



Limits of Object-oriented Programming

| | |
|-----------------------|---|
| Published date | March 2006 |
| Version | 1.0 |
| Author | Pymma (Paul Perez) French to English Translation (Patrick J Stockreisser) |

Pymma group 22 Chestnut Avenue, Rickmansworth Herts WD3 4HB United Kingdom

Contact UK +44 79 44 36 04 65

Contact FR +33 6 99 36 07 10

Contact NL + 31 621 273 973

Email contact@pymma.com

Web site www.pymma.com

Table of content

| | |
|---|-----------|
| Introduction | 3 |
| Object-oriented Programming | 4 |
| Encapsulation..... | 4 |
| Object Domain | 7 |
| Limits of the Object Model | 8 |
| Limit created by memory | 8 |
| Limit created by location..... | 9 |
| Limit related to the organisation of the objects.. .. | 10 |
| Consequences of these limits..... | 10 |
| Inter-domain exchange..... | 10 |
| Other Problems | 11 |
| Need for a Neutral Format and the emergence of XML..... | 12 |
| XML Characteristics..... | 12 |
| End of Encapsulation | 12 |
| Conclusion | 13 |
| About the author | 14 |
| Appendix | 15 |
| Everyday Life..... | 15 |
| Synchronous and asynchronous communication..... | 15 |
| Encapsulation..... | 15 |
| Behavior in an XML document..... | 15 |

Introduction

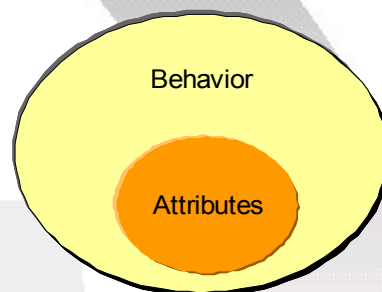
In the 60's, data-processing programs were very different to those of recent times. Not because they were either better or worse, or had greater or fewer bugs than those of today, but the way in which we thought about programmes was different. In 1969, NASA managed to send 2 astronauts to the moon with a processing power so weak you would not even wish to offer it to your nephew for Christmas. Desktop PC's we currently use everyday are much more powerful than is necessary to send men into space. Yet, sceptics asks why with all this processing power is it still not possible of having a 10 page Word document with images without a bug appearing. I would reply that that is another story.

Programmes were considered like a succession of instructions, functions, and procedures whose code described how specific data should be treated. In France, the methodology in the 70's, « merise » (<http://www.devaki.org/nextobjects/merise.html>), recommended the complete separation between data and the treatment of the data during the entire period of analysis. For the creators of this method, this meant that the structure of the data and its organisation had no influence on the operations (what we will now call the processing) and conversely the operations did not have any influence on the structure of the data. Still, as a student at that time known as "merisienne", I did not understand (and still don't understand) that such nonsense could be set up in certainty and be presented like methodology by almost all the French service companies of the time. With its passing, that must perplex us on the veracity of our current certainty.

At the same period, laboratories were working on some new programming techniques to simplify the writing of data processing programs. From this research emerged Object-oriented programming. In contradiction to the merisienne approach, Object-oriented (OO) programming proposed to place instructions and data in the same entity. Today, this entity is called an object. The goal of this restructuring is to enable easier modifications to the representation of the instructions and the data. To better utilities current terminology, I will use the term **attributes** and **behavior** in the place of instructions and data.

Object-oriented Programming

At the risk of being criticised by the purist of object-oriented (OO) programming, I propose that you imagine an object like a sort of shell in which there are 2 distinct elements: The attributes of the objects and its behaviors, a bit like an egg, where there is the yellow and the white.



An object is like an egg

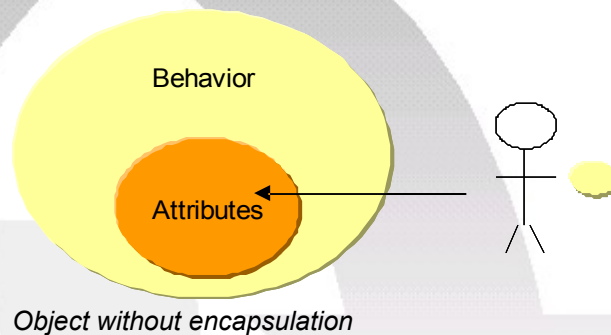
Of course, reality is much more complex than that. An egg is simple, everyone knows what it is and that helps much more with imagination than directly with Java code for the purpose of understanding.

Encapsulation

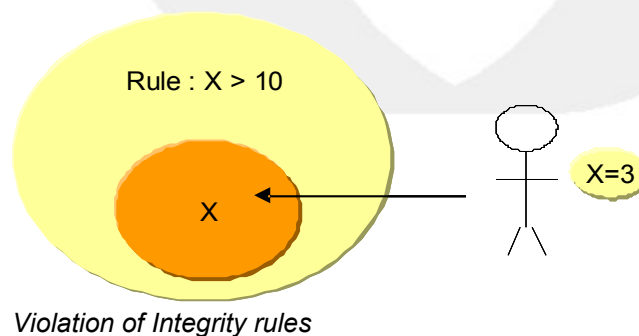
Object-oriented Programming puts forward 3 principle concepts: Inheritance, encapsulation and polymorphism. Here, we will not discuss inheritance or polymorphism. The explanation of these concepts is not the purpose of this paper, you may easily find all the necessary information for their comprehension on the internet. Here, I will only talk of encapsulation. In my opinion it is the most interesting characteristic proposed by OO programming, because it brings closer our way of programming to "everyday life". (See appendix)

The other properties of OO programming do not have this characteristic. Inheritance, for example, makes it possible for an object to behave like his/her father. In life it is rather the reverse which one observes.

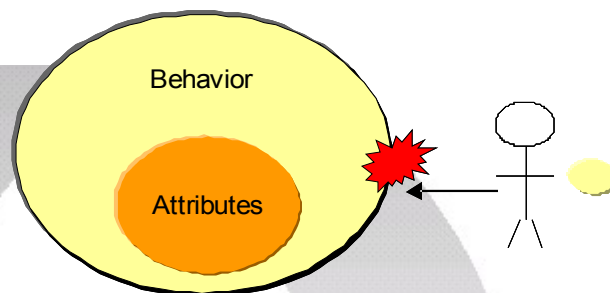
Let us now return to encapsulation. This property of OO programming is comparable with the shell of an egg. It maintains the yolk and the white in a protected space. Encapsulation has the same function, it protects the attributes and the behavior of the object and prevents any external entity from reaching the internal structure of the object.



In the example above an external entity accesses the attributes of an object in which encapsulation is not assured. It is possible for the entity to apply the behavior of its choice to the attributes of the object. It is a bit like having access to the memory of another person by hypnosis or subliminal techniques and modifying it without this operation being approved by the reasoning of that person. Inconsistencies appear very quickly at the studied subject, and in programming, it is the same thing.

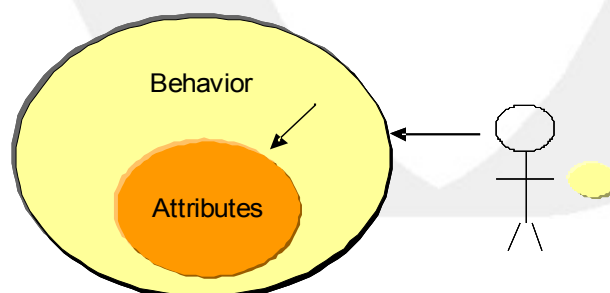


Let us suppose that in our object, there is an attribute X which for reasons specific to the business, must always be greater than 10. By having access to the internal structure of the object, an external entity can impose that X is equal to 3 and thus violate the business rules. It consequently creates a strong inconsistency in the object.



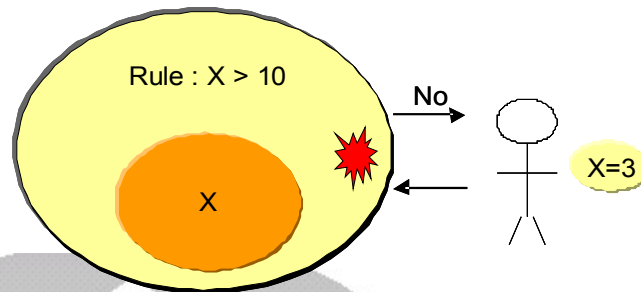
Object with encapsulation

Fortunately, encapsulation comes to our help and avoids this type of problem. The black border around the object represents the encapsulation. It acts as a barrier which prevents any direct access to the internal structure of the object. However, if the object represents a bank account, it is necessary for a banking application to update the balance of the account and to modify the attributes of the object. In this case, instead of directly accessing the attributes, an external entity will require the object to carry out the modification on itself. Hence, the rules of the behavior can not be bypassed any more.



Delegation of requests to the object

If we return to the example above, where X cannot be lower than 10, if the external entity requires that X be equal to 3, there will be a rejection of the request.



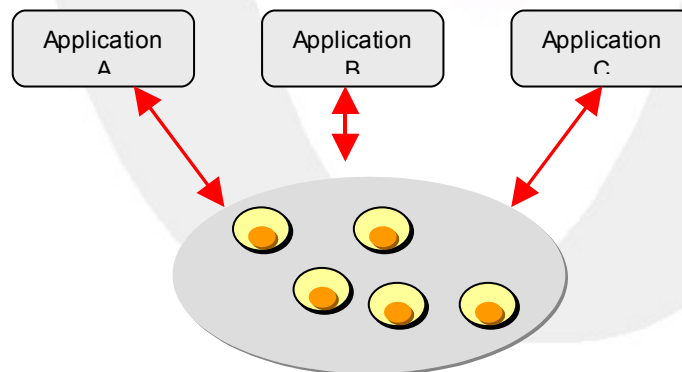
Rejection of incoherent requests

Encapsulation is a fundamental property to ensure the coherence of data in a system.

Object Domain

In object-oriented terminology, the whole of the data of a system is often called Object Domain (OD) or business domain

The object domain is the unit formed by the objects created and used by an application. Several applications can use the same domain simultaneously.

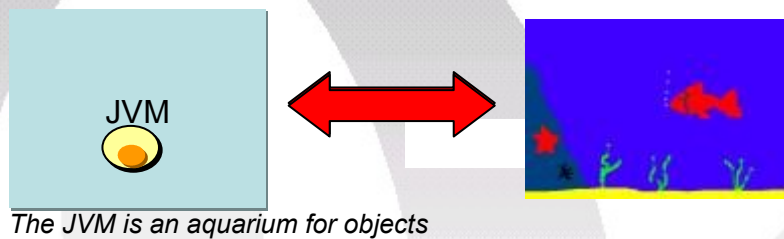


Multiple applications for an object domain

An object domain can contain thousands to hundreds of thousands of objects and a multitude of applications can contribute to the maintenance of that domain. There is no theoretical limitation with an object domain.

But obviously in practice it is not the case and limits exist. I describe three common limits often encountered, these can be related to limited memory, different locations between applications and OD or caused by their organisation being too different to be compatible. To illustrate these various limits, I will show examples in Java.

When a Java object is created, the natural environment in which it evolves/moves is the Java virtual machine (JVM). To make things simple, I will compare this situation with that of a fish in an aquarium. An object in a JVM is like a fish in water, as long as it is inside it is well, but as soon as it leaves the aquarium it dies.

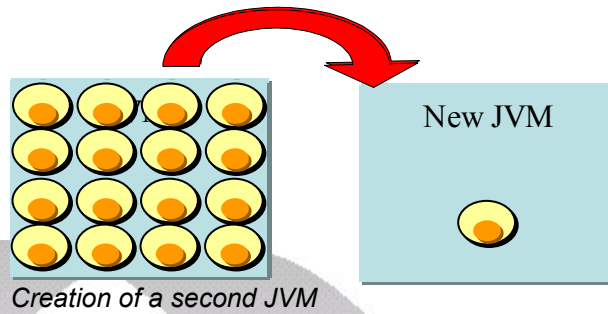


Is this a problem? No, not if anyone ever makes the fish leave water. But when an application meets one of the limits described above, we see that it becomes necessary to make an object leave its environment of origin.

Limits of the Object Model

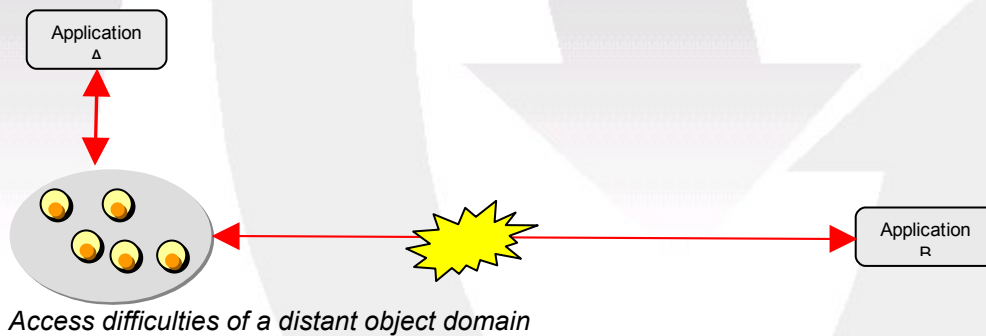
Limit created by memory.

When in a domain, the more objects created, the space available for more objects decreases. At some point when the maximum size of the JVM is reached it is not possible to create any new objects. It is necessary to create a second JVM (to use a second aquarium). In order to synchronise two ODs, it is necessary to transfer the objects of one OD to the other. (Have you already tried to transfer a fish from one aquarium to the other without a scoop? Welcome to the sport!)

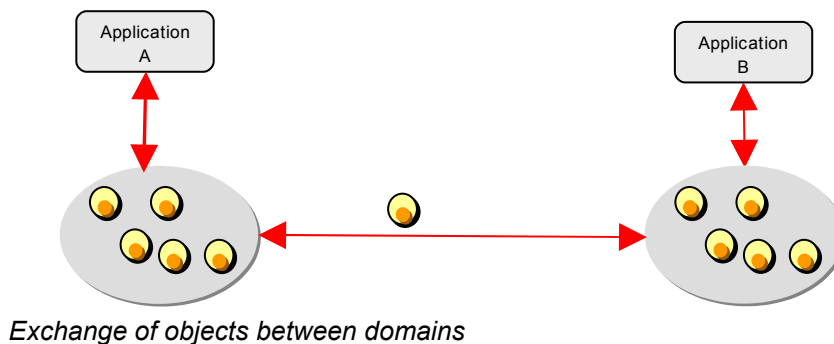


Limit created by location

Let us suppose that for technical or political reasons, two applications sharing the same OD must be physically distant from one and other. A big distance prevents effective access to the distant domain (limitations often due to the performances of the network)



It is practical to create a second OD to decrease the access times. The techniques of synchronisation between ODs are not detailed in this paper. It should just be noted here that a transfer of objects is necessary for this synchronisation.



Limit related to the organisation of the objects.

One may wish to enable two independent applications to work together without it being possible to amalgamate the two domains. Here is an example: Two banks A and B wish to amalgamate their applications. The first bank identifies its customers by a sequentially allotted number. The identification of the customer of the second bank reflects the organisation of the bank. It is composed of a code for the area, the region, the name of the branch and the number identifying the customers of that branch. In this case the incompatibility is too strong between the two methods of identification of the customer to enable a fusion to take place. Even if the two applications work together, the lack of unification of the data structures makes it necessary to maintain two different ODs .

Consequences of these limits

The consequence of these limits is the use of several operating domains. That amounts to having several aquariums in the same part.

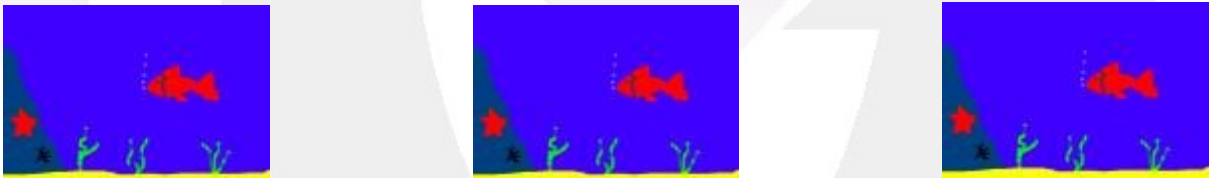


Figure 1: More domains = More aquariums

Famous dialogue :

| |
|---|
| <p>Donkey: <i>Hello princess</i> Princess Fiona: <i>Oh it's a donkey that speaks!</i> Shrek : <i>Ah yes, at thats the problem!</i></p> |
|---|

Having several operating domains is the beginning of the difficulties because it is necessary to be able to transfer an object from one domain towards another. In the rest of this paper we will see the consequences of this problem.

Inter-domain exchange

To explain clearly how inter-domain exchanges operate, I will still use the communication example between Java virtual machines (JVM).

The first rule which should be known is that, like a fish cannot live out of its aquarium, a Java object can exist only in the JVM where it was created. However, if several ODs take part in the same process, it is necessary that they can communicate, i.e. to exchange objects.

In designing the JVM, Sun proposed a simple technique for this type of exchange. This technique is called Serialisation. The serialisation makes it possible to create a kind of pipe between the JVM by which the objects can pass.

Serialising the objects limits the performance of the applications. To pass in this pipe, the objects must be flattened, then once arrived in the destined JVM, returned to their initial state. This process is called de-serialisation.



Communication between domains

Other Problems

The architects would be happy if the problems generated by the presence of multiple ODs related to only the performance. The situation becomes complicated simply when ODs do not use all same technology. If serialisation is used for the transfer of a Java object of one JVM to the other, no native settings to Java are maintained when the partner application is written in COBOL, C++ or C#. The reason for this problem is simple: The behavior of the object is written in Java code, it is not included/understood by the applications written in COBOL or C#. In the same way, the binary representation of the attributes (String, Int, length, date) is specific to Java technology, consequently, they cannot be transferred either. This is similar to transferring a fish to a bird cage or conversely a bird to an aquarium.

Here we stop our comparison with the fish and the aquariums, and the image of an egg will take all its significance now.

Need for a Neutral format and the emergence of XML

During the 90's, the need for interworking between various platforms became critical, the appearance of the Internet broke the walls of the last fortresses which did not want to open its doors to the outside (Mainframes of banks for example). It became necessary to enable communication between various applications and of course architects and developers were confronted with the problem of transferring objects between different technologies.

To solve this difficulty of communication and to avoid the proliferation of "point-to-point" protocols, the idea to use a Neutral format emerged. Neutral means that the message is independent of the technology which created it. The immediate corollary is that any technology can read and emit this type of message because there is no element of dependence on another technology. XML answered this requirement perfectly and has become the standard impossible to avoid for the communication between heterogeneous platforms. Today, it would be unthinkable to imagine these type exchanges without XML.

XML Characteristics

The way in which the specifications of the W3C (www.w3c.org) structure and organise XML documents, resembles greatly at the organisation of a Java object. One finds tree structures in which it is possible to navigate through, store and find information. But (and this "but" is all the objective of this paper) XML messages are able to transport only attributes. XML specifications describe in detail the organisation of the data... but will give you no indication on the behavior of the data contained in the XML messages. (see appendix)

Gradually XML documents have replaced the objects in the exchanges between different platforms of technology, consequently, the common rule is to send only the data of an object and not its behavior.

End of Encapsulation

Let us return to our egg example.

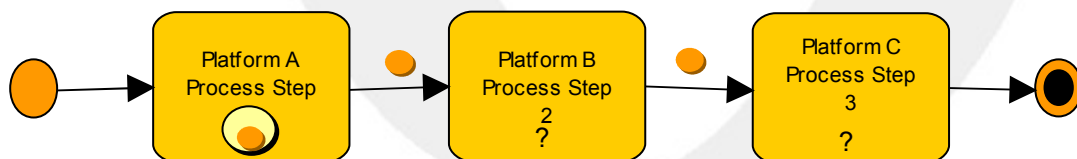
In a domain, we represented the objects like eggs where the white and yellow are protected by a shell. If we communicate with other systems and use XML (and for reasons of standardisation we use and we will use XML) we can transfer only the yellow (i.e. attributes), It is thus necessary to break the shell to separate the yellow from the white and to leave it without protection.

Thus, we return to the situation described to the beginning of this paper, before the emergence of object-oriented programming. The data is exposed to the external entities and can be modified directly without checking of the rules of the behavior. We highlight here a limit of OO programming. Its originators did not envisage the need to communicate between different platforms and alas, there does not yet exist a neutral standard of communication making it possible to completely describe the behavior of an object.

One of the principal goals of object programming was to create an entity where attributes and behavior were dependent. To reinforce this bond the concept of encapsulation was put forward. Object programming is still unable to formalize in a neutral way a behavior and thus to transmit from one domain to another makes it impossible to transfer a complete object between heterogeneous platforms and the creation of continuous behavior between different ODs.

Conclusion

The principal message that we can draw from this paper is that after years of object programming where attributes and behavior were bound and protected by encapsulation, we now reach a limit from which this type of programming is not possible any more. The need for interworking between applications and thus between operating domains prevents a standardisation of the behavior of an object throughout the execution of a process.



Process being carried out on several platforms

Which behavior will be applied to the attributes of the object in platforms 2 and 3? Is there a guarantee of coherence for each stage of a process? Well nobody knows. This uncertainty forces us to think about the interaction between applications differently, and to forget the idea of single ODs. This makes me think of an atomic physicist who would transfer the calculation for the trajectory of a satellite. Who would change it from quantum mechanics to Newtonian traditional mechanics. It changes a rule, and the way of thinking of dimension.

What are the new concepts to be applied, the new rules to be defined? We will try to approach these problems in the second part of this paper.

End of document

About the author

Paul Perez is co-founder and Chief Software Architect of Pymma. Paul is an author, software architect consultant and speaker, brings more than 17 years experience helping corporations utilize object technology for mission-critical information systems. Prior to co-found Pymma, Paul was an independent software consultant, working for large financial services companies in France, UK, Benelux, Israel and Germany. Paul's extensive experience in high-performance architecture design helps the company's clients solve real-world business problems through technology. As chief software architect, Paul assists current and future client engagements and develops best practices based on his domain expertise. Paul holds a post master degree from Paris-Orsay University and is a Sun Certified Enterprise Architect and holds several other certifications

Appendix

Everyday Life

This idea of “everyday life” is an empirical rule in the evolution of data-processing technologies which we noted and summarized as follows: We think at Pymma that any technology or methodology bringing closer the data-processing applications to our ways of thinking and acting, generally goes in the good direction. Here, two simple examples are detailed to help understand:

Synchronous and asynchronous communications

Contrary in the majority of the data-processing applications, the communications in “everyday life” are asynchronous. It is a simple observation: when you ask a question, you do not remain blocked as one large beta in waiting of an answer. And well this behavior of large beta is that of the majority of our data-processing applications which communicate in a synchronous way. When the asynchronous communications are used, the applications become much more powerful and flexible devices than their synchronous counterparts. The use of a property of “everyday life” improves quality of the applications

Encapsulation

In “everyday life” when you converse with another person, reactions are related to their social condition (who they are) as well as their character and their education. I find that there is an equivalence between the past and the situation of a person and what we call in object programming, attributes. One can raise some equivalence between the character of a person and the behavior of an object. Never, except in serious pathological cases, a person cannot disregard who they are or their education when they give an answer. Nobody can separate these two components from their personality which are their attributes and behavior. By creating the object, the laboratory researchers brought programming closer to “everyday life” by gathering in the same entity (the object) attributes and behavior.

Behavior in an XML document

Document Type Definitions (DTD) and XML schemas can make one think of behavior rules. There are two answers to such a remark:

- It is not possible to compare the richness of behavior suggested by a program in Java with the rules of validation of a schema. They are in two different dimensions.

- Neither the rules of validation, nor the validation itself are supported by XML documents themselves. For that, it is necessary to associate the document, an XML schema for the definition of the rules and a XML parser for the validation of the document. Thus, it seems difficult to me to think that XML messages will transport indications on the behavior of the object.

