"Slap a GUI on It"

Your M Future Will Be Easier if It's Object Oriented

by Erik Zoltán

Can M evolve to use Windows effectively? Yes it can, but the rapidly increasing complexity and more exacting user demands in this high-productivity environment have placed an entirely new kind of challenge on the language. To meet these challenges, M developers must be able to incorporate the best features of environments like C++, Visual Basic, Delphi, and Java without sacrificing the traditional strengths of M—its power, flexibility, and simplicity.

Of course, the current "stampede" to Windows has raised concern, among some, about the future of M. Originally designed as an operating system, most modern M systems have an arms-length relationship with the host system, which has been viewed as something essentially foreign. Now that the computer industry is converging, with a single operating system poised to achieve clear dominance, many are wondering about the long-term ability of M to exploit the inherent power of the graphical user interface (GUI).

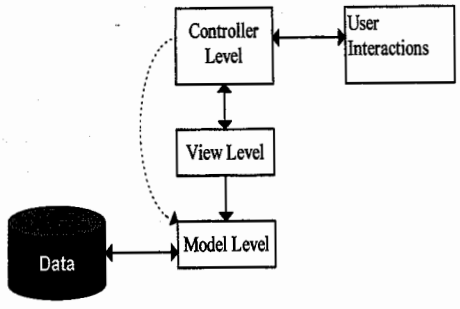
But the great virtues of M are obvious—persistent, self-organizing sparse arrays for convenient, efficient data storage; the ability to create generalized software using indirection; an admittedly cryptic but nonetheless extremely powerful language syntax. These virtues have led to a rapid development process in an unusual but simple, easily-understood environment. Surely a dynamic, flexible language like M can adapt to meet the challenges of Windows, just as old-fashioned languages like C, BASIC and Pascal have shown a remarkable ability to transform themselves.

In the past few years, Windows has become increasingly object-oriented, and in the future will be even more so. This critical fact points the way to a clear path for the continued evolution of M. In object-oriented (OO) systems, objects assume a similar "arms-length" relationship with each other. They communicate by sending messages in a clearly-defined, consistent way. A number of fundamental changes are needed to make M object-oriented, and an ANSI subgroup has made a great deal of progress on a solid OO standard for M.

In addition to its virtues, M has always had some drawbacks. Lacking a formal model of application development, and imposing few restrictions or requirements upon the developer, it has always been too easy to create un-maintainable code in M. Of course, no new paradigm can force programmers to write better code, but this article will argue that the Model/View/Controller paradigm, when combined with the OO model, will definitely make it much more likely.

What is the MVC Paradigm?

Without getting too technical, the Model/View/Controller (MVC) paradigm is a way of constructing software that separates an application's capabilities into three separate, independent levels.



The ***model*** level, at the bottom of the above figure, is for representing data. In traditional M terms, this refers to global databases, such as patient and physician files. In M, any piece of data, in any global, may be created, modified, or removed by any M routine. The consequences of this are well-known: if a program bug causes invalid data in a patient record, it may take a very long time to correctly isolate the problem.

In an OO system, by contrast, patient objects communicate with physician objects (and others) by sending messages, and all the code to modify a patient object's internal data must reside within the patient object itself. Thus, if a program bug causes invalid data in a patient object, then the patient object itself is ultimately responsible for preventing this.

If the model level represents the back end, then the ***controller*** level is the front end, consisting of interactions with the user. In traditional M terms, this refers to the code directly concerned with input/output operations. In an OO system, it consists of objects on the screen displaying and collecting data, and allowing the user to trigger certain events within the application (e.g. by clicking on a button.)

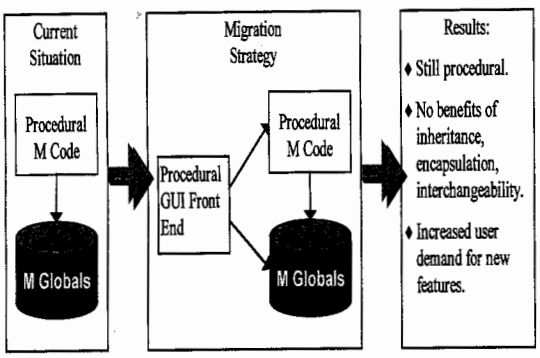
The *view* level provides an interface between the model and controller levels. In producing a report, for example, a view-level object would drive the entire process, interacting with model-level objects (representing the data) and controller-level objects (displaying information on the screen). The view level is generally in charge: it sends messages to the model and controller levels, thereby driving the process. The controller level consists of the concrete user interface, while the view level is an abstract presentation that could just as easily be printed on paper, as a report, as it could be displayed on the screen using controls.

The MVC paradigm suggests that these three levels must be kept ***separate.*** Model-level objects are responsible for representing the back end, controller-level objects are responsible for the front end, and view-level objects are responsible for overall control of the process, integrating the two other levels. If a model-level object attempts to send messages to objects at other levels, then the MVC paradigm has been violated. It is technically permissible for the controller level to communicate directly with the model level, but to do so is considered bad form.

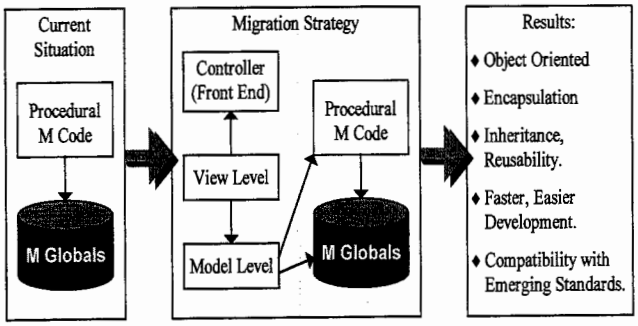
In traditional M systems, it is possible for any routine to modify any global node, ***and*** to perform I/O operations. That's why legacy M code ***consistently violates*** the MVC paradigm! It's possible not to, in M, but doing so requires tremendous discipline. The temptation to mix the three levels together within a single routine can be very hard to resist. (Note to M programmers: a single FOR loop, using $ORDER to traverse the database and containing WRITE commands, actually spans ***all three levels***: FOR = view, WRITE = controller, and $ORDER = model.)

Two Migration Paths

There are two Windows migration paths one can choose, as illustrated in the following diagrams.



By migrating to a procedural GUI front end, developers continue to blur the line between the three levels of an application. The result, of course, does not benefit from the strengths of object-oriented M. Even so, there is an increased user demand for new features. As more such features are added, user demands tend to increase (not decrease), and the procedural code is subjected to increasing stress.



When the migration strategy combines object orientation with the strict division into model, view and controller levels, the benefits are many. Encapsulation makes it easy to know which code is responsible for which results. Inheritance makes the code much more reusable. This means that future development, to meet increased user demands, will be faster and easier. And, since future standards will also be object-oriented, it will be much easier to maintain compatibility with them.

Encapsulation and Interchangeability

In object-oriented M (OOM) systems, a principle called ***encapsulation*** forbids one object from directly "looking inside" another. Think of each object as a little capsule: you can't reach inside without breaking it. Instead, objects must exchange information by sending messages. This gives each object a ***clearly-defined role***, and makes the three-level MVC paradigm easy to enforce.

One advantage of the MVC paradigm is that it makes ***interoperability*** easier. The model level might include your M global database and some of the existing routines; the controller level might be a front-end written in Delphi, or a native OOM front end; and the view level might be an OOM system functioning as an interface between the two.

Another advantage is ***interchangeability***. Theoretically, it should be possible to replace all of the controller-level objects with new ones, thereby implementing a different user interface. As long as the new controller level supports the same messaging protocols, the model and view levels should not require significant changes. (But generally there are changes to the messaging interface. Even then, only the view level will have to change in order to support it: the model level should not be affected unless the system's overall data modeling needs have changed.)

In traditional M systems, this interchangeability is not readily apparent, because the controller, view, and model levels have all been mixed together. Thus it often seems easier to rewrite an entire M program than to isolate and modify all of the controller-level commands. Imagine taking a terminal-based M application and trying to incrementally modify it to use a GUI interface instead. Would such an ***incremental*** migration path even be possible?

Or imagine hard-wiring a prototype interface between your M database and a Visual Basic front end. If you decide to redo the front end in Delphi, what percentage of your M code to support the VB front end would be able to work, interchangeably, together with Delphi?

The GUI migration path often envisioned by organizations needing to make this transition does not involve OOM. This results in a continued blurring of MVC lines. Thus, ***interchangeability is never achieved***. At considerable expense, a new GUI front end is developed that represents a hard-wired set of interactions between M and, for example, Delphi or Visual Basic. Thus, each new feature can require coding changes to both the front and back ends. A complete overhaul of the front end (which is almost inevitable) may result in scrapping the whole thing and starting over. Worse yet, the prospect of losing so much work may prevent the needed overhaul, resulting in an application that does not meet the needs of users and is also difficult for programmers to work with.

The procedural solution is a "one-trick pony." The work involved is generally not reusable. Finding bugs, making changes quickly and easily, and customizing code becomes problematic. And, because there is no encapsulation and no inheritance, each change produces a ripple effect, resulting in chronic support problems. If I don't do it right the first time, I'll have to work much harder to fix it, later on.

Of the GUI systems that I've seen M developers touting in recent months, too many of them fail to meet the criteria of mature Windows applications. Interactions with their front ends tends to be very limited, and they rarely capture the power and flexibility of M.

The OO Future of M

I have been working in OOM for quite a few years, now, and it seems like every few months a new door opens up, making it easy to create some previously-unthinkable Windows application in M. Because it's object-oriented, my code is nearly always reusable, and old classes of objects are continually being recycled in new ways to meet the needs of the next project. And if I go back to an older project, whose component objects have been upgraded since I last used it, then I may be pleasantly surprised that useful new features are already in place, or are now easy to add. If the procedural solution is a "one-trick pony," then the OO solution is more like an "energizer bunny." If the procedural solution is a shallow, temporary solution, then the OO solution is a deep, abiding solution.

And, because of inheritance, I can add a new feature in many places at once, producing a sudden renaissance in the capabilities of a number of related kinds of objects. If I don't do it quite right the first time, I can always fix it later, and the benefits will be easily reusable throughout the system.

The difficult challenge of migrating to Windows also gives us many exciting new opportunities. We can no longer afford to accept compromises we took for granted in the past, because the applications of the future will place tremendous new demands on the work we are producing today. As an OOM programmer, I can be confident that the work I do will remain useful in the future. And I want this work to survive, because otherwise I'll only need to re-create it later.

The benefits of OOM are clear: it is easier to find bugs, since all code is self-contained within the appropriate object. It is easier to make changes and customize an application, since internal structural changes to one object do not affect other objects unless their communication patterns are altered. OOM code is often better written and more structured and nearly always more modular. Because of inheritance and reusability, each application is easier to create than the last, and the benefits of OOM become more apparent as an application's complexity increases.

The M language is complete for procedural programming, but no longer adequate in the object-oriented world of the present and future. A great number of companies in the M community have already begun the transition to objects, but in the majority of cases it is an ad hoc, partial transition. We need to fully embrace OOM and develop a long-term strategy to migrate our existing systems. This argument has started to win acceptance in the M community: that's why ESI has now been awarded a DoD contract to provide TCP/IP, DCE, and CORBA connectivity to our M-based EsiObjects™ programming system. This exciting new contract will open up M databases to the next generation of data interchange standards.

I think that OOM represents a clear path to the future. The creation of ***fully*** object-oriented M systems, including classes and multiple inheritance, a robust and flexible messaging syntax, encapsulated objects whose integrity can't be violated from outside—all of these are making M a better language on its own. Combine this with full support for the most advanced features of Windows, and an M application is able to send messages to an object without needing to worry whether that object lives inside the M environment, or whether it exists at the Windows level (for those of us using OOM today). When the power of M is combined with this level of interoperability and integration with other applications and the operating system, then M becomes a very attractive development environment for the 21st century. **M**

Erik Zoltán is a freelance consultant who has been programming, writing, and teaching in the M community for the last 6 years. He has also done extensive work with EsiObjects™ for ESI Technology Corporation.
