I haven't really written any documenation still; I AM LAME, okay?

old stuff:

send mail to wave@media.mit.edu with any questions.

The scenario I wished to address was this:

IB, as it currently ships, is designed to be used with a compiled language, Objective-C. Given that, IB is a reasonable prototyping environment for applications which use already compiled objects, but falls down parts of the interface depend on objects which are yet to be written. Even if they're written, they need to be palettized to be useful in "test interface" mode in IB.

Basically, I wanted to have my cake and eat it too. To be more explicit, there are two incompatible methods of using IB to help prototype an app:

Having your cake

You've already given a lot of thought to the objects you will compose your application with. You've written the code, and palettized the objects. You can now go into IB and prototype your application, by dragging objects off your palettes, hooking them up, and going into test interface mode. Unfortunately, you can't redesign any of your objects on the fly. This wouldn't be that bad if you weren't crippled by IB's target/action paradigm. (more about that later)

Eating your cake

You sit down to IB with some pencil sketches of your app, but much of the details are still unformed. You start dragging out UI objects, and you start defining some new classes. You define

outlets and action methods, hook things up, think some more, change your mind. It's great. Unfortunately, "test interface" mode is now worse than worseless; it's annoying because since many of the objects in your nib file haven't been written yet, it doesn't show you anything like what your interface will eventually look like.

So what's the solution? Well, there are several possibilities. Two nice ones I've seen lately is Thomas Burkholder's TBinder work (in the TTools mini-example) and VNP Software's UIBinder. These are nice because they integrate well with the pure Objective-C model that IB promotes. Unfortunately, these are still too limiting for my work. A different approach would be to take some interpreted language which lends itself to embedding and try to shoe horn it into the IB paradigm. That's the route I've taken.

Tcl

Tcl (pronounced "tickle") stands for the "tool command language", an embeddable extension language designed by John Ousterhout at the University of California at Berkeley. The best way to learn about tcl is to pick up the Addision Wesley book John wrote (coming out sometime in early 1994, preprint available in PostScript form from sprite.berkeley.edu). There is a great newsgroup comp.lang.tcl, which spends most of its time with issues surrounding Tk, the X toolkit based on top of tcl. I've shipped version 7.3 with these palettes, which, at the time of this writing, is the most current version.

I've been using tcl as a programmer for five years, and think it's great. It has some problems, the most notable one being that there is no compiler, but it's advantages (amazingly well supported, portably written, rock solid implementation, great documentation, fast enough for what you should be using it for) far outweigh its problems.

One problem that I've always had with IB is how its target/action

paradigm promotes a separation of a UI object's state dependencies from its actions when used. In other words, you might have a slider which, when grabbed, sends an "updateTemperature:" msg to some object. But who updates that slider? Well, the IB answer is that some other object has an outlet which is connected to that slider, and it's responsible for sending messages to update the slider when (say) the temperature changes from some other source. There is *no* way to look at that slider and see where it's state comes from. This has driven me nuts for years, but until this summer, I didn't really have a clear idea of how to fix it.

Tk, which is the X toolkit which uses tcl, promotes a different UI paradigm that I call the "database" paradigm. The idea is (using my temperature slider example from above) there is some tcl variable called (say) "temperature" which slider manipulates (by sending a message to the tcl interpreter to set the value of the variable temperature) and that the interpreter takes care of sending the

slider a message to update it's value whenever the variable "temperature" changes *for any reason*. What's especially neat about this is that you can inspect the slider widget and all the info you need is there: what message does it send whenever it gets frobbed, and what is responsible for it's value.

I basically took this idea and extended it to allow a UI element to have it's value based on an arbitrary tcl command. The simple case of course is supported (value based on a single variable), but I think this arbirtrary command notion is pretty powerful and useful. Whenever any of the variables in that command change, the tcl interp automatically tells each UI element that depends on it to reevaluate themselves in the current context. Note that this isn't foolproof; I only do a one pass check through the tcl command for variable names, so if you're hiding a bunch of global variables in some procedure name, I'm not going to recurse down through to find them. This isn't lisp; it's tcl...

This solves a lot of really complex problems (see the

WW3DPalette and my "malleable media" stuff), but the easy win is it obviates the ubiquitous "Controller" class that everyone writes for stupid little applets. For example, take a look at the "Calculator" example in NeXT's IB documentation. Then take a look at my version in ../Examples/WavesWorld/TclCalculator. No stinkin' linking, that's what I say...

Anyway, I realize I need to document stuff more, but play around with the calculator demo, then go play with the examples in the WW3DPalette, and then try to come up with your own examples. I'll be happy to help; I need more examples, and documentation gets easier to write the more examples you have to point at.

go play.

- wave