as a *TextSource*. But it says nothing about what such classes do inside the methods it mentions.

What good is such a thing? A Class Interface acts as a license. Very different people might have a license to practice medicine, but the license lets all those who have it *play certain roles*—they can write prescriptions, for example. Any doctor, no matter what their specialty is, can play the role of writing a prescription.

A Class Interface is similar. We can attach a Class Interface to an actual Class, and then that class can play the role defined by that Class Interface.

To make that work, after we define the *TextSource* Class Interface, we can add *TextSource* to a class's list of Class Interfaces, like so:

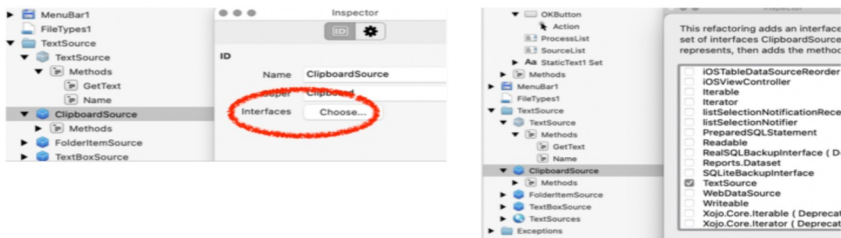
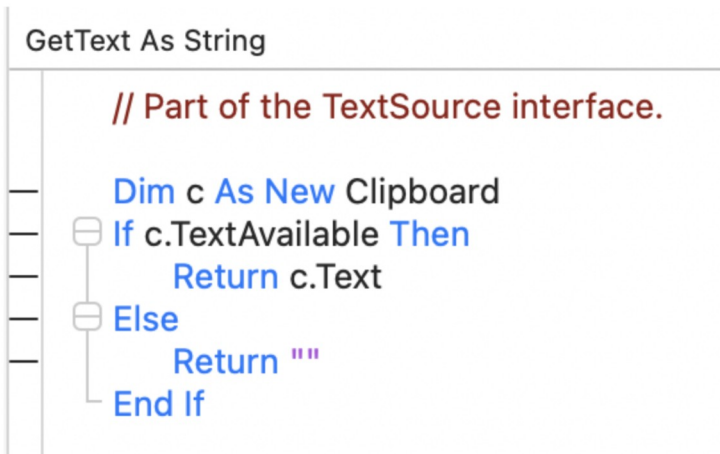


Figure 6: The *ClipboardSource* Class with its Class Interface

We must add all the methods in the Class Interface to the class (the IDE will actually offer to do this for you when you first add the Interface to the class, but it won't change the methods if you change the Class Interface definition).

A class can have any number of Class Interfaces in its Interfaces list. You can modify the list in two different ways: you can type comma-separated Class Interface names into the box, or you can click on the little grey dot, and choose from a list of all the Class Interfaces in your program. You'll notice if you pull up the list that REALbasic already has quite a few Class Interfaces defined. You should refer to the REALbasic documentation for information about what they do.

So our *ClipboardSource* class is a subclass of *Clipboard*, but can also be treated as a *TextSource* object, because it *satisfies the TextSource Class Interface*. Here is the code for its *GetText* method:



```

GetText As String
    // Part of the TextSource interface.

    Dim c As New Clipboard
    If c.TextAvailable Then
        Return c.Text
    Else
        Return ""
    End If
  
```

Figure 7: *ClipboardSource.GetText*

We can, similarly, create a *FolderItemSource* subclass of *FolderItem* and a *TextBoxSource* class (with no SuperClass), each having *TextSource* in its Interfaces list, and therefore each having a *GetText* method.

We are now almost ready to define the *TextProcess* type, but there is another detail we should clean up first.

OOP Connections

A general principle to be observed in any kind of programming, and one that OOP facilitates, is that any given thing the program does should involve as few parts of the program as necessary. One example of that is how the various objects get listed in the window, so the user can choose them.

We can observe this principle in our program by making it possible to add a new type of source, process or display to our program while requiring as few changes elsewhere as we can.

To handle building the lists in the main window, we provide a general set of calls in our window that adds a new source, process or display to the lists in the window. This sort of thing:

```
AddSources(ParamArray ts() As TextSource)
    For Each t As TextSource In ts
        SourceList.AddRow t.Name
        SourceList.CellTag(SourceList.ListCount - 1, 0) = t
    Next
    SourceList.Sort
```

Figure 8: The *MainWindow.AddSources* method

In case you're unfamiliar, each cell of a *ListBox* has a *Variant* value associated with it called *CellTag*. This will typically be used to hold a reference to the object that takes responsibility for that cell or row. That's exactly what we're doing here. You'll remember that our OK button's *Action* Event Handler pulls the tag out and *casts*³ it to a *TextSource*.

We will also have *AddProcesses* and *AddDisplays* methods that are just like this one, only they expect *TextProcess* and *TextDisplay* objects, and will set up the *ProcessList* and the *DisplayList*, respectively.

³*Casting* is just telling the compiler that an object can be treated as a certain type. A variant can hold just about any type of value, so we have to tell the compiler that the thing we're pulling out is going to be a *TextSource*. You can cast to a Class or a Class Interface, and you do it just by using the Class or Class Interface name like a function.