COMPUTING INTEGRITY
INCORPORATED
60 Belvedere Avenue
Point Richmond, CA 94801-4023
510.233.5400 Sales
510-233.5444 Support
510.233.5446 Facsimile

PROGRESS SOFTWARE BUSINESS ALLIANCE
ISV Partner

OERA Strategies: Object-Oriented or Not?

Version 30 April 2007

Thomas Mercer-Hursh

Introduction

The OpenEdge Reference Architecture (OERA) describes an overall structure and pattern for the architecture of modern applications, but does not in itself specify any particular implementation. Progress Software (PSC) has generated a series of whitepapers over recent years which discuss various aspects of this architecture with some sample code, but this code is limited in scope and there is no pretense of it illustrating production-ready techniques. More recently, PSC introduced AutoEdge, an "application example" meant to illustrate one way in which OERA principles might be applied in a real application, but it too is built on a number of simplifying assumptions and does not pretend to offer production ready code on which customers might build real systems without significant further work.

With the exception of three very recent whitepapers, all of the sample and example code has been built using procedures, albeit with some "imitations" of object-oriented (OO) design principles. However, since OpenEdge release 10.1A over a year ago, the Advanced Business Language (ABL) has contained true OO capabilities. This raises the obvious question, particularly since the broader world of software design patterns and tools is dominated by OO, of whether future sample and example code should utilize the OO features in ABL.

The three recent whitepapers and associated webinar dealing with using OO for OERA suggest that PSC believes that some form of guidance on using OO in these contexts is an appropriate goal to which PSC should commit some resources. However, the models presented in these materials do not conform to generally accepted OO design principles. Instead, they present a sort of hybrid model with some OO characteristics, but which also makes use of ABL features for handling data in a relational fashion, rather than in an object fashion. I have questioned the desirability of this approach in forums on the Progress Software Developers Network (PSDN) and it is the intent of the current writing to explore this question more fully.

Questions

There are a number of interrelated questions which arise in examining the question of whether and how to use OO in OERA sample and example code. These include:

1. What are the pros and cons of the various ways in using or not using OO for samples, examples, or production code?
2. Should PSC be selecting a best practice strategy from among these options to advocate exclusively going forward or should it create materials using multiple strategies?
3. Should there be an "AutoEdge 2.0" which incorporates lessons learned and possibly illustrates a different strategy than the one used in the initial AutoEdge?
4. Is it desirable for PSC to attempt to create production quality examples or even framework components or should PSC confine itself to the current level of samples and examples?

A complete treatment of all of these questions exceeds the scope of the current writing, but I believe that at least some consideration of all of these questions is appropriate while focusing on a comparison of these approaches.

The Strategies

For ease of reference, it is convenient to give each of what I see as three basic strategies a name. Since the factor that distinguishes these strategies is their use or non-use of OO programming methods and principles, the names I am proposing logically relate to their use of OO. These are:

- NOO: Non-OO, i.e., a strategy which uses only ABL procedures, not classes, although it may use some OO terminology and may imitate some OO concepts.

- ROH: Relational-Object Hybrid, i.e., a strategy in which programming is in the form of ABL classes and using OO principles such as inheritance, but in which data is expressed in the form of temp-tables (TTs) or ProDataSets (PDSs) which are passed between objects, rather than having the data encapsulated in an entity object with its behavior.

- TOO: True OO, i.e., a strategy where there is a traditional OO encapsulation in which data and behavior are encapsulated into entity objects and it is the object, not just the data, which is passed between layers.

NOO

The bulk of existing OERA materials from PSC do not utilize any OO language constructs. There is some imitation of OO terminology and thinking, but in a fashion consistent with ROH, i.e., data is communicated between the data access layer and the business logic layer through ProDataSets. Because this strategy uses no OO language constructs, it is more familiar to the general ABL programmer and to a large extent it is a strategy which does not rely on the latest release of ABL in order to be implemented. This later point is possibly a bit deceptive, however, since other language enhancements such as `REFERENCE-ONLY` are also limited to recent releases and play a significant role in current implementation models, so it is not as if the current models can be backported to version 9.x environments without change. There is also some question about the familiarity which most ABL programmers have with ProDataSets as these have not been widely used in the general Progress community.

Superprocedures, which have often been cited in terms of their possible virtue for imitating OO capabilities, have a certain simplicity of use compared with objects, e.g., fairly simple disciplines or managers can result in an effective singleton with little or no actual coding and a superprocedure reference does not need to be created in each procedure where it is referenced in the way that is required for objects. The flip side of this simplicity, of course, is that their use is less deterministic and more difficult to model and has no compile time checking to insure correctness. Encapsulation is entirely a question of programmer discipline and use.

Passing of PDSs from layer to layer requires compatible temp-table and PDS definitions in both sender and receiver, which tends to require the use of include files for these data structures in components from multiple layers, leading to poor encapsulation and tying of multiple components to a single data structure. The big downside of the NOO strategy, of course, is that one doesn't gain any of the benefits of the OO language features including strong typing, compile-time checking, and strong encapsulation. Moreover, one needs to adapt and interpret design patterns from the OO

world rather than using them directly and one cannot fully utilize the capabilities of OO modeling tools in design and generation of the system.

ROH

The newest materials on using OO for OERA utilize what I have been calling a relational-object hybrid strategy in that they use OO ABL constructs such as classes and inheritance, but data is held in relational-type structures such as PDSs and is passed from object to object, layer to layer, in much the same way as in the NOO strategy. The concept behind this hybrid approach is that ABL is strong in providing constructs for dealing with relational data and that this strength should be exploited in any OO implementation. Like similar NOO handling of data, one can argue that this is also a more familiar approach to handling data for most existing ABL programmers.

The intuitive objection to this strategy from the perspective of those familiar with OO principles is that the data is not encapsulated, but rather is in a very exposed data structure. While this local data structure can be abstracted to some degree from the stored form, e.g., it can be appropriately denormalized, the structure of the data is exposed wherever it is used in exactly the same way as the structure of the database is exposed in older code that accesses the database directly from all components. Thus, any alteration in the structure of the data may impact every component that uses it, unlike OO encapsulation where any amount of change may be made to the data structure internal to the object without impacting any component using the data, as long as the contract of the object is preserved.

It can be argued that the relational PDS structures of this strategy are easier to work with than having to create methods to accomplish equivalent behaviors because these structures are can be accessed by familiar ABL relational language constructs such as `QUERY` and `FOR EACH`. However, this also means that one is embedding the relational logic of access in all of the places the data is used instead of centralizing the method of access in a single entity object. Thus, one has duplication of code and fragility and brittleness in response to change. If the data were only ever passed between a data access object and what is called a business entity in AutoEdge and that all other usage employed the business entity, then the possible points of change might be controlled, but then it seems that one would not be achieving the full ease of use which is driving ROH, but instead would be approaching the TOO encapsulation. I.e., if one has decided that passing a PDS is the right thing to do between the data access layer and the business logic layer, it seems likely that one would decide that any transfer of data within the data access layer should also involve passing a PDS.

Moreover, the supposed familiarity and convenience of continued use of relational access patterns may be somewhat illusory since most ABL programmers who are unfamiliar with OO patterns are also unfamiliar with PDS programming. Using the data in the PDS may be familiar, but the techniques of creating it are not.

TOO

One certainly needs to exercise care in adopting OO designs and principles from 3GL OO languages into ABL, a 4GL, because there are advanced language features in ABL which should be used in preference to more primitive approaches in those other languages. A classic example of this, on which I have written previously[1], is the collection class, which should almost certainly be implemented in the current ABL using temp-tables, rather than the primitive hash tables and unordered lists typical of, for example, Java. And, of course, one also needs to recognize that not

[1] See http://www.oehive.org/ExceptionClass

everything done in an OO language is good OO. There are a number of common practices in other OO languages that we should not seek to emulate since they are not good practices from a true OO perspective.

Nevertheless, one of the most basic of OO principles is the encapsulation of data and behavior into objects. This provides a reusable component whose internal implementation is concealed from the components using the object, thus allowing implementation details to change and capabilities to be extended without disturbing any other object. If an object changes internal implementation, no other object needs to change. If an object extends its capabilities to implement new features, but preserves the existing contract, no object using the existing contract needs to change. The only object which needs change, if any, is one that will utilize the new extended capabilities.

Even if data is directly held in the form of a temp-table or a PDS, wrapping that data structure in an object with appropriate methods means that the definition of the data structure needs to appear in only one place. All other components can utilize the data through methods on the object and that object's behavior can remain consistent despite changes in the underlying data structure. This approach also means that data access logic is implemented in one place only, so if there is any need for revision, it is not necessary to search out scattered pieces of ABL in various places in the application, but rather the change can be made only in the object containing the data.

One possible challenge to the traditional value placed on encapsulation in OO is the increasing use of datasets in some OO languages, particularly .NET. However, it seems that the primary role of these datasets is in sending data "over the wire", not in communicating between objects within a single session. It is quite reasonable to want to serialize data for transmission like this and, indeed, there is some evidence that serializing it to XML may have advantages over even sending it in the form of temp-tables or PDSs. Even were it possible to send entire objects across the wire, one would be transmitting a great deal that did not need to be sent since it is possible to instantiate a fresh instance of the object based on the data alone. Moreover, by sending just the data, one also gains the freedom of instantiating a different object with the same data, e.g., a read-only version.

It should also be noted that, while traditional ABL programmers have not been trained in OO, there are a fair number of ABL programmers who have also worked in other languages such as .NET and Java, possibly for user interface code, or who have been exposed to OO thinking in other contexts, e.g., the OO nature of Actuate Basic. Moreover, any younger programmers are likely to have received education using OO languages. So, it is not as if there are no OO resources in the ABL community or that OO is necessarily unfamiliar to all ABL programmers. Indeed, while there is clearly a learning barrier for an existing ABL programmer that does not know OO to learn OO, the barrier to learning the ins and outs of PDS may be even greater because PDS are unique to ABL and there are only PSC materials to learn from.

Summary

While all of these strategies have some apparent virtues and faults, these virtues and faults are not equal. The NOO approach has certainly had validity up through the introduction of 10.1A, since there was no alternative, and probably has some continuing virtue because not every Progress site is on the latest release of Progress and yet every site can benefit by thinking in terms of OERA. The NOO approach is more easily understood by the bulk of ABL programmers and can serve as a source of best practice models, even when full OERA implementations are not being done.

On balance, however, it is difficult to justify the ROH strategy considering the way in which it breaks the fundamental principle of OO encapsulation. While it does allow for handling data in a way that is more familiar, it scatters those data handling expressions through the application instead of encapsulating them within an object. Classic ABL data handling is still appropriate within an object, but effective OO benefits are only achieved when the data is encapsulated with the behavior. Whatever virtues the ROH strategy has are outweighed by its disadvantages.

My recommendation, therefore, would be for PSC to utilize the TOO strategy for all forward-looking materials while considering also continuing the NOO strategy as a supplement, if resources permit. When possible, it would be preferable to derive the NOO models from the TOO ones or at least to evolve them as much in parallel as possible so that those attempting to acquire OO can see the correspondence between the two. Ultimately, this may require revision of the current models being used for NOO code.

It is worth noting that, if one orients toward the TOO strategy, one optimizes the potential for leveraging modeling tools, object generation technology such as MDA, and the facility by which OO ABL developers can leverage established OO design patterns in their work.

Associated Issues

There are other issues which are related to the choice of OERA strategy which also deserve some discussion in this context. One of these is the role of the ProDataSet in relation to OO code. While in some ways the PDS is an example of a higher level construct, such as one expects in a 4GL, it is also possible to see it as a sort of proto-object, i.e., something one did while waiting to get around to "real" objects. From that perspective, one might question whether, now that we have "real" objects, the PDS had outlived its usefulness since it has limitations and rigidity which true objects don't have. This idea may be reinforced by the difficulties which a number of people seem to have had in getting PDSs to "behave" properly, possibly because their use is not well understood, but also possibly because they have a rigid behavior which may not fit all circumstances. Similarly, while it seems attractive to have such PDS functionality as the `WRITE-XML()` and `READ-XML()` methods available as built-ins, experience seems to suggest that the built-in functionality is sometimes insufficient or inappropriate in some way and one is forced to write one's own methods anyway[2].

Still, there are some very nice features built in to PDSs such as change tracking, which would be desirable features for their use in objects or desirable features for objects to possess in their own right. I would suggest that this relationship be tracked as TOO OERA models develop in order to see whether objects might acquire features now associated with PDS or if PDS and objects can be provided with desirable linkages. It may be that appropriate "wrapping" of PDSs in objects can result in a combination which provides the best of both. It should be our expectation that there are valuable synergies available here, but we should not be mindlessly committed to using PDS if they turn out not to be a good fit. Note that the very thing which makes PDSs attractive to those advocating the ROH strategy, i.e., their preservation of the relational access which is ABL's historical strength, is potentially suspicious outside of the data access layer since objects are not inherently relational.

---

[2] One example of this which came up recently on PSDN was someone who wanted the WRITE-XML() method to nest a sub-table under a particular field in the master table in order to maintain compatibility with the XML being currently generated, possibly since the detail was related to that field and that would be the natural XML expression. The only options for WRITE-XML() are sequential tables or tables nested in a master-detail structure, reflecting the underlying relational orientation. The solution was to forgo the built-in method for a hand-written one.

One area which illustrates this issue is the contrast between a set object and an individual object. In the NOO and ROH strategies, there is a tendency to use PDSs to hold data, even when there is only a single "record" expected. One gains a certain flexibility this way, since a single interface can support either single records or sets, but there is a great deal of overhead associated with using a PDS to hold a single instance. Indeed, there is also a great deal of additional code required to use this approach for an object which is intended as a single instance since one needs both the properties and the link between the properties and the underlying PDS or TT, whereas the properties alone are sufficient otherwise. There is a great deal of simplicity which comes from simply using properties, although one may be losing the change tracking inherent in a PDS.

It should be noted that it would be a very attractive addition to the ABL if it were possible to define properties which were mapped to a buffer and this buffer could have some of the capabilities of PDSs and TTs such as change tracking. A very simple direct syntax for this structure would keep single instance objects very simple and yet provide advanced functionality with minimum coding. This is an example of a way in which thinking about connecting PDS functionality with object functionality can lead to enhanced capability.

A related issue is whether a set or collection object should consist of a temp-table of individual objects or whether it should be simply a PDS with an object wrapper. As a general purpose tool, the former has appeal in the same way that generic collections are useful in Java, but there are certainly cases where there is also appeal for having a PDS containing the equivalent of the relational data behind a whole set of entities and using traditional ABL language to iterate on and process that set. For certain types of data and certain operations, the latter is likely to be compelling, e.g., the ability to provide many indices and routes of access on a set of orders. There is no reason, however, that a set object of this type can't return a single entity object in response to a method so that externally all logic can continue to operate in the same way as if the set had contained actual objects. At the same time, the generic collection object also almost certainly has a role.

Another associated issue is the difference between the terminology now in use for NOO and ROH strategies in which a Business Entity is a business logic layer component containing the logic related to the entity and which accepts a PDS from the data access layer versus the use in the TOO strategy, where one has true entity objects which are marshaled in the data access layer and passed to the business logic layer for use. Some care needs to be taken in documentation to insure that the former use is not confused with the latter as the later is, I believe, the expected meaning in any OO context.

AutoEdge 2.0 and Framework

Having only recently released AutoEdge and knowing that it was a project which took more time and resources to complete than were expected, it is difficult to be thinking in terms of the possible need for a second version. However, I think there are some excellent reasons why this should be considered.

The most apparent reason, of course, is that if one determines that TOO is going to be the primary recommended strategy going forward, then it would clearly be desirable to have an OERA application example which illustrated the use of this strategy. And, it should be recognized that "doing it all over again in TOO" isn't really doing it all over again since the application has already been written and not only can a great deal of the existing code be used, but even when it needs to be significantly repackaged the existing code documents the needed behavior in great detail so there

is no need to go through the "discovery" process which certainly must have accompanied the original design.

However, one of the compelling reasons to consider this revision is that AutoEdge has been and is likely to be used as a model for production systems, when it is not really appropriate to be used as such a model. While PSC disclaims that it was intended as such a model, it is the closest thing available to such a model and so people hungry for models are going to look at it that way. In a recent webinar, one of the participants talked about his reason for selecting PSC to provide the mentoring for his company's transformation efforts by saying "they wrote this stuff". I.e., there is a tendency for people to look at PSC whitepapers, code samples, and code examples as authoritative communications from the source. If the person looking at this material is sophisticated about the issues involved, they are in a position to evaluate the materials fairly in the way they were intended, i.e., as pedagogical or as starting points for one's own considerations. But, if the consumer is not particularly sophisticated about these issues, no amount of disclaimer is going to keep them from thinking they must be authoritative.

Not only can this situation lead to disappointment and frustration as people discover empirically that these models are not ready for production use, but there is a huge opportunity cost because the response is likely to be minimum patching necessary to get the code working, not a discovery process that leads to a high quality solution. Had the materials they looked at originally exemplified best practice for production systems, then these users would not only have had a much happier development experience, but they are likely to end up with better software based on these better models.

In fact, I think there is a superb opportunity here to not just create a better example, but also to simultaneously extract from the example individual components which are appropriate ingredients in a production quality framework, components which could be used directly, not just as models. Naturally, no one form of component is perfect for every use, but by separating these components as individually published elements with a defined interface, then it also becomes possible to publish alternative versions of the components for different purposes. For example, one might have an authentication and access control component whose original version was based on tables in a Progress database and then create an alternate version which was based on an LDAP source. Both have the same signature and are plug-compatible according to the need of the site. Even before alternate versions are actually created, accompanying whitepapers can easily point to how such alternate versions might be created and fit into the same interface. I would suggest that it would be well worth considering making this an open source project so that contributions to improved and alternate forms could also come from the Progress community.

While arriving at AutoEdge 2.0 in full form and a collection of candidate production framework components would be a project that would take some time to complete, benefits can be derived even quite early in the project. My recommendation would be to start the project with a combination of open solicitation and direct interviews with leading APs to define the range of requirements of production systems, i.e., what is it that people really need or what would they like to have if they had it all to do over again. This can be documented as a set of goal specifications, probably an evolving document or set of documents, and would be highly useful in and of itself for companies to use in identifying their own needs. This could be followed by white papers proposing plans for each of the framework components identified from these needs. The whitepapers again provide a useful focus for discussion and decision by those involved in designing their own

solutions. This would be followed by actual initial versions of the framework components which are useful both as actual components and for illustrating programming principles. Using the open source approach, these would serve as the floor for an on-going effort of creating different "flavors" to suit different needs.

An actual AutoEdge 2.0 implementation could follow this in parallel since once one had identified a framework component, one would isolate and remove that aspect of AutoEdge and replace it with the component, thus illustrating its use. Simultaneously, one could evolve aspects of the AutoEdge specification itself to make the schema and structure better reflect the complexities of a production system. I am not suggesting that one would recreate an application as complex as a real world production system, but that there are a number of unnecessary simplifying assumptions in AutoEdge which clearly don't correspond to the way that production systems work, but which require only a small amount of additional complexity to make them much more production like. A good example of this is the inventory system, which at a minimum should have multiple "warehouses" to identify multiple storage locations and the list of available demo vehicles should be a separate table with properties appropriate to a demo which do not apply to a car not used as a demo. Anyone with a background in real ERP systems could easily add some of this kind of more robust structure without making the actual application unduly complicated to create or understand.

Prior to beginning this development one should also gather feedback on other lessons which might have been learned in the process of writing AutoEdge which were not incorporated into the released version. One example of this is that the naming convention and directory structure are not suitable for a large application and this is something which could easily be done such as to provide a better model. An OO version should almost certainly use the com.progress.autoedge type package naming which is common in most OO environments.

While this might seem like a large undertaking, I believe that it is one whose benefits are substantial and the work required modest in proportion to those benefits. As proposed, this project would produce:

1. Whitepapers documenting the needs of a number of common production-quality framework components;
2. Whitepapers documenting available strategies for fulfilling these needs;
3. Sample framework components which can be used as is or adapted to local issues;
4. A reference application illustrating good OO design practice;
5. A reference application illustrating the use of the framework components; and
6. A reference application which actually illustrates the use of ABL in a production-quality model.

This seems to me to provide enormous benefit both for assisting those developing new ABL applications and for those contemplating application transformation following OERA principles. It also could be used extensively in education and consulting efforts.