Subject: Re: IDLVM and retall
Posted by JD Smith on Fri, 08 Sep 2006 16:26:53 GMT
View Forum Message <> Reply to Message

On Fri, 08 Sep 2006 10:57:48 -0400, Ben Tupper wrote:

```
> David Fanning wrote:
>> JD Smith writes:
>>
>>> Here's a better example:
>>>
>>> pro MyClass:f1, ev
>>>   if something_is_wrong self->Error
>>>   do_calc
>>> end
>>>
>>> pro MyClass:f2,ev
>>>   self->f1,ev
>>>   do_another_calc
>>> end
>>>
>>>> Can this be a difference between Windows and UNIX?
>>> If you simply return (or return twice, if you were able to do that
>>> from within self->Error), you'd be right back inside MyClass:f2 and
>>> do_another_calc would run, which is not what you want.  With RETALL,
>>> the entire calling stack is returned from, not just one step in the
>>> stack.
>>
>> OK, I see what you mean. Yes, when you are trying to
>> merge widgets and objects things do tend to get a little
>> messy. I don't find myself buried this deep in my widget
>> programs, which is why RETURN has always worked for
>> me. But I do remember some messy, messy code in my
>> Catalyst Library to keep error messages from propagating
>> up the whole damn call stack! Essentially, we had to
>> keep track of whether the error had already been
>> "handled" or not, and keep issuing RETURNs if it had.
>> Your solution is probably better looking than this. :-)
>
> Hi,
>
> At risk of exposing my arrested development, I harken back to exchange I
> witnessed between David and Martin Schultz eons ago.  I can't remember
> if it was on the newsgroup or "off-line".  But I do remember clearly
> that Martin and David leaned toward a FUNCTION event handling method
> rather than a PROcedure - the function event handling method returned
> 0/1 for fail/success. If I remember rightly, the decision was driven by
> the messaging aspect of events (or pseudo-events) and that functions, if
>  nothing else, pass messages by default.
>
>  Would that suffice to resolve the issue?
>
>  function MyClass::f1, ev
>     if something_is_wrong then begin
>       self->Error
>       return,0
>     endif
>     return,do_calc()
>  end
>  function MyClass::f2,ev
>     OK = self->f1(ev)
>     if OK then OK = do_another_calc()
>     return, OK
>  end
>
>  Of course, it was a long time back and I had hair and a shorter
>  attention span. I still have the same attention span - but the hair has
>  gone AWOL.
```

The features I want in my error handling system are:

1. The Error method prompts the user with an error (either printing it of popping up a dialog), and either stops processing (non-interactive), or clears the entire calling stack to continue the application (interactive).

2. Can be called from *anywhere* on the calling stack, not just an event handler, but any method function or procedure, at any arbitrary depth.

3. The routine which calls Error doesn't have to check its return value, return anything special or do anything else to signal an error.

4. Doesn't require use of the traditional event callback system to work, since I use an object messaging framework which expands the concept of widget events to any generic "message" with which two objects can communicate.

For this reason, the standard tricks using the event handler didn't work for me, and that's why I came to RETALL. I've recently found that running your primary event loop in XManager while in the VM (vs. the active command line, for the non-blocking version) with a top level catch is a perfectly acceptable equivalent solution, if and only if you have exactly one entry point into your application (i.e. one predictable "first" call to XManager). Luckily, this is true for a compiled binary app, which is where RETALL doesn't work.

It's particularly nice just to say:

```
pro MyClass:Foo

  if x lt y then self->Error,'X must be GE Y'
  x-=y
  ...
end
```

and not worry about the details, no other handling required.

I should re-emphasize that this only works if you can cleanly unwind
the entire calling stack at the point the error is signaled and still
have a runnable program. Objects help a lot with this, given their
persistence and slow "churn" in instance data (vs. a state structure
which might be moved in and out of local scope thousands of times a
second). If you are careful and only change important instance data when
the operation has succeeded, it's easy to make methods which can be
interrupted in the middle, and then later called again with no ill effects.

JD