A CASE STUDY OF DISTRIBUTED AND EVOLVING APPLICATIONS USING SEPARATION OF CONCERNS

Hamid Mcheick
Department of Computer Science and Mathematics, University of Québec at Chicoutimi, 555 Boul Université G7H 2B1 Chicoutimi, Canada

Hafedh Mili
Department of Computer Science, University of Québec at Montréal, Case postale 8888 succursale Centre-ville H3C 3P8 Montréal, Canada

Rakan Mcheik
Department of Computer Science, Institut des sciences appliquées, Beirut, Lebanon

Keywords: Distributed applications, separation of concerns.

Abstract: Researchers and practitioners have noted that the most difficult task is not development software in the first place but rather changing it afterwards because the software's requirements change, the software needs to execute more efficiently, etc. For instance, changing the architecture of an application from a stand-alone application, to a distributed one is still an issue. Generally speaking, we should encapsulate distribution logic in components through the borders of aspects oriented techniques (separation of concerns) in which we define an aspect as a software artefact that addresses a concern. Although, theses aspects can be offered by the same object that changes its behaviour during lifetime. We investigate through a case study the following ideas. Firstly, what we need like modifications to transform local application to distributed one, using a number of target platforms (RMI, EJBs, etc.)? Secondly, we analyze aspects oriented development techniques to detect what is the best technique that corresponds for changes requested to integrate a new requirements such as distribution.

1 INTRODUCTION

Separation of concerns are attractive because they exhibit long advocated software characteristics like modularity and cohesion, and their impact on software development has been described as another computing revolution on a par with those of stored programs and programming languages (Kiczales et al., 1997), (Mili et al., 2002), (Constantinides and Skotiniotis 2004). In the context of a distributed application, different sites, and different users may see different concerns/aspects of the same objects, including different functionalities, different access rights and privileges, different quality of service parameters, and so forth. Addressing these concerns means adding and changing code that crosscuts normal modularization boundaries, i.e. typically objects and methods.

The distribution has been considered by many researchers and practices as a technical aspect that can be handled independently of functional aspects (e.g. (Soueid et al., 2005) and (Mcheick et al., 2007). Consequently, we should encapsulate distribution logic in components through the borders of aspects oriented techniques. Firstly, the difference caused by the distribution has to be showed in OO program, i.e. what we need like modifications to transform local application to distributed one? Secondly, we should analyse aspects oriented techniques to detect which the best technique corresponds for changes requested to integrate the distribution.

Transform a stand-alone application to distributed one is, in the most cases, a solved problem. Indeed, existent distributed platforms use a variation of patterns (for example, stubs proxy) and various compilers provide automatically the majority of code used in distributed objects (for example, IDL compiler of Corba). Previously, researchers have suggested the use of middleware approach to distribute an application. Although, this middleware (like pattern proxy) resolves the most of circumstances, it stills some changes that need to be integrated: i) Object lifecycle: remote object creation is still different from local object creation. The researchers propose to use object factory, which is accessible using naming service, ii) The objects, that become distributed, should implement the interface needed by clients. Thus, the objects that should be translated between clients and servers should implement serializable interface, for example in the case of RMI, and iii) the invocation to remote methods can throw exceptions, which are linked to remote objects of clients.

These changes imply modifications in clients programs and in implementations classes programs at server side. We will investigate theses modifications using aspect oriented software development techniques. Each technique was developed with a set of problems in mind. Subject-oriented programming (Harrison and Ossher, 1993), and its incarnations (Tarr and Ossher, 2000), are purported to support feature-oriented programming and integration, favouring the separation of functional concerns. Aspect-oriented programming (AOP, (Kiczales et al., 1997) was built with architectural, non-functional concerns in mind. Our own view-oriented programming (VOP, (Mili et al., 2002)) was developed to handle functional concerns. Beyond the original intents of their designers, what kinds of problems are these methods best suited for?

The case study consists of taking a sample application, and submitting it to one maintenance scenario, consisting of adding a new architectural requirement, namely, distributing (or “remoting”) some objects. Through this experiment, we would like to, 1) gain some insights into the classes of problems that each method is best suited for, and 2) to explore whether distribution is a separable concern that can be added to an existing application after it has been built, using one of the three aspect-oriented development techniques.

The next section includes a brief overview of the aforementioned techniques and describes a distributed view-based model. Sections 3 and 4 describe the case study aimed at comparing three aspect methods where a distributed software requirement is investigated.

2 BACKGROUNG

We describe the various methods of separation of concerns and then distribution issues with aspects.

2.1 Aspect-oriented Techniques

Subject-oriented Programming. It views object oriented applications as the composition of several application slices representing separate functional domains or add-ons (features) to existing functional domains. Such a slice is called a subject and consists of a self-contained, declaration-wise, object-oriented program, with its own class hierarchy.

Subject-oriented programming enables us to compose these two hierarchies (subjects) into one that, generally speaking, consists of, i) the union of the interfaces (signatures) emanating from the two subjects, and i) the composition of the implementations of the methods that are defined in more than one subject (Ossher et al., 1995). A major limitation of SOP is the compile-time binding of the various subjects. Also, because the granularity of composition is the method, composability requires some pre-planning (Mili et al., 1996).

Aspect-oriented programming. It recognizes that the programming languages that we use do not support all of the abstraction boundaries in our domain models and design processes. Aspect-oriented programming requires three ingredients: i) general purpose programming language for defining the core functionalities of software components, ii) An aspect language for writing aspects, i.e. code modules that address a specific concern and that cross-cut various components in the general-purpose language, and iii) An aspect weaver, which is a pre-processor that “weaves” or “injects” aspects into the base software components to yield vanilla flavour components, coded in the general purpose programming language. The output of the aspect weaver is next fed into regular programming toolkit (compiler, linker, etc.) to yield the application.

View-oriented programming. We view each object of an application as a set of core functionalities that are available, directly or indirectly, to all the users of the object, and a set of interfaces that are specific to particular uses, and which may be added or removed during run-time. The interfaces may correspond to different types of users with similar functional interests or to different users with different functional interests. We set out to provide support for the following: i) enable client programs to access several functional areas or views simultaneously, ii) support the addition and removal of views (functional slices) during run-time, making objects support different interfaces during run-time, and iii) have a consistent and unencumbered protocol to address objects that support views. Accordingly, our implementation is based on: i) providing an API for manipulating views during run-time (adding and removing, activating and deactivation), ii) transforming code that uses views by replacing simple (core object) references by references to the wrapper, when needed.

2.2 Distribution Objects with Aspect

The combination of aspects and distribution is interesting for three reasons: i) distribution (garbage collector) is, itself, one of those design aspects that crosscut implementation classes, and that clutter the code without bringing in any new user-defined functionality. It would thus seem to be a perfect fit for a technique such as aspect-oriented programming, which appears to be particularly well suited for separating design-level concerns, ii) depending on the separation of concerns technique, objects that embody several concern may be fragmented, which may raise a number of issues for distribution, and iii) considering that different functional areas usually imply different data ownership and use privileges, to what extent can aspect, role, or view boundaries can be used as units for distribution—and possibly for duplication—in a distributed application context (see more details in Mili et al., 2006).

View-oriented programming deals with the distribution of object views mainly for two reasons: i) different clients need to share various combinations of different core object views, ii) object views can change behaviours during lifetime (see for example, (Mcheick, 2006)). Figure 1 depicts the case where an object implementation consisting of a core object and its views residing on the same server and where several clients have access to different sub-sets of the core object views. For instance, clients can have access to a combination of views (V1, V2 and V3). Client1 has access to views V1 and V2 when client2 has access to V2 and V3. It is important to note that the functionalities of V1 and V2 are not always available to client1 and similarly for V2 and V3 with respect to client2. In fact, the availability of the functionalities of theses views to their respective clients depends on the attachment operations invoked on the core object from the respective client. In this respect, the following requirements have to be satisfied: the core object must provide the implementation of the different view combinations required by clients.

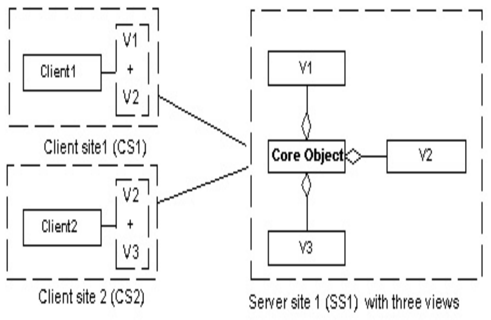

Figure 1: Architecture of distributed object with views: One server and many clients.

Two issues need to be addressed: i) how to make the same server object implement two or more client interfaces, and ii) where to handle the dispatch of multiply implemented methods (methods implemented by several views or by the core class and one or more views). Distributed platforms such as (CORBA, RMI, and EJB) typically ensure location transparency by providing proxy objects for the current objects residing on remote servers. These platforms use stub and skeleton classes, which are automatically generated to support the communication and the transparency. In this respect, a server object can offer different interfaces to several clients. We use the delegation approach which defines a tie subclass of the skeleton class. These classes (stub, skeleton, etc.) are modified to support view programming and include object lifecycle that is not standardized by CORBA (OMG, 2005). To offer multiply-implemented methods, we have the option of simply forwarding method calls to the server, and let the server side dispatch method calls or raise exceptions if a method is not currently supported. Alternatively, we could handle the dispatch on the client side, and then ensure that any call that goes to the server will get answered (see (Mcheick, 2006) or (Mili et al., 2002)). In terms of code transformations, this means: i) client side view management, ii) client side dispatching, and iii) a call is made locally or remotely.

3 A CASE STUDY OF DISTRIBUTED AND EVOLVING APPLICATION WITH ASPECT TECHNIQUES

3.1 Issues

The three methods described in sections 2.1 propose different modularisation boundaries for aspects:

- With subject-oriented programming, a subject is a class hierarchy which is definitionally self-sufficient, i.e. all the methods and attributes that are referenced are either locally defined or declared.
- With view oriented programming, a view is an object fragment, which is also definitionally self-sufficient, but that needs the core object to execute.
- With AspectJ™, the original "aspect-oriented programming" language (for example, AspectJ), an aspect is an amorphous construct that can take many forms, but essentially, an aspect is meant to implement cross-cutting concerns, i.e. concerns that pertain to several objects in a collaboration.

Generally speaking, the aspect-oriented software development (AOSD) community associates subject-oriented programming—and its descendant, HyperJ™—with functional concerns, whereas AspectJ™ is associated with non-functional ones, including architectural aspects, error handling, security, etc. That may have been the original intent of its developers, but as these methods are being appropriated by their users, new unanticipated uses are being found regularly.

This raises a number of related questions. First, can we use HyperJ™ to implement non-functional concerns, and conversely, can we use AspectJ™ to implement functional concerns? Second, assuming that HyperJ™ is more appropriate for functional concerns and AspectJ™ is more appropriate for non-functional concerns, what is it about these concerns that would make them require different modularisation boundaries. Tarr et al. hinted at the answer by noting that developers usually use a dominant decomposition to handle a first set of concerns, and then try to « slap » the other concerns on top of that (Tarr, Ossher, 2000). If that is the case, then the dominant decomposition is likely to be functional, which could explain why non-functional concerns tend to be cross-cutting. A more fundamental question is, are all concerns separable (see (Mili et al., 2006))? Aspect-oriented software development methods have been able to separate and package concerns that were thought non-separable with the object paradigm, but are all aspects separable, theoretically, and it is just a matter of finding the right packaging, or are some aspects inherently non-separable? In (Mili et al., 2006), Mili et al. have attempted to lay the foundations for answering such a question.

For our purposes, a subsidiary question is whether distribution is separable concern? Distribution is an architectural concern, not a functional one, and one is led to wonder whether it can be isolated into an "aspect", be it an AspectJ™ aspect, or a HyperJ™ subject, or a JavaViews view? The answer to this question is not only theoretical: if we are able to separate distribution, as a concern, into an aspect, that means that we can take a simple single-process or single-virtual machine application, and make it distributed by composing it with distribution aspects (or subjects or views). This has important practical implications, both from the point of view of application development, as well as from the point of view of re-engineering or scaling up existing applications.

Accordingly, we devised a case study that would enable us to gain some understanding into these issues. This case study consists of a simple computerized sales application that manages customers, orders, products, and inventory. To compare the three methods described in section 2, we consider an evolution scenario for the application, which is consisting of adding a non-functional requirement.

3.2 A simple Application

Our application supports a range of behaviours, including: 1) managing customers: this includes CRUD (Create, Read, Update, Delete) operations on customers, as well as account management 2) managing orders: this includes CRUD operations on orders, as well as order follow-up (e.g., figuring which fraction of an order has been processing), and invoicing, 3) managing deliveries, and 4) managing inventory: CRUD operations on products, as well as ordering more products from suppliers.

As mentioned above, we considered two evolution scenarios, one adding a functional requirement, and the other dealing with an architectural requirement.

As for the non-functional requirement, we initially thought of reproducing a variant of the architectural requirement implemented in the case of the ATLAS case study reported by Kersten & Murphy (Kersten & Murphy, 1999). However, those requirements did not apply to our case. Accordingly, we looked at the problem of turning a simple application into a distributed one.

4 ASPECTIZING DISTRIBUTION

4.1 Issues

As we mentioned before, turning a regular application into a distributed one is, for the most part, a solved problem. Existing distribution frameworks all use a variant of the proxy pattern, and various compilers will automatically generate most of the code involved in "distributing" objects. However, this code is typically generated before the application domain code is written. The sequence generates then edits works for forward engineering development but not for re-engineering or distributing existing applications.

With existing applications, the classes that represent objects that are to be distributed need to undergo some changes, and we will explore what those changes are. More problematic changes need to occur within programs that use those classes. Those concerns are i) Creation of remote objects (lifecycle) is different from that of local objects, and ii) Handling remote exceptions: remote method invocations may raise a number of exceptions that may either be related directly to the remoteness of objects, or that may be remote re-castings of user-defined exceptions. These can occur anywhere within a method in the client program.

Both subject-oriented programming and view-oriented programming allow composition only at the method level. Only aspect-oriented programming supports composition at sub-method levels, with some restrictions (entry and return points, exceptions, etc.). Thus, aspect-oriented programming seems to be, a-priori, the best fit for handling these kinds of aspects, on demand.

In the next section, we discuss the required changes that we need to make to an existing program to distribute some of its objects. We will discuss these changes in the context of specific technologies: Java RMI, and the EJB architecture. The CORBA distribution framework shares many characteristics with Java RMI and EJB, and will not be discussed.

4.2 Required Changes

If we want to distribute objects, we need to make changes to both the classes that implement the objects, and the code that uses them. To get a grasp on the kinds of changes that need to happen to client code, we show excerpts of a program that creates an object, and invokes methods on it (figure 2):

```
public class Main {
public static void main(String[] args) {
Company retailer = new Company("Home Depot");
…
Order newOrder = new Order(aCustomer);
// fill up the order
…
retailer.processOrder(newOrder);
…}}
```

Figure 2: A program that creates an object and invokes processOrder() method.

We considered the changes that needed to be made for three distribution frameworks: Java RMI, CORBA, and EJB. The EJB framework is the most complete—also, the most complex—and the most widely used. Thus, we will look at the changes required to deploy an existing class as an EJB.

4.2.1 Domain Classes

For illustration purposes, we look at what needs to be done for an entity bean, and we don't distinguish between local and remote interfaces. Thus, given a java class that we want to distribute, we need to make the following changes:

- Extract an interface (Remote) containing all of the application domain methods of the class, and make sure that all of the methods raise either RemoteException or EJBException. Also, make sure that all of the arguments are either serializable, or are themselves references to other distributed objects
- Create a class that represents a unique identifier for objects of this EJB: this is the primary key class. It can be any java class, as long as it is serializable.
- Extract an interface (Home) containing one create method for each public constructor the class has. This interface also needs to support a findByPrimaryKey(…) method that takes an argument that is an instance of the primary key class just mentioned.

Modify the existing java class to: i) make the class implements EntityBean, ii) add the appropriate exceptions to method signatures, iii) add lifecycle management callback methods (ejbPassivate(), ejbActivate(), ejbPostCreate(), etc.), iv) Implement the methods of the home interface (modulo some renaming), and v) in case of bean managed persistence, implement the load and save methods.

4.2.2 The Code that Uses Domain Classes Remotely

In the context of distributed objects, a client program cannot create an instance of the remote object using the traditional "new": it needs an object "creator" that it can ask to create a remote object on its behalf. That object creator is often also a distributed object, and we either need a way to create it remotely, or to locate it on the remote server. With EJBs, it is the "Home" object that has a global identifier (JNDI name). For CompanyHome, that name is ejb/HomeImprovementRetailers (figure 3):

```
public class Main {
public static void main(String[]
args) {
// Company retailer = new
Company("Home Deport");
Context initial = new
InitialContext();
Object ref =
initial.lookup("ejb/HomeImproveme
ntRetailers");
CompanyHome companyHome =
(CompanyHome)PortableRemoteObject
.narrow(ref,<home interface
class>);
Company retailer =
companyHome.findByPrimaryKey("Hom
eDepot");
      …
//Order newOrder=new
Order(aCustomer)
Order newOrder =
retailer.createOrder(aCustomer);
// fill up the order
…
retailer.processOrder(newOrder);
 … }}
```

Figure 3: An object creator that creates a remote object using EJB Home.

4.3 Implementing the Changes

4.3.1 Changes to the Domain Classes

The changes that need to be made to the domain classes are of two kinds:

Creating new interfaces (and a class) based on the existing one. This operation can be done by parsing the domain classes and getting the important information out, or by using the Java reflection package to extract the desired information. We are not really extending the behaviour of the base classes here, and we should not interpret the aspect-oriented programming techniques as code manipulation tools: they are first and foremost techniques for changing the behaviour of programs regardless of how that modification takes place.

Modifying the existing class. As we saw in section 4.2.1, those changes consisted of changing some of the type information of the class, and adding methods. The added methods are of two kinds: i) lifecycle callback methods which are added as-is to all domain classes, and ii) class specific methods, which implement the methods of the home interface.

Adding the type EntityBean to domain classes (with the statement "implements EntityBean") can be done in both AspectJ™ and HyperJ™. In HyperJ™ all we need to do is to combine the existing subject with another one that defines the domain class as implementing the interface EntityBean. If we merge the two subjects by name, we get the desired behaviour. However, HyperJ™ creates a new subject to combine both subjects. This implies that we should modify the code clients that use the existing subject if this code needs to use the new composition subject. With AspectJ™, we can get the desired behaviour by using the so-called inter-type declarations. This is not possible in JavaViews. With JavaViews, we can add the desired behaviour (EntityBean) as a view, if we wish, but it does not change the static type of the domain class.

With regard to the addition of the lifecycle management callback methods, all three methods support it. The most elegant solution is, in our opinion, the HyperJ™ solution because it involves merging the existing domain class with a class definition that includes the lifecycle management methods. AspectJ™, again using inter-type declarations, can do the same thing.

The following shows an aspect that uses inter-type declarations to add the "implements EntityBean" directive, and (some of) the new lifecycle management methods (figure 4).

```
public aspect
ejbCompanyEntityBean {
    declare parents : Company
implements EntityBean;
    …
    public void
Company.ejbActivate()throws
RemoteException,EJBException{
      }
    public void
Company.ejbPassivate()throws
RemoteException,EJBException{
      }
    public void
Company.ejbPostCreate()throws
RemoteException,EJBException{
      }
…}
```

Figure 4: An aspect shows how we can add a directive and methods.

The issue of adding exceptions to method signatures is a tricky one. First of all, note that none of the aspect-oriented techniques enables us to modify the signature of existing methods to add exceptions. However, all methods can help us extend the behaviour of a method with a given signature, and that can include throwing an exception.

To understand how we deal with these exceptions, we need to understand the rationale behind EJBException and RemoteException. First, the RemoteException signals to the client that an exception occurred on the server side, but the client does not necessarily has access to the details of the exception that occurred on the server side: the server side exception class may not even be in the client's class path. An EJBException is an exception that occurs within the EJB container. Because it is an unchecked exception, developers need not include it in the methods Home and Remote interfaces: if one such method raises the exception, the container will catch it, wrap it inside a RemoteException and sends it over to the client. The container will not wrap a business/application exception: it will be raised on the client side as is. This means that if containers weren't sticklers about types ☺, we could emulate the effect of having the exceptions in the signatures by actually throwing them in wrappers put around the bean class methods using what AspectJ™ and JavaViews calls before and after advices/methods, and what HyperJ™ calls brackets.

4.3.2 Title Changes to the Code that Uses Domain Classes

The client code shown in previous section shows the kind of changes that we need to make to the classes that use the application classes that I wish to distribute. Method invocation on remote objects works exactly the same way as with local objects, and that is the beauty of the proxy model. However, getting a handle on the first remote object, either through creation or through look-up, is different.

Code excerpts such as the following lines taken from previous section can occur anywhere within a client program. HyperJ™ and JavaViews perform behavioural composition at the method level. Therefore, there is no way that we can redefine object creation or access within client programs to use the remote model. AspectJ™ can extend behaviour at many join points: method call (from the outside), method invocation (inside, upon entry), method return (inside, before exiting), when we raise exceptions, when we reference an instance variable, etc. However, all of these joints have some meaning for the virtual machine, i.e. they correspond to specific operations of the virtual machine. We can not extend a program at any instruction.

```
1. …
2. Context initial = new
   InitialContext();
3. Object ref =
   initial.lookup("ejb/HomeImprov
   ementRetailers");
4. CompanyHome companyHome =
   (CompanyHome)
5. PortableRemoteObject.narrow(re
   f,<home interface class>);
6. Company retailer =
   companyHome.findByPrimaryKey("
   HomeDepot");
7. …
```

If developers use factory patterns to create and look for objects as a general practice, remoting objects becomes much simpler because we localize the changes to the methods of a single (or a handful of) factory(ies). But for general-purpose programming, we cannot "remote" the manipulation of objects systematically using any of the aspect-oriented techniques.

5 CONCLUSION

In wide-enterprise information systems, changing the architecture of the application from a stand-alone application, to a distributed application has been investigated in this paper. Generally speaking, we tried to encapsulate distribution logic in components through the borders of aspects oriented techniques in which we define an aspect as a software artefact that addresses a concern. A number of techniques collectively referred to as aspect-oriented development techniques, have been proposed that offer new artefacts (beyond method, class, or package) that can separate new kinds of concerns that tend to be amalgamated in object-oriented programs. As users adopted and appropriated these methods, new unanticipated uses appeared and raised the question: which method is best suited for which class of problems?

In this paper, we reported on a case study that submitted an application to evolution scenarios: 1) adding functional features to the base application, and 2) changing the architecture of the application from a stand-alone application, to a distributed application. In that scenario, we evaluated the impact of the change, and explored ways to implement it using each one of the techniques. This experiment has important practical applications for distribution: if we are able to encapsulate the act of remoting an object, into a separate software component (or aspect) that we can compose or weave into simple Java object, we would have solved an important (re)engineering problem. We argue that if there are many object types that need to be modified, AOP allows to gather the modification, in a modular way, into a unique entity (aspect). That is not always easy to do it with SOP that changes the name of existing class and the client code.

We studied the Enterprise Java Beans distribution pattern, because of its popularity and complexity and not surprisingly, we found that some changes cannot be modularized into a separate aspect. Is this a problem with the distribution solution (the EJB architecture) or with the distribution problem? This, again, is not an idle theoretical question. One of the premises of Model-Driven Engineering is that architectural design and the coding of business logic are fairly independent activities, enabling us to "code once" and "deploy everywhere". The transition from platform-independent model (PIM) to platform-specific model (PSM) applies an architectural mould (e.g. the EJB pattern) to a bunch of domain class. If we could write the business logic in a way that is entirely independent of the deployment infrastructure, we can write it once in the PIM, and deploy it to different platforms. In transformational systems jargon, this is equivalent to saying that architectural design and business logic elaboration (coding) are two commutative activities (Baxter, 1992). This experiment seems to suggest that they aren't where the biggest hurdle is lifecycle management. By abstracting object lifecycle management, we will probably succeed.

REFERENCES

Baxter, I., 1992. *Design Maintenance Systems*. CACM, vol .35 no. 4, pp. 73-89.

Büchi, M., Weck, W., 2000. Generic Wrappers. *In ECOOP'00*. LNCS 1850, pp. 201–225.

Constantinides, C., Skotiniotis, T., 2004. Providing Multidimensional Decomposition in Object-Oriented Analysis and Design. *Proceedings of the IASTED International Conference*. Innsbruck, Austria, Fub.17-19.

Harrison, W., Ossher, H., 1993. Subject-oriented programming: a critique of pure objects. *In Proc. of OOPSLA'93*. pp. 411-428.

Kersten, M., Murphy, G.C., 1999. Atlas: A Case Study in Building a Web-Based Learning Environment using Aspect-Oriented Programming. *OOSPLA'99*. Denver, CO, USA.

Kiczales, G., Lamping, J., Mendekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M., Irvin, J., 1997. Aspect-Oriented Programming. *Proceedings of the European Conference on Object-Oriented Programming (ECOOP07)*. Springer-Verlag, Finland, pp. 220-242.

Mcheick, H., Mili, H., Msheik, H., Sioud A., and Bouzouane, A., 2007. ASPECTGC: Aspect Garbage Collection for Object lifecycle management. *Proceedings of ACM (ICICIS'07)*. Cairo, Egypt.

Mcheick, H., 2006. Distribution d'objets en utilisant les techniques de développement orientées aspect : programmation orientée aspect, programmation orientée sujet et programmation orientée vue. Thèse de doctorat, 273 pages, *Université de Montréal*, Québec, Canada.

Mili, H., Harrison, W., Ossher, H., 1996. SubjectTalk : Implementing Subject-Oriented Programming in Smalltalk. *In proceedings of TOOLS USA 1996*. Santa Barbara, CA, July 29 - August 2nd, 1996, Prentice-Hall.

Mili, H., Mcheick, H., Dargham, J., 2002. CorbaViews: Distribting objects with several functional aspects. *Journal of Object Technology*. USA.

Mili, H., Sahraoui, H., Lounis, H., Mcheick, H., Elkharraz, A., 2006. Understanding separation of Concerns. *Fundamental Approsches to Software Engineering, FASE'06*. Vienna (Austria), March 27-29.

OMG:www.omg.org, 2005.

Ossher H., et al., *Subject-oriented composition rules*. In *Proc. OOPSLA '95*. Austin, TX, Oct. 15-19, pp. 235-250.

Soueid, T., Yahiaoui, N., Seinturier, L., Traverson, B., 2005. Techniques d'aspect pour la gestion de la mémoire répartie dans un environnement CORBA-C++. *In Proceeding of NOTERE'05*. Gatineau (Québec), Canada.

Tarr, P., Ossher, H., 2000. HyperJ User and Installation Manual. *IBM Corporation*. http://www.research.ibm.com/hyperspace, USA, 2000.