Evolving M in Object Technology

by Jerry Goodnough and John McManamon

Recently, there has been a rising tide of awareness and discussion about object-orientation (OO) and its place in the M community. We certainly are pushing along this awareness not just as part of the work we are involved in, but in a belief that the object-oriented (OO) paradigm is a necessary evolutionary step for any technology, whether it is M or COBOL or C. Articles are appearing about OO technology explaining what it is and what it does. There are many books and classes that teach about OO concepts. Here, we would like to explore a fundamental question for M, based on the premise that M will be shifting toward the use of the OO paradigm. This shift is inevitable because the science of software engineering is evolving in that direction and has been for many years. Basic business needs and requirements compel us to move in that direction. Beyond that point, however, we must look at the role M should play to take advantage of the power of OO. This is where uniform agreement may not come so quickly. Specifically, we need to ask whether M should become an *object-using* language or an *object-defining* language. This distinction, which will be explained, is a significant one since we use it to raise some important questions and challenges that M faces based on how we answer that question. In essence we are asking "Do we want M to be a 'mainstream technology' à la C++ and others?" We must come to grips with this question and be ready to accept the consequences that our response will force upon us.

Object Usage

When discussing object-oriented languages, a crucial distinction is whether the language *uses* objects or has the additional capability to *define* objects. An object-using language is one that has functionality to connect to and communicate with other objects. This type of language is along the lines of a component-based language (such as Visual Basic) that provides us with a set of objects we can create, use, and destroy at run time. These objects may include presentation layer components such as controls and containers, or they may be *worker objects* that provide a service. Examples of these types of objects include collection objects, or sorting objects that accept data and sort it on some primary key.

This does not assume that just because we can talk with (use) an object, that the object can talk with (use) us. Fundamental questions to ask in the case of object-usage are what role should M play in communications with external objects and how is this role realized. Is M seen from the outside world? And if so, is M seen as just one large object, or can we achieve finer granularity?

Since an *object-using language*, by definition, only uses objects and does not define them, there is no modeling of the data "as objects" within the language environment. The system data and functionality not provided by the components will follow the traditional paradigm of data and procedures as distinct entities, with all the traditional problems those entail.

Object Defining

An object-defining language is one that not only communicates with other objects, but has the additional capability to define *new types* of objects. There are alternative technologies to provide this definition. Traditionally, object-oriented languages have used Class structures to manage the complexity of object modeling. These structures provide a basis for reusing fundamental portions of the model through inheritance and subclassing among others. With the power to define new objects, we can create systems that reflect the actual business model directly. The ability to model the business directly, leverage existing knowledge, and code through reuse are formidable arguments for pursuing object-definition.

The impact of the *object-usage* versus the object-defining approach is greatly felt in fundamental systems analysis and design. Working with object-usage, designers develop an architecture using components that other people have produced and will try to fit those components into the system as best they can. But the essential business model, which is unique to the environment, is not affected in any significant way. The designer is left with external components fulfilling certain roles and core system components remaining as traditional, partitioned code and data.

continued on page 28

When using an *object-defining language*, the designer can use other components, but truly can build the core business model into the design because of the ability to define new objects. Thus the system implemented from this design more fully expresses the business model and adapts well to changes and modifications over time. The objects purchased or produced can include large composite framework objects as well as small component objects.

Making M an Object-Using Language

Being an object-user has certain advantages. The more objects we can communicate with, the better. We may be able to make use of many object features and open ourselves to the outside world. In terms of using and realizing the other advantages of object-orientation, however, we haven't gone very far. Our data are still separate and are not encapsulated. Data and functional relationships are hardwired into the system with all the corresponding dependency nightmares. Our code is not any more reusable than traditional M code. Our systems then simply become traditional programming platforms with the additional functionality to use externally defined objects.

Some may argue that this is not all that bad. It may be that object-usage is a proper role for M to play. It allows M to take advantage of component-based software without laying the burden of the OO paradigm on the backs of implementers and designers (not to mention the MUMPS Development Committee). This may be a reasonable alternative, but to make this decision, we must have a clear delineation of the strategic advantages and business reasons for doing so. The case must be convincing as to the benefits for M in evolving in that direction. We also must understand the risks entailed in limiting M to the role of an object-using environment. Are these risks outweighed by the benefits?

If users are to rely solely on object-usage via external interfaces, we argue that M may fall into the same fundamental problems encountered with the M Windowing Application Interface (MWAPI). While it is an extremely elegant solution, in the M tradition, by supplying simple hooks to external and high-level interfaces in the Windows environment, the MWAPI stops short of allowing access to the lower-level, internal functionality. This capability would have made the MWAPI more than just elegant: It would have made it truly useful in the sense of establishing a fully capable environment for Windows development.

In becoming an *object-using language* we risk the same fate. Users will have useful hooks to the higher-level functionality exposed by the objects they want to use, but the myriad aspects of functionality that are not exposed (yet critical to their being able to fully use them) will be out of reach. Therefore, we argue that both levels need to be addressed: the higher-level abstractions as well as the low-level interfaces that are not exposed.

We must further ask, "If M is going to be an object-using language, is M then a reasonable environment for object users?" If we want our technology to grow and be widely accepted, we must not only decide what role we play, but what our world will look like after we have committed to that vision. Will those who are interested in searching out new and efficient technologies be enticed to go with M Technology? Or are there other alternatives that provide a better object-user development environment? If we cannot demonstrate that ours is the better technology, then we will be relegated to no better role than that of a data repository with the technology experiencing limited or stagnating growth potential.

If we are going to limit the role of object-orientation in M to simple-component usage, we had best be prepared to live in that world and take steps to improve our position as an object-using environment with respect to other technologies.

Making M an Object-Defining Language

On the other hand, if we decide that M's leverage needs to couple powerful database capabilities with the full potential of the OO paradigm, we must likewise look at what that vision entails. While we believe this is the path M must take, we don't hesitate to say that it won't be without some cost. If we are going to incorporate the power of the paradigm in the language, then we must invest in the training, tools, and knowledge that the paradigm requires. If we will have the power to model our systems, we must have the knowledge to use that power wisely. This means opening up the traditional M programmer's view of the world to include an understanding of the object-oriented framework and apply it to the problem at hand. This includes not just learning concepts, but understanding object modeling and design tools, among other topics. It must be noted, however, that though an object-defining language extends the level of complexity in systems, it allows a multitiered approach to solving the issues of that complexity. Work can be leveraged across the organization by specializing expertise within certain areas, for example class builders and application developers. An organization may take advantage of work done by other

Because of our work and experience in this area, using the reasons cited above, we hope to provide a compelling argument for adopting an object-defining language instead of merely object-usage.

organizations as well as inhouse development that can be reused and extended. The organization does not have to invest in soup-to-nuts training for every employee and thus should not be intimidated by the complexity that object-defining brings to the table.

Besides investing in the necessary brain-power, we must also look at what M must do in order to become a player in this arena. If people are looking for an OO-development language, why should they choose M over another language? In order to meet this challenge, M must be able to participate in the outside world in a client-server environment. Can M participate with the outside world as an equal, or must it have control? What support is needed from M to support its role as a client? As a server? Dynamic Link Library linkage and communicating data with different data types are likewise key issues to be resolved.

Conclusion

There are many concerns involved that will not be easy to solve. But if we do not offer the necessary services to open the M environment to the world at large, we risk the real possibility of M becoming less and less useful to people. We need to look at these concerns now, as we start down the road to object-orientation and the future of M, and decide on our vision.

Because of our work and experience in this area, using the reasons cited above, we hope to provide a compelling argument for adopting an object-defining language instead of merely object-usage.

Whether it is *object-usage* or *object-defining*, we must be willing to accept the consequences of that vision and be ready to handle those challenges in order to keep M viable and growing strong into the next century. M

Jerry E. Goodnough is a software development manager and chief architect of ESI Technology Corporation's product called **EsiObjects**.

John A. McManamon is a senior software engineer at ESI Technology Corporation and has been responsible for Object Technology instruction and **EsiObjects** development.

ESI Technology Corporation is located in Natick, MA, 01760; telephone 508-651-1400, Fax 508-651-0708.