Messaging – the missing link

In the last 3 articles, we've looked at some nifty abstract class building and form embedding techniques. Just to reiterate:

1. We used an **array of class** and some class building techniques to simplify the creation of child classes.
2. We enhanced the method to use a **class function** as a pseudo constructor to create a child instance of our class.
3. We **embedded forms** in parent forms to allow better encapsulation and modularity.

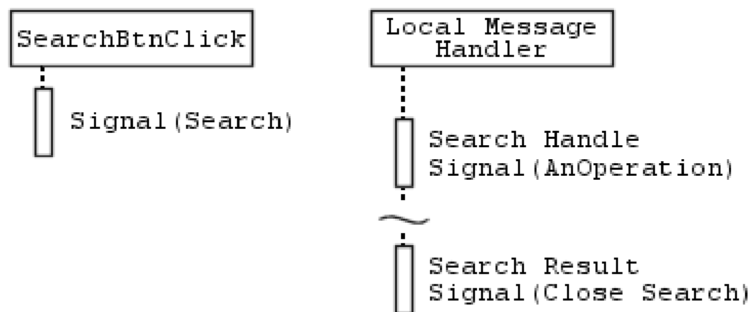
The next thing I'm going to look at is **internal messaging**, similar to the Windows message queue. The idea behind this is that segments of code can have much lower coupling, so they can interact without really knowing all that much about each other.

But like any approach there are pros and cons, the latter being that we have to manage an internal "database" of messages and message structures. Ideally these messages and structures should be somewhat generic, which has a tendency to give us a little bit more work to do than if we were calling a form or component directly.

The second and more difficult thing for people the grasp is that messages are not linear compared to the normal linear flow of code. Think of the following piece of code:

```
//1. Create the form
With TSearchForm.Create do
try
 //2. Run the Execute Method specifying
 //   the operation required
 Result:=Execute(AnOperation);
finally
 //3. Free the form
 Free;
end;
```

In messaging terms, this would require a number of operations – as follows:



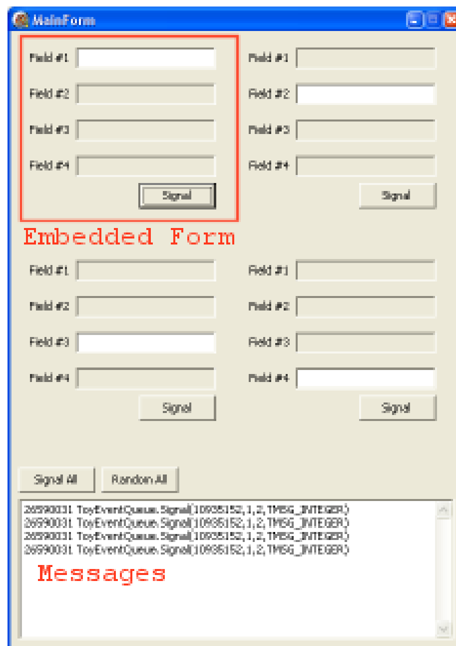
[SearchBtnClick] Firstly, we need to initial the request to create an instance of the search form. (Assume) somewhere in our application there is a component service that will act on this message.

[Local Handler #1] The search handle is returned and we issue a request to perform AnOperation similar to the original Execute function.

[Local Handler #2] Finally we have a search result, and we request the search form to finalise itself.

This approach can (and does) cause confusion, and even though this is not multi-threaded, developers not familiar with this approach can find it difficult. Even beyond this it is fundamentally a more awkward approach, as what was one linear piece of code is now essential split into 3 pieces.

But even without this complexity, there are some scenarios where there is an immediate win when using a messaging approach. The following is a very simple example that demonstrates this beautifully (*all code for this is available as a separate download*).



The main form (pictured) contains 4 panels, and on each panel an instance of an embedded form is created.

The embedded form supports 3 messages:

INIT: Configure the form – basically tells each form what Field to make editable.

CHANGE: To signal the change/update the change of an editbox.

RANDOM: To fill the forms editable field with a ~random~ number and signal the change.

The bottom memo control displays a list of messages that pass through the system.

Although a bit contrived, it is a good example of how forms – that really have no knowledge of each other can pass information. As messages flow through the system, relevant messages can be acted on by discrete parts of the system as appropriate.

In this case, the Signal button in each embedded form issues the following:

```
EventQueue.Signal(MSG_WINDOW,
                  MSGW_CHANGE,
                  TMSG_CHANGE.Create(myMode, myEdit.Text));
```

In this case, an MSG_WINDOW message is passed with a subtype of MSGW_CHANGE. Included in the message is an instance of the TMSG_CHANGE class. In this implementation the message queue will free any attached object instances attached to the message when the message broadcast has completed – unless told otherwise.

In the example the so-called internal database of messages are stored in the fMessages.pas file. One design issue that should be considered is in the use of the TMSG_CHANGE class in the application. This is in essence not really a generic message (ok – you'd probably put something better in place) as its purely designed to hold the field index (integer) and field data (string) coming from the embedded forms. The design issue arises as follows:

1. If non-generic message classes are stored in the message "database", the file can become large and unmanageable, and can - depending on implementation start to generate a number of unwanted dependencies between different source files in the application.

2. If non-generic messages are stored in their respective source files, different source files acting on those messages need to link to those specific source files – increasing coupling.

Ideally, the message structures would contain a sufficient amount of generic structures that most every scenario could be addressed by passing one or more messages to complete a request or notification. Naturally the more messages you need to pass to say request an operation the more complicated the code becomes.

Finally the messaging system in the example is pretty simple, but it implements one nifty feature. When a message is put on to the queue, it is received by each "client" in the order that those clients were added to the queue – with one exception. The sender is always the last client to receive the message that it sent!

In the example all the messages are notifications. However the simple messaging queue also supports requests, in that a message can be issued, pass down the queue, and processed with the results by the sender when it is completing its journey.

The example code is available with this 5 minute article includes the simple messaging component. This component needs to be installed in the IDE before loading the example file.