The OMG Business Object Facility and the OMG Business Object

Oliver Sims

SSA Object Technology

In January 1996, the OMG issued a Request for Proposal (RFP) for "Common Business Objects" and for a "Business Object Facility". This is one of the first OMG RFPs to address explicitly the application developer and end user. The Business Object Facility (BOF) takes a "top-down" view, looking at the needs of the user and application developer, rather than the previously normal "bottom-up" approach, aimed at the infrastructure builders' needs. The BOF has two major objectives:

- Enable **interoperability** of independently-developed business objects as "plug-and-play" components of the information system
- Provide **simplicity** in the development, deployment, maintenance and use of business objects for application developers and users.

An additional objective of the RFP is that there should be a direct correspondence, in understandable business terms, between the business model and the run-time business objects[1] which are components of the information system.

This paper discusses some important implications of these RFP objectives on the general "shape" of the run-time business objects, and on the Business Object Facility which enables and supports them. Before discussing implications, however, we need to expand on two concepts which are fundamental to the RFP objectives. These are "Interoperability" and "Simplicity".

Interoperability

The RFP talks of both "plug-and-play" and of "interoperability". These two are not synonyms. Let's deal with "plug-and-play" first.

Plug-and-Play

"Plug-and-play" means the ability to introduce a business object into a run-time environment so that an end user (or systems manager) can make use of it for some business purpose without any intervention being required by IT professionals. This process is sometimes known as "composition", or, of course,

[1]The RFP defines the term "application component" to mean an object implementation that is the run-time manifestation of a business object. The term "business object" is used in the RFP as a synonym for "application component", and to refer to a design/modelling construct. In this paper, the term "business object" is used to denote both the run-time information system software object, and the design/modelling-time construct. This will, I hope, avoid confusion between an application component which really means a business object, and an application component as something a developer uses to build something else, but which is in no sense a plug-and-play thing of direct use to end users.

"installation".

Installation ("plugging-in") of the business object must assume no compilation or linkage step, merely a very simple installation process, ideally requiring only base operating system facilities, into a running environment.

While plug-and-play implies that the object plugged in will be able to "play" (i.e. work), it does not necessarily imply interoperability - that is, does not imply that it will be able to interact effectively with other plug-and-play business objects.

Interoperability

"Interoperability" means that a newly plugged-in business object can be used (perhaps by an end user) in conjunction with, and interacting with, other business objects, such that their interaction performs some function useful for the business.

Since the developers of those business objects can never know with what other objects their products may be used, then such interaction must be able to be ad-hoc - that is, unplanned and unforeseen by developers. This is sometimes called "ad-hoc integration".

Interoperability introduces and important philosophical point. In general, when components (in our case, business objects) are to be integrated, there are two general approaches - or philosophies - relating to the process of integration:

- Optimistic - it is assumed that the business objects will interact as required, and if they don't, then it can be easily fixed
- Pessimistic - it is assumed that the business objects will not interact successfully, and therefore a "fix" process must firstly be executed - even when it is not required.

I much prefer the optimistic approach. While the pessimistic approach is more likely to identify potential problems up-front, it is also likely to impose increasingly lengthy procedures before anyone can plug and play and interoperate. In real-world terms, this would be like forcing someone to buy a micrometer in order to measure the precise diameter of a spark plug before being allowed to fit it. Sounds a bit dictatorial, doesn't it? I am strongly of the opinion that should the OMG adopt a pessimistic approach, then plug-and-play, not to mention interoperability, will be lost in a general move towards demanding sophisticated integration tools that only developers can hope to use. Such an approach would be much easier for the implementors of a Business Object Facility; however, it would kill the very objectives of the RFP.

Having said that, it is well worth noting that anyone who wishes to do so can fairly easily impose a pessimistic approach onto the optimistic one. The reverse is not true. If the Business Object Facility were to provide for plug-and-play *only* through some "fit-and-pre-test" tools, then one could only move towards some optimism by introducing an *additional* layer of tools, which "drove" the underlying tools based on some non-standard set of assumptions.

This paper assumes an optimistic philosophy for both plug-and-play and interoperation.

Simplicity

By "Simplicity", the RFP means hiding software technology complexities so that the application developer, whose skills lie in business solutions (as opposed to the software technologist who is skilled in system programming), can viably *implement* business objects. In turn, the business objects themselves should be plug-and-play interoperable components which an end user can use directly as parts of a business solution.

What *is* an OMG business object?

We know that a business object is something that exists; it is a cohesive lump of software in the run-time environment. Can we derive a more specific description than this?

An executable ...

The plug-and-play requirement means that the business object must be delivered as a separate executable - developed independently of other similar executables.

By "executable" I mean compiled and linked code - a binary - which, having been delivered by some developer and deployed into the Business Object Facility run-time environment, needs no other preparation before being executed or run. Examples of an executable are a Windows DLL or EXE file, or a Unix shared library member (again, a file).

Why do I rule out a specific language run-time environment, or perhaps a specific 4GL? Well, recollect that OMG standards and specifications deliberately do not pre-define any specific language. Although the RFP does not state this explicitly, it is nevertheless clear that business objects should be able to be built with any of the common languages in use today (or, hopefully, tomorrow!). That is, a developer of a business object should have a choice of languages in which to develop the business object. Hence the Business Object Facility should support language-neutrality; and hence it cannot itself be bounded by a specific language environment.

If the "executable" is written in an interpreted language, then an aspect of the language binding would be to provide a run-time layer which would map the Business Object Facility to the interpreter, and to the interpreted code. A 4GL could generate code, or intermediate interpreted code, to the required shape.

... of a specific "shape"

In order to plug the business object - the executable - into the Business Object Facility, then the business object must be of a known and standard software "shape". If it were not, then the receiving run-time system into which it is plugged - the Business Object Facility - could neither handle it, nor make it available to users and/or other business objects.

Implication 1: The Business Object is an independently-developed executable, and its software "shape" is defined by the Business Object Facility.

In my opinion, it is likely that this executable will be more easily handled as a dynamically-linkable entity, so that it can, if required, be run in the same address space (process) as other business objects. Assigning a separate address space for each object implementation seems to me to be sub-optimal, for a number of reasons (outside the scope of this paper).

Having looked at the implications of plug-and-play, let's now discuss interoperability - or ad-hoc interactions between separately-developed business objects.

Loose binding ...

Consider two separately-developed business objects. Interaction between the two must be able to be completely ad-hoc - that is, without that interaction necessarily having been planned for or even foreseen by the developers of the business objects. In other words, two business objects might meet for the very first time when a message is sent from one to the other. Before that event, such interaction may well never have been planned for - or even imagined - by any developer.

Clearly, for interaction to be effective, both business objects must embody some common concepts. That is, each developer of the interacting objects must have "agreed" to use the same concepts - but without their previously having agreed to agree! This must imply a common understanding of concepts, both those to do with the basic behaviour of objects (such as "Set" an attribute to some value), and those to do with the domain of interaction (such as "accounting" or "medical records"). Basic behaviour might be defined by the Business Object Facility. Let us assume that domain-specific behaviour and concepts are defined in some standard and publicly-available form. Within an organisation, one can see how this might be achieved; for inter-organisation commonality, we enter the realm of Common Business Objects (the other part of the RFP) - and of possible future domain-specific business object models.

Now if ad-hoc interaction depended on both developers using firstly the same IDL, and secondly the same version of that IDL, then the chances of successful interaction would be small. This is because IDL defines binding between objects in terms of computational detail such as data types, structures and sequences. For example, a user may know that two objects both embody the concept of "age". However, if the run-time interaction between objects is in terms of an integer by one object and a string by the other, then not only would there be a type mismatch, but there would also be confusion as to which of several integers (or strings) passed in a message is the one encoding a value for the concept "age".

What is required is a much looser kind of binding - one that can be done for the first time ever at message time, where the developers cannot be expected to hit on identical computational details.

Let's test this assertion. IDL can be said to provide a kind of distributed linkage editor, so that type-checking etc. can be done at build time - without actually having all components of the executable present to be bound, and without having entry points unambiguously identified. Now ask of any IS Manager whether he/she would like all the organisations' applications to be link-edited together, an implication being that a change in any interface may require re-linking a substantial number of parts. You would be thrown out of the office! As IS departments and business solutions move towards inter-application interaction, most implementors find that they need a much looser binding than can be achieved with IDL as it is currently used.

Implication 2:   Business object interoperability requires loose binding.

What we're saying here is that business objects should not be glued or welded together; they should be clipped or blue-tacked together. The lower-level technology objects used by middleware are the ones that need to be glued or welded together with much tighter binding. Indeed, CORBA and the current OMGfacilities and OMGservices are ideally-suited to such use.

It might be thought that an implication of this discussion is that IDL needs to be changed, or significantly enhanced. I do not believe this is necessarily the case. However, further examination of this topic is beyond the scope of this paper.

What we now need to touch on is this: how do we in general provide loose binding - or binding on concept only?

... with "Semantic" data

Loose binding is provided by minimising the "surface area" (Cox, p.16), or number of separate elements that the developer needs to consider at build time. What is required is to provide binding at the highest possible level - preferably at the level of the *semantics* of the interaction. It should not matter which computational details (type or sequence of data items) developers choose; the thing they must share is a common understanding of the concepts being communicated.

Common understanding alone, however, is not a sufficient condition for ad-hoc interoperability. The concepts must be encoded in some way so that at run-time each object can recognise the other's concepts. This seems to imply strongly that the semantics of interactions, and of data passed between business objects during those interactions, must be able to be encoded within message data. This can be done by having all message data in a self-describing form, where labels (metadata) for data values are passed together with the values. In fact, in a given message between business objects, the various items of self-defined data can be placed into an object - a "semantic data object". The business object message then contains, as its message data, an instance of the "semantic data" class.

Such an approach can also handle type mismatches, by performing automatic type conversions where required (this can give problems, but experience - with the "Newi" product from SSA Object technology - has shown that the benefits significantly outweigh the disadvantages).

Semantic interaction also has the great advantage that it can correspond directly to business object models. Thus some attribute label "CustomerName" in the model also flows in encoded form at run-time in object interaction.

Implication 3:   Message data should be encoded semantically.

Here is a simple example of the use of semantic data. Suppose I wanted to send a message to some object, and to send it a customer name, a customer number and a credit limit. Without semantic data, I might send the following structure (again, in pseudocode):

```
Structure:
   string  30
   string   7
   integer
End-Structure
```

This would be encoded in the message at run-time something like this:

```
"Smith & Co.                    AB12345000150000"
```

To make sense of this, the recipient would have to share with me the following pieces on knowledge, none of which appears in the message:

- the sequence of items (name, then customer number, then credit limit)
- the size of each item
- the type of each item in the structure
- the length of any variable-length items in the structure
- what each item actually *means* (its semantic value)

If I were to encode this same information as semantic data, then this is what might appear in the message I send:

```
"CustName=Smith & Co. | AccountNo=AB12345 | CreditLimit=1500.00"
```

To make sense of this, the recipient would have to share with me only one piece on knowledge which does not appear in the message:

- The meaning of the labels "CustName", "AccountNo" and "CreditLimit"

Types, data lengths and data structure can be hidden from the developer by the semantic data class. Methods provided by this class would enable the recipient to "pull out" any one data items independently of others. So to get the Customer Name, the recipient might code (in the invoked method of the receiving business object) something like this:

```
name = SemanticDataObjectReceived<--Get("CustName")
```

There is clearly more to semantic data than we can deal with in this paper. For example, homonym and synonym handling must be able to be applied without touching the delivered business objects themselves. Nevertheless, the above shows in outline how semantic data can deliver significantly loose binding to business objects by focusing surface area on the *semantics* of the interaction. Furthermore, and importantly, experience has shown that building and parsing semantic data can be made viable for the application developer.

We have concluded that the business object is an executable. We now go on to consider what it looks like to the developer.

What does a business object look like?

Hiding complexity

A major objective of the RFP is that a business object should be able to be built by an IS developer who is focusing on business logic rather than on software technicalities such as threads, memory management, multiple different system-level APIs, etc. (We might dare to go further, and to say that, with the appropriate tools, a business object should be able to be built by an ambitious end user.) Meeting this objective implies that the significant amounts of complexity inherent in writing business solutions directly on existing interfaces provided by the ORB, CORBAfacilities and CORBAservices must be hidden.

Why hide rather than simplify? The reason is this. Although it is generally possible to simplify the syntax of lower-level software complexities, it is seldom possible to simplify the semantics as well - to hide the range of knowledge needed to drive the simplified syntax. And it is in the semantics that the complexity typically lies.

Hiding complexity is done by imposing constraints on the programmer's freedom of choice.   An ideal approach is to hide complexities through constraints, but without preventing a software technologist to move, with clever programming, outside those constraints.

A good way of imposing constraints is to provide a specific "programmer's model" - a specific "shape" of code. This is not at all a new idea; it is what teleprocessing monitors do - they define the shape of a transaction program, and so hide significant computational software complexities. After all, the programmer cannot expect to build a monolithic application of any shape and have it be both simple to build *and* be plug-and-play and interoperable!

However, we have already said that the business object must be an executable of a specific "shape" in order to be plug-and-play. This might seem to be serendipitous. On the other hand, it might be evidence that we are on the right track - two different requirements lead to similar conclusions. So let's define a shape that can hide complexity.

Implication 4:   The business object must be of a specific technical software shape.

Also, remember that one of the requirements mentioned in the RFP is that the developer should create a unit of delivery which maps as closely as possible to the business object concept defined in the business object model.

The conclusion is that the unit of delivery should itself be an object - or more precisely, the implementation of a class.

Implication 5:   The business object should be the implementation of a class.

A further constraint is that the developer should be able to use a variety of languages, including perhaps procedural and scripting languages, to build the business object implementation. This means that such things as inheritance, handling multiple instances, providing space for instance data, etc., cannot be left to an OO language. That is, these things must be handled *outside* the unit of delivery.

Implication 6:   The business object must be language-neutral

Finally, by the nature of plug-and-play business objects, the developer *cannot* know about such things as when his/her code must be loaded, which thread it will run in (if at all), how many instances of his/her class are created - and hence how much memory is required for instance data, etc. For this reason, the loading and invocation of executables, memory management, thread management, instance management, communications management, blocking issues, etc. are all things which must or should also be placed *outside* the code implementing the business object executable.

That so much must be removed from the concern of the developer is a welcome conclusion. For such things *should* be of no concern to the developer, if we are to succeed in hiding complexity. In addition, by defining what factors should not be handled by the developer, we gain some purchase on the question of what the Business Object Facility itself must or should handle.

The Business Object programmer's model

We are now almost ready to suggest a possible programmer's model - the "shape" of the executable - of a business object. There is one other consideration which arises from the language-neutrality and loose binding implications. That is, method resolution cannot be done by executables having multiple entry points, since some languages do not support this. There are various ways to overcome this constraint, including a call-back registration approach (however this again limits the language choice). The simplest way of dealing with this constraint is, I believe, for each unit of delivery to have a single well-known entry point, and for the developer to handle method resolution *within* the business object implementation.

With this, and other factors discussed above in mind, then a "shape" of business object implementation might look something like this (shown in procedural pseudocode; OO pseudocode could also be used):

```
Start (business object invoked when a message arrives)

   Determine message

   StartCase handle message

      Case message = "Query"
         code for responding to a query
         invoke superclass
         return

      Case message = "Commit"
         Self "Query"
         Send result of query to database object
         return

      etc.

   End Case

   invoke superclass

End
```

Figure 1

The Business Object Facility could map all events of interest to business objects to incoming messages. Such uniformity assists with the simplicity objective.

This shape of code can be compiled and linked, and the resulting executable assigned a position in a class hierarchy. Interpreted languages can also be managed. When a message is sent to an instance of the class for which this code is the implementation, then the code is loaded, and the message passed to it. The message itself might itself be a small-grained object whose attributes might include:

·Message Name
·Id of target business object
·Id of invoking business object
·Semantic data object sent by invoking object
·Semantic data object for data to be returned from invoked object

Instance data for the specific instance could be handed to the developer automatically. For example, a buffer containing the instance data could be handed to the implementation at the same time it is invoked by a message.

Interaction between business objects

In the pseudocode above, we see the object sending messages both to itself, to its superclass and to another object. This is one area where, in real business systems, significant complexities lie in wait to trap the unwary. And in turn, this is an excellent area to exemplify hiding complexity. Indeed, a major advantage of the shape of code shown is that it provides a highly suitable vehicle for hiding complexity.

Consider, for example, an asynchronous messaging capability. This would be tailored to the application developer, in a top-down way. The first conclusion would be that the asynchronous message is asynchronous *with respect to the developer's code.* The major problem with this is how to provide the response to the developer, without him or her having to engage in call-backs, thread management, or other complexities. We may determine, for example, that the application programmer would like to be able to do the following:

- Write a single statement which causes a message to be sent asynchronously
- Define the message he/she wants to be returned with the response
- Have that response message delivered to him/her as all other messages are

From this, we might say that the programmer should be able to write something like the following, where in the method "Search" the message "DoSearch" is sent as an asynchronous message to business object X, and the result is requested as message "DoneIt":

```
Start (business object invoked when a message arrives)

   StartCase handle message

      ...

      Case message = "Search"
         ...
         SendAsync to X, "DoSearch", response message = "DoneIt"
         ...

      Case message = "DoneIt"
         handle result of "DoSearch"
         ...

      ...

   End Case

   invoke superclass

End
```

Figure 2

Experience with business objects to date has shown that while asynchronous messaging (from the point of view of the application programmer) should be used with caution to avoid getting into FSM (Finite State Machine) coding complexities, its use is sometimes necessary in real business systems. Normally, however, the programmer would prefer to use synchronous messaging, perhaps something like this (note that for clarity, handling of message data is not shown in Figs 2 or 3):

```
Start (business object invoked when a message arrives)

   StartCase hande message

      Case message = "Search"
         ...
         SendSync to X, "DoSearch"
         handle result of "DoSearch"
         ...

      etc.

   End Case

   invoke superclass

End
```

Figure 2

In both the above cases, there is no inherent computational complexity visible to the programmer. However, business object messaging of the kind described above does imply complex issues of blocking, re-entrancy and thread management (and of the underlying communications layer). These issues, and their resolution, can be hidden from the application developer through use of a specific programmer's model of the general shape described above. The complexities are exported to (handled by) the Business Object Facility, which defines and supports the programmer's model - the "shape" of the business object.

For example, in addition to the issues mentioned, the BOF would define "asynchronous" from the point of view of the developer; for example, "The developer will not receive a response message until he/she has completed the method in which an asynchronous message was issued".

Again, consider the recipient of a message; the developer should not have to be concerned as to whether his business object was invoked with an asynchronous or synchronous message. The programmer's model shown can enable the BOF to provide a model of invocation which could hide any differences from the developer. One of the things that makes this feasible is the use of semantic data objects for all message data. Thus the BOF, if it needs to handle message data on behalf of a business object, will always see just one type - a "semantic data object".

Mapping to the business model

As mentioned at the start of the paper, a major objective of the RFP is to provide a direct correspondence between a business object defined in an object model and a defined component in the information system. The programmer's model above, being itself a class which is enabled to live and breathe by the Business Object Facility, clearly provides a basis for this objective to be achieved.

Business object models will not infrequently define things other than class attributes and behaviour. They will also define such things as superclasses and relationships. Again, many facilities such as transaction support, business object name spaces, events, "views" of objects and roles will be required. It is worth noting that the Business Object Facility could provide frameworks to handle such things. Such frameworks could feasibly be built using the business object programmer's model, and using the Business Object Facility as their run-time environment. Of course, these frameworks would in many cases wrap specific uses of the CORBAfacilities and CORBAservices. This may be a useful way to hide the complexities inherent in some of those facilities and services.

Finally, the question of dependencies must be addressed. If plug-and-play components have too many dependencies on other pieces of software, and those dependencies have other dependencies, then the chance of achieving true plug-and-play is significantly reduced. The business object model described, by defining a clear first-cut design rule for the granularity and content of executables, can help significantly with handling this problem.

Summary of implications

Here is a summary of the various implications found in the course of the paper:

- The Business Object is an independently-developed executable, and its software "shape" is defined by the Business Object Facility.
- Business object interoperability requires loose binding.
- Message data should be encoded semantically.
- The business object must be of a specific technical software shape.
- The business object should be the implementation of a class.
- The business object must be language-neutral

Areas of standardisation

To enable the above implementation shape of business objects, the Business Object Facility would probably have to define standards in six general areas:

- Technical description of the executable
- Interfaces to supporting/enabling CORBA objects
- Message structure (CORBA says nothing about this)
- Precise way in which the BOF uses CORBA and CORBAservices and CORBAfacilities
- Management of class hierarchies
- Specific interfaces for tools, and support for ("hooks") underlying systems management and security facilities (these may be subsumed in the above five areas)

An immediate question is, can the above general approach be implemented without significant extensions to CORBA? I believe the answer is yes. In this paper, I cannot expand significantly on this answer. Suffice to say that I believe there is a very natural way to see a business object implementation as a CORBA object - and provide the characteristics described above.

References

Cox: "Object-Oriented Programming - an evolutionary approach", Brad J. Cox & Andrew J. Novobilski, Addison-Wesley 1991

OMG Document CF/96-01-04, "Common Facilities RFP-4, Common Business Objects and Business

Object Facility"