**The article from this draft was published in 1995 in Spectrum Magazine**

# GUI or OOUI?

by Huey, Loouie, and Dewey[1]

**This is yet another in the twenty year series of articles to the Pick community by Chuck Thomas, whose object is, as always, to make your life better through information, obfuscation, mysticism, innuendo, and gross generalities about current and future technologies. Some of what follows may be true.**

***"One man's ceiling is another man's floor."***

OK, so now that you've got the entire organization committed (or they have your acceptance) to a new round of **GUI** applications to replace/supplement/enhance your legacy Pick applications, consider this: what you have developed and/or are developing may already be obsolete technology. The GUI era is rapidly coming to its end. *What?!?* Does this mean we are returning to old character interface? Nope. It means that we are into the **OOUI** (rhymes with screwey) era.

An Object Oriented User Interface ("OOUI") is best defined by example. Notice, by the way, that GUI isn't defined in this article. It's like the seat belt instructions on the plane: if you don't already know, what the Hell is the use? OOUI, on the other hand, does deserve explanation. The basic thrust of an OOUI is that it is the next step in the paradigm shift of computer power to the ultimate user.

In character applications ("ch-apps"), the programmer determined the sequence of events for the user; in GUI-apps, the user determined the sequence of events, but the programmer determined results of each action; in OOUI-apps, the user determines the sequence of events and the results are contextually determined. A simple example of contextual determination is drag-and-drop: if you "drag" (e.g., point to an object and hold the mouse button down while moving the mouse across the screen) and drop it (i.e., let up on the mouse button) in a file folder, it contextually means either "move the file" or "copy the file"; if you drop it on a trash-basket icon, it means "delete the file".

GUI has dominated the early 1990's: Windows/3 has become the dominant desktop environment of business and GUI desktop applications have become the standard for excellence. It is hard to buy a computer without a GUI pre-installed. Last year alone, Microsoft sold over twenty million copies of W/3, half of them in the US. Yet, Microsoft is working as hard and fast as it can to introduce the replacement of W/3. "Chicago" (now known as Windows/95) is already in the hands of software developers, and is the replacement for W/3 in Microsoft's grand strategy for the desktop; NT, and its progeny "Cairo", is the server in their vision. Is Chicago really the better Windows...and, if so, why? Windows is a GUI environment; Chicago and Cairo are Object Oriented environments, as is IBM's OS/2. The Mac, since you asked, resides a little south of OOUI, in the village of West World.

---

[1] With apologies to Warner Brothers, but it rhymed so well I just had to do it.

Incidentally, IBM is spending over $50 million in its current advertising campaign for Warp (the marketing moniker for it's latest OS/2 release). IBM has, over the years, sold a total of seven million copies of OS/2; and the end-user street price of Warp is about $49. If IBM sells another four million copies at $49 each through this campaign, and realizes twenty-five percent of the gross, they will just about pay for the campaign itself. Noodles. They are noodles. IBM is acting like a suburbanite fumbling for his car keys on a dark city street.

A question arises: "What's the diff[2] between OOUI and Object Oriented Programming[3] ("OOP")?"; as does another question: "Can character applications be Object Oriented?". An answer to the first question is that OOP is an underlying technology behind OOUI, and non-OOUI applications can be written with an OOP, but OOUI applications are very difficult to write without an OOP. Popular OOP's are C++, PowerBuilder, and SmallTalk. The second answer is...I forget the question...oh, yes character applications can be Object Oriented, but they do not have an OOUI. As in everything else, there are few crisp borders here, and characteristics of OOUI's are found in GUI's, and non-OOP environments acquire OOP characteristics over time.

Three of the defining characteristics of objects are inheritance, class encapsulation, and polymorphism. The fourth is... I dunno. Inheritance is the ability to abstract common behavior from an ancestor object and then specialize that behavior in a descendent object. Class encapsulation is the arrangement of applications into objects or classes of objects, defining the properties and behavior of the classes as the principal organizational basis for programs. Polymorphism can be illustrated by one of its underlying principals: name overload. For example, you know how to use "add" to act with a variety of operands. You can add 2 and 3; you can add 2.09 and pi; you have to imagine that it can get a little more complex.

Object-oriented programming is largely declarative, rather than procedural. That is, by declaring the state of the object, or something about its state, its use is determined. This leads to simpler (read "faster & less expensive"), more reliable code. This assumes that there are objects to begin with, which is a very large assumption.

**Power to the People**

My theory is that as a technology matures, its ownership shifts from the innovator to the consumer of the use of the technology, and that ultimately the real understanding of the technology is not with the original innovator, but with the user. Therefore, improvements to the technology most often originate with the user, and are demanded by the user. The Pick industry segment is surely conscious of this concept. In our economic system, the user votes with their dollars in the marketplace.

The short history of the computer industry has followed this theory concisely. At first, computer companies were vertically integrated, making the entirety of their systems. Only the gurus knew how to train the beasts to do useful work. Over time, printers were generalized, then terminals, then tape drives, then disk drives, then memory, and

---

[2] sic...the kind of person who would ask this question would, in fact, speak in abbreviations.

[3] OOP, along with OOA and OOD are generally attributed to Coad, Yourdon & Nicola.

finally even the processor chips themselves. Software has followed suit: first tools, then applications, and finally even environments have become independently available. This all represents a shift in power from the producer of the computer to the ultimate consumer of the resource; instead of relying on your manufacturer for improvements at their pace and cost, you can acquire hardware and software components independently and put together your system your way.

What makes this shift work is "standards", and that is what you vote for. You might think that it costs more for a producer of hardware or software to comply with standards, rather than doing things their own ("proprietary") way, but amazing as it may seem, as products become increasingly standardized, they become less expensive. Go figure. And just because the products meet a complex web of standards, each producer still manages to develop an edge, some advantage that provides benefit to the consumer. The edge might be raw performance, lower cost, more functionality, easier use, quicker installation, better support, prettier pictures, or no cholesterol; whatever you perceive to be better.

So it is with the user interface. At first, our interface was that we filled in forms and submitted them for keypunch (well, some of us remember that); then we typed on teletype model 33's at ten characters per second; then we had glass teletypes, and our applications gradually migrated from unreadable to forms-like; then we had immediate editing and help messages. All of this suffered from the fact that someone else dictated how we did business. I mean, some kid two years out of college was formulating how buildings full of mature adults spent their days; you know, the programmer ruled. Now we are in an era where the user rules. Strange how computer power shifted to the people just as Marxism/Leninism went terminal.

It is difficult to mention "standards" without discussing "integration" briefly. The cost of open systems has been integration. As a computer user, you have always paid for integration; the cost of integration of a closed (proprietary) system sometimes was that it was expensive to make what you had work with something else. Contrary to every consultant you will talk with, I advise you to not worry too much about integration. We are in the youth of open systems, and the computer industry is just beginning to understand your need for "plug 'n play". It is happening fast. The LAGS (Latest And Greatest Stuff) already supports OLE 2, which provides a means for ultra-integration of applications. An example of this is the auto-launch of an Excel spreadsheet in a Word document, totally seamless in the end product document.

**Developing OOUI Applications**

Two kinds of application development projects are those which extend the functionality of an existing application and those which produce new applications. Some of the most used and useful applications in your business have been around a long time, and have evolved as your business needs changed. If a long-term application needs some major surgery, sometimes it is better to view it as a new application. The benefits of doing this include the obvious opportunity to refresh technology and the not so obvious re-thinking of the application, knowing all that you do now but didn't when the thing was originated. The point is, sometimes it is better to toss it than to turn it.

Today when you start an application development effort, you must decide what development environment you will use to develop the application. This is more than a toolset, it is a framework and a mindset. You might begin by defining your development methodology[4], which will determine how you will develop the application. Some places worship at the alter of methodology; others ignore its existence. The most successful large scale client/server development efforts generally follow an approach called "RAD", Rapid Application Development. The RAD approach is based upon building a logical data model first, physicalizing it, and then building a series of iterative prototypes. Old-style development sought to define and detail design everything prior to any implementation. Follow some numbers: suppose you wish an application that meets 90% of your user functional requirements. It might take six or eight months to cycle through user meetings, design specifications, and more meetings, before the spec covers the 90%. Then the construction of the software begins. With RAD, first you model the business data, and then you may prototype to meet the 70% level of the requirements initially, but quickly. Meeting with the users, you flesh out a second prototype to meet 70% of the remaining requirements. Finally, a third iteration achieves 70% of the remaining requirements. You end up with (70% + 70% * 30 + 70% * 9) = 97% of the requirements in three simpler iterations, not having to know everything before you began. And you can do it, presumably, with less "Senior" labor. The project is done faster with less cost and meets a higher level of functional requirement. The users actually are able to see and play with a navigational prototype early in the game, and their enthusiasm and participation remain high.

Software is like bacteria; you can't normally see it, but it'll generally make you feel bad. Also, it constantly grows and changes, adapting to new environments. Sometimes it is passed from one host to another. Viruses are a whole other thing, however. We have to have some bacteria (software) to be healthy...like that which helps process our food (business information). Anyhow, since software grows and changes, since the its use changes even when the software itself does not, object programming provides many long term benefits. I knew I could tie that in, somehow.

So, whatdya say OOUI was, again? I didn't says what is was, I said what it was like. Here is how OOUI effects my work... at ICON Solutions, we are doing a great deal of PowerBuilder (PB) development, and PB is an object-oriented development environment. We have developed what is known as a Base Class Library (BCL) to support our development.[5] The Base Class Library is a library of re-useable business objects that enable us to rapidly prototype, develop, and deploy feature-rich software. The BCL gives us a well-developed infrastructure out of the box. We are now developing another level of objects for a specific industry's applications, an Application Class Library (ACL). The ACL is build over the BCL, and gives an organization the ability to rapidly deploy new, "customized" applications, without the requirement to build the application software functionality.

Let's say you want a Sales Order application. On that application there is a place for an item's price. When building the pricing object, and the data model to support it, one

[4] There are many Object Methodologies...Booch; Coad ,Yourdon & Nicola; Class-Centered Modeling; Demeter; Graham/SOMA; and others.

[5] ICON Solutions incorporates its Enterprise Builder library in the PowerBuilder software development projects it performs for Fortune 1000 companies.

might have, simply, a single retail price. Over time, various user needs broaden the functional requirement...pricing might be step-wise by volume, aged, based on order mixes, vary by location; you know all kinds of pricing mechanisms. By changing the price object as needs change, the Sales Order application itself inherits the enhanced functionality.

Imagine a software vendor with a vertical industry ACL. For a given investment, the software can be developed broad and shallow or narrow and deep. By building broad and shallow, the vendor can approach a larger market with a seemingly more robust product. Functionality of the product can be grown as it is needed to meet a specific requirement.

What about where you are? Tomorrow starts later today, so it's not too late. Don't continue to make what should have been good yesterday meet yesterday's standards. Don't build for today. Build software for tomorrow, but build it today. You have a vision of what things should be. Set expectations; meet the vision. Then don't concentrate on the risk of failure; concentrate on the risk of not succeeding.