First Step of Security Model for Separation of Concerns

Conference Paper · January 2009

DOI: 10.1109/AICCSA.2009.5069400 · Source: DBLP

Hamid Mcheick
University of Québec in Chicoutimi
162 PUBLICATIONS **389** CITATIONS

Hafedh Mili
Université du Québec à Montréal
184 PUBLICATIONS **3,582** CITATIONS

First Step of Security Model for Separation of Concerns

Hamid Mcheick, PhD Computer Science, Eric Dallaire, Master Student
Department of Computer Science and Mathematic
University of Quebec at Chicoutimi, UQAC
555 Boulevard de l'Université Chicoutimi, G7H2B1, Canada
hamid_mcheick@uqac.ca, edallair@uqac.ca

Hafedh Mili, PhD Computer Science
Department of Computer Science
University of Quebec at Montréal, UQAM
Case Postale 8888, Station Centre-Ville, Montréal, Québec
H3C 3P8, Canada
hafedh.mili@uqam.ca

***Abstract*—The separation of concerns (SOC), as a conceptual tool, enables us to manage the complexity of the software systems that we develop. When the idea is taken further to software packaging, greater reuse and maintainability are achieved. One of the methods of SOC is view-oriented programming (VOP) in which an object can change its behaviors and play different roles (views) in their lifecycle. In VOP, an object's response to a message depends on the views currently attached to its core instance. This view-programming suffers from security issues to protect the privileges of each client who needs to access different views of the same object. In a previous article, we introduced a view security model based on changing the signature of each method to authenticate a client privileges to access object views. In this paper, we present the main parts of our views security model without changing each method signature. Java security model is applied to views to support transparent authentication. These issues are discussed through an example.**

I. Introduction

In wide-enterprise information system, objects change behaviors during their lifetime, regularly acquiring and relinquishing properties. View-oriented programming (VOP) [10] is one of the separations of concerns approaches for allowing objects to change behaviors during their lifetime, when that change takes place within the same program. Mcheick proposed a view-oriented model [5] to answer three challenges: i) dynamic behavior change, ii) supporting multiple interfaces, and iii) distribution of these views.

In the existing view programming approach [2, 5], clients require explicitly the activation, the deactivation, the attachment and the detachment of remote views, even if these views are distributed on different sites. In distributed view-oriented programming, different client sites may access different view combinations of the same object, known as interfaces. Therefore, the servers have to manage several interfaces requested by different client sites. Each view passes through several states: active, inactive, attach, detach–called view lifecycle [3]. For example, a client site 1 cannot detach a view that is still referred by client site 2. Consequently, there is a need to manage the view security in local and distributed environment especially: the management of shared views by the different clients, and the management of the concurrency of these clients with contradictory interests. In this context, a first issue has to be addressed when a distributed object offers different views to different clients. Suppose that client1 activates two views V1 and V2, and client2 actives two views V2 and V3. Let us also suppose, for the sake of explanation, that client1 calls a f() of V1, which calls g() of V3. A security access problem would be if client1 somehow tries to perform a g() operation of V3, which is not activated by client1. Each program client has to get only its privileges to access an object. Consequently, the access to these views has to be specified to preserve the privileges of each client. Client1 cannot access to some operations which are not belonging to his domain of privacy. Another issue has to be addressed: in the existing view programming model, each client has to decide which views it needs. We prefer to manage views in a transparent way for clients. This situation can happen because object base can't identify the caller who has to access some views. Existing view security model is decided explicitly by the clients.

Our idea is to let the wrapping application object handle transparently the security privileges and identify each client based on Java security model. The views security privilege would define what the user or client can do and which aspects the user has access to, as determined by the access level of the security privileges on the side of core object. Some researchers have proposed an approach which uses the authentication of each client based on Java security model [9]. Also, traditionally, the concurrency to access objects or different parts of these objects has been solved by synchronization [8]. But in our case, the problem is the concurrency and dynamic behavior of an object which can activate and deactivate a view dynamically. We introduced a first step of that model [6], which needs an implementation. Also, this last model authenticates a client based on changing the signature of each method call. In this paper, we propose a dynamic and security view model and its implementation based on Java security model.

The integration of security model in legacy applications requires the instrumentation of their code. This instrumentation can be handled easily by aspect-oriented approach which allows decomposition in terms of aspects. These aspects crosscut classes and modules [1, 5] and so far an object security management can be handled as a technical aspect.

The next section includes a brief overview of Java security model and separation of concerns approach. We present in section 3 the problems of access privileges in view-oriented programming. Section 4 describes a view security model to support implicitly the authentication of each client. The implementation of this model is illustrated in this section. We conclude in section 5.

II. Background

This section describes briefly separation of concerns such as view-oriented programming and Java security model.

A. *View-Oreinted Programming(VOP)*

As object technology matured, researchers and practitioners have noted that a number of software development concerns cannot be handled using the modularization boundaries inherent in object-oriented languages, namely, method → class → package. A number of techniques, referred to as aspect-oriented development techniques, have been proposed that offer new artifacts (beyond method, class, or package) that can separate new kinds of concerns that tend to be amalgamated in object-oriented programs. Those concerns are obtained using a general solving heuristic that consists of solving a problem by addressing its constraints, first separately, and then combining the partial solutions with the expectation, that, 1) they will be composable, and 2) the resulting solution is nearly optimal. For this heuristic to yield satisfactory results, the concerns that we are trying to treat separately must be fairly independent, to start with, so that they don't interfere with each other. Further, the problem solving activity itself needs to yield solutions that are composable. These issues are discussed in details by Mili et al. [4]. Briefly, each aspect-oriented technique was developed with a set of problems in mind. In this section, we describe briefly VOP.

The behavior of an object with views depends on the set of views that are currently added and active. When the object receives a message, its answer depends on whether its core functionality or one of the attached views supports the requested behavior. If no implementer is currently available for the requested behavior, the request is denied. To understand VOP, we explain it through an example.

```
import com.walmart.core.Customer;

import com.walmart.finance.*;

import com.walmart.operations.*;

(1) Customer myCustomer =
Customer.getInstanceWithID(id);

(2) myCustomer.attach("Loyalty");

(3) myCustomer.attach("CreditWorthiness");

(4) float val = myCustomer.getCreditLimit();

(5) myCustomer.printCustomer();
```

In line (4) the programmer invoked a behavior that is available in the CreditWorthinessCustomer view on the instance of Customer, without referring explicitly to the view instance. The underlying mechanism is a pre-processor that replaces line (4) with the following line,

```
(4')float
val=((CreditWorthinessCustomer)myCustomer.getV
iew("Credit"+"Worthiness")).getCreditLimit();
```

because it knows that `getCreditLimit` () is available in the view class CreditWorthinessCustomer, but it does not know for sure that at the time that the call is made, an `CreditWorthinessCustomer` view is attached and active, and we cannot sort this out at compilation time. Line (5) shows the method `printCustomer`() which is supported by both `LoyaltyCustomer` and `CreditWorthinessCustomer`. We adopted the approach advocated by Harrison & Ossher [7], which consists of composing the various method implementations.

B. *Java Security Model*

The Java security architecture is composed of many libraries. JAAS (*Java Authentication and Authorization Service*) is applied on views model because it's easy and light to implement. Also, JAAS can help us to solve a part of separation of concerns security problem: How to make client identifiable, how to identify this client and how to control it.

B.1 Authentication Process

Before defining permissions and privileges, the client must be recognized and authenticated without any ambiguity. With platform JAAS, client authentication is done in safety and with reliability independently of who's responsible for the execution of the code. The authentication process is executed through a sequence of operations which is summarized in the following class diagram (figure 1).

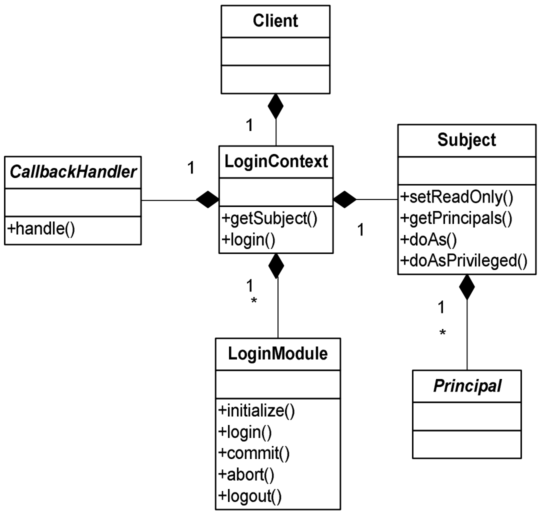


Figure 1. Class diagram of authentication process.

A client first loads a Login Context which corresponds to a personalized security zone. The context contains a method login used for validation process and entity creation. Each entity is represented by a Subject object.

This context next calls the login method of each loginModule. A module responsibility is to implement an authentication routine. The result of a successful login will add identities (Principal) to the entity (Subject). Those identities can be of any type, for example numerical signatures, identifiers or other information of identification such as `userId` (`uid`) or `groupId` (`guid`).

The next step is to call `setReadOnly`() method which effectively sets the object non modifiable. The client can get a reference of the « identifiable » entity via `getSubject`() method. The class « `Subject` » thus contains a list of « `Principals` » and it could validate the permissions. This subject model facilitates the authentication of client to access a given view.

B.2 Authorization Process

The process of authorization is carried out following the authentication. With the help of the identities included in the entity, it will be possible to apply restrictions on our identified client. The authorization with JAAS extends the current Java security architecture which uses security policies to specify access and privileges of the running code.

A system resource (a view by example) could be protected by permission. An important characteristic of the permissions is that they must be coupled to a secure object such as an identity (Principal). For example, a client authenticates itself and an identity is represented by a class VOPPrincipal which is added to its entity (the Subject class) under the name « admin ». A corresponding permission could be created and it could result in this rule:

IF the identity connected to the current execution in the current security context has a named identity « admin », instance of VOPPrincipal:
THEN grant the permission
ELSE refuse the access

Permissions are registered in a policy file.

B.3 Using JAAS

How the call of a method coming from an anonymous class may start the authentication and authorization processes? The key is the class Subject!

Subject has static methods `doAs`() and `doAsPrivileged`() receiving the entity instance (a subject) and a privileged action as parameters. The method `doAs`() associates the current subject with the current security context and calls the method `run`() of the privileged action.

The Java Security Manager runs the privileged action but instead of considering the calling instance privileges, it use those added in the Subject. This process is automatic and transparent and could be easily resumed: If some privileged instructions need particular permissions to be executed, the security manager checks Subject's identities and authorizes or interrupts the execution. Figure 2 illustrates these situations.

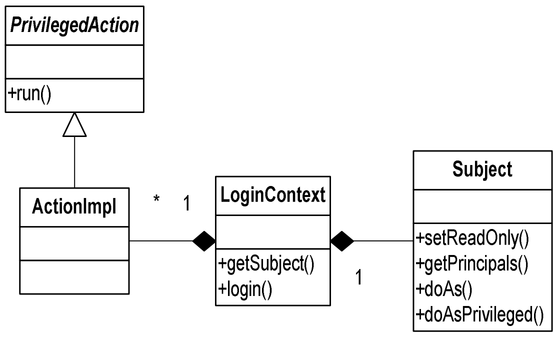


Figure 2. JAAS Model.

Finally, the authorization process can be summarized as follows:

- The client log itself and tries to execute an action under it's define identities.
- The security manager validates the action done under those particular identities.
- As soon as the privileged action tries to reach a protected resource, the manager checks the security policies.
- If the call remains in its login context (sand box), execution continues as normal.
- If the call leaves the sand box, the security manager detects an illegal access and throws an exception.

III. PROBLEMS

Before defining our security model, let us illustrate the problem by the following example. We have an application object named OBJ_APP. OBJ_APP contains a core basic object (OBJ_B) and three views (V_1, V_2 and V_3), having three functions respectively `f(x)`, `h(x)` and `g(x)` (see figure 3).

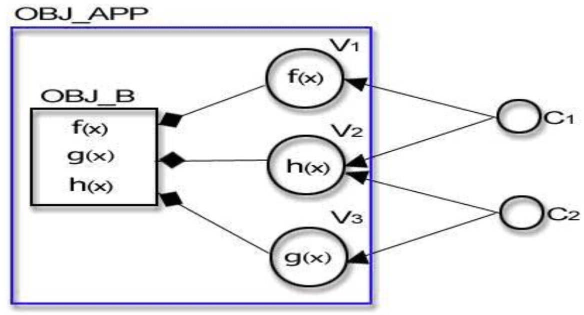


Figure 3. An application object OBJ_APP contains a core object OBJ_B and three views V_1, V_2 and V_3, C_1 and C_2 are two clients.

Suppose we have the following situation:

- The client C_1 attaches V_1 and V_2 to OBJ_APP
- The client C_2 attaches V_2 and V_3 to ABJ_APP
- The application object (OBJ_APP) allows an access to the base object (OBJ_B) using the three views V_1, V_2 and V_3.
- When a method is called on OBJ_APP, it will be redirected to each active view or core base object in which this method is implemented.
- Let us suppose that C_x represents a client x. Each client is a user program.

With traditional view oriented programming, our approach is to use OBJ_APP dynamically through the evolution of functional needs. Authentication procedures are not present and clients are anonymous. OBJ_APP doesn't have any mechanism to identify its client. If we couldn't get a secure communication, we could not be able to support security and sharing issues.

We propose a security model where each client uses an authentication supported by OBJ_APP (which, conceptually, represents the complete object). We'll use permission's validation process supported by the Java Security Manager to validate this authentication.

First of all, let us see some of the principal problematic appearing with the integration of our security model.

A. *Problem #1: Client role*

The client role was to attach and activate its required views. In our previous diagram, C_1 and C_2 shared view V_2. What will happen in the following situation: C1 calls h(x), C_2 detaches V_2, C_1 calls h(x) again. OBJ_APP will throw an exception. What of C1 permissions? In two consequent calls to h(x) by C_1, behaviors of OBJ_APP change. Client role isn't adapted to a multi-clients approach and must be revaluated.

B. *Problem #2: Who is the Client*

From a conceptual perspective, C1 and C2 are distinct clients but from OBJ_APP perspective they could be any class. So the challenge is to make the client unique without heavy modifications to each implementations of OBJ_APP (one per OBJ_B). The client must continue to call `f(x)` and at the same time tells OBJ_APP « I'm C1 » without changing the signature to `f(x, uid)` as we did in the last model [6].

C. *Problem #3: Remembering the Client*

This one is very tough to implement, our OBJ_APP must implement some sort of client profile in order to proceed with authentication. The identifiable client must be identified in order to continue. Otherwise, every client C_x would be allowed to call any view. This recognition could be done by a third party (a relation <C_x, <server, OBJ_APP>>) but to keep our code light and simple, we'll keep a two parties relationship <C_x, OBJ_APP> with no dedicated authentication server.

D. *Problem #4: Access issue*

Inside OBJ_APP, a permission mechanism must be implemented to protect views and methods. Suppose V_1.`f(x)` must call V_3.`g(x)` during its execution. To call `f(x)`, we need to be sure C1 has access to both V_1 and V_3. Hard coding security in the views is in many ways stupid so it's excluded. So, we need to provide a way to check if a particular identified client C_x has access to a particular view V_x without writing it directly in the classes.

E. *Problem #5: Some other problems*

Derived problems have been identified in this research. For example, Synchronization problems are part of any multi-users program. Because of the dynamic interactions between OBJ_B, OBJ_APP and the views, deciding where's to put validation is very touchy and error prone. Another problem is what we called « partial call ». Let us consider a call to `OBJ_APP.f(x)` which could execute three function calls: V_1.`f(x)`, V_2.`f(x)` and V_3.`f(x)`. If each view modifies OBJ_B, what would happen if the client has no access to V_3? Method calls to V_1 and V_2 could be executed and could throw exception to V_3. However, `OBJ_B` has been modified with no rollback possibility. Therefore, those problems remain tied to implementation and won't be discussed in the present solution.

UQAC and NSERC Canada

IV. Security Model for Separation of Concerns

The security model is composed of many implementation classes. We present parts of these classes. Our solution is based on the Java 5 Security Package, specifically on JAAS (Java Authentication and Authorization Service). Both our client and OBJ_APP solutions are based on it.

A. *The Client*

Our first attempt was to use a class encapsulating the need of the client and communicating with OBJ_APP but the Security Manager use the security context of the caller (the client), a anonymous class with no particular identity. Using RMI client-server procedures only move the problem forward. With `RMIRegistry`, skeleton is only another anonymous object, so there's no gain. After many unsuccessful attempts, we have been force to accept the fact that the only way to authenticate a client to OBJ_APP is by altering the call stack or to be precise, to change the security context of the caller at runtime. Our client make the call but under a pseudonym, the Subject.

The caller should not be an anonymous function but an identified client C_x. To succeed, we introduced a new client class `OBJ_APPSecure` which wrapped our OBJ_APP instance and the Subject receive from the client. For each OBJ_APP we generate an `OBJ_APPSecureImpl` that inherits from `OBJ_APPSecure`. Its role is to receive the function call from the client and sends the information to the execute method in `OBJ_APPSecure` and returns the result of the client.

```
protected Object execute(String methodName,
Class<?> params, Object[] values) throws
VOPException {

  Method method = null;

  try {

    if (params == null)

       method = objApp.getClass().

       getDeclaredMethod(methodName);

     else

       method =  objApp.getClass().

       getDeclaredMethod(methodName, params);

       return objApp.execute(subject, method,

                                values);

    } catch(Exception e){}

}
```

With the help of the Java class Method from the reflection package, we transform the method call to a simple object and transfer the information to OBJ_APP. C_x continues to use the same signature `f(x)`.

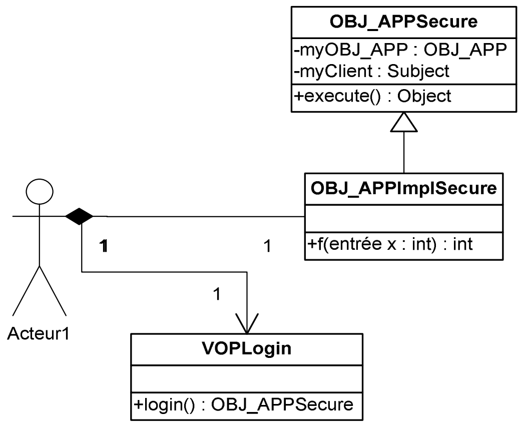


Figure 4. Authentication of client using transform method call.

`OBJ_APPImplSecure` comes from a factory named VOPLogin and its role is to identify the client, to create the secure object around OBJ_APP and the subject instance. The subject is the logical (and secure) representation of the login done by the client. This object (`Factory`) will create a client alias to help in security procedures. `VOPLogin` encapsulates a `LoginContext`, its related classes and our own implemented `LoginModule` (see section 2.2.1 for details). Therefore, authentication technology is completely hidden from the client. The following sequence diagram resumes the authentication process as seen from the client perspective.

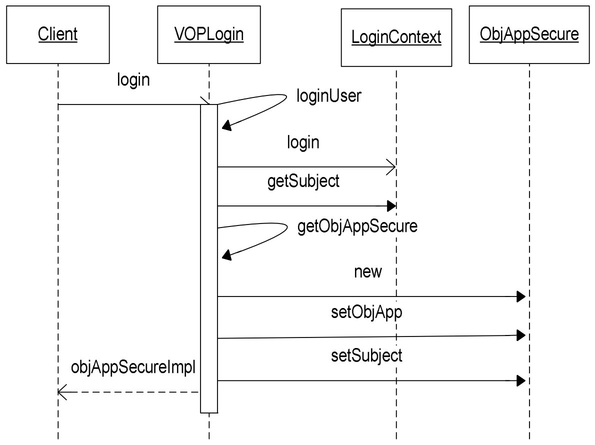


Figure 5. Sequence diagram to authenticate a client.

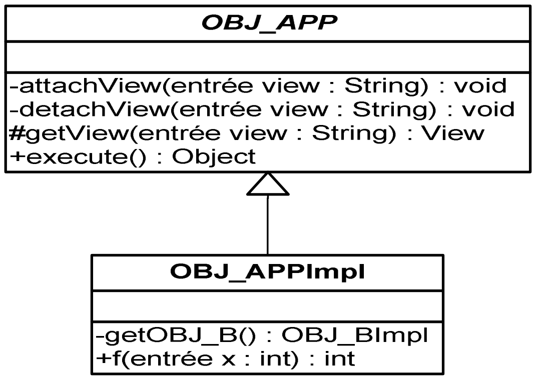
Here's an example of the client code for f(x)

```
OBJ_APPImplSecure myOBJ_APPImplSecure = null;
   try {
     myOBJ_APPImplSecure=
    (OBJ_APPImplSecure)VOPLogin.login
    ("ObjAppImpl");
    myOBJ_APPImplSecure.f(x);
  } catch(VOPException vope) {}
```

B. *OBJ_APP*

`OBJ_APP` doesn't know the calling class but it receives the Subject instance as parameter of the method execute.

```
public Object execute(Subject subject, Method
method, Object[] values) throws VOPException{
  Object objReturn = Subject.doAsPrivileged
       (subject,new VOPPrivilegedAction
       (method, values, this), null);
       if (objReturn instanceof VOPException)
         throw (VOPException) objReturn;
       return objReturn;
}
```



OBJ_APP
-attachView(entrée view : String) : void
-detachView(entrée view : String) : void
#getView(entrée view : String) : View
+execute() : Object
OBJ_APPImpl
-getOBJ_B() : OBJ_BImpl
+f(entrée x : int) : int

Figure 6. Object application and its subclass.

In the `execute()` method, we're preparing to call the method `f(x)` of the current object OBJ_`APPImpl` (this) but with the alias « subject ». The `doAsPrivileged()` method will try to execute a privileged action inside a sand box defined by Java Security Manager. The privileged action is very simple: trying to call « `method.invoke()` » which, in our case, is a call to `f(x)`.

```
public Object run(){
   try {
     return method.invoke(objApp, valeurs);
    } catch (Exception e) {
      String message = null;
      if((Exception)e.getCause() instanceof
                                  VOPException)
        message  = "Can't access method (" +
        ((Exception)e.getCause()).getMessage()
        + ")";
      else
        message  = e.getMessage();
        return new VOPException(message);
      }
    }
```

Inside the try catch block, the invoke method call `f(x)` of OBJ_`APPImpl` and it will try to call `f(x)` from every implemented view. Accessing a view is an operation controlled by the security manager. So, if the client has sufficient right, invoke method continue as normal. In the other case, an `AccessException` is thrown by the security manager, catch in the run method and wrapped by a `VOPException` that is returned to the client. These procedures, the bridge between our client and OBJ_APP, may seem complex but it's JAAS ways of doing and the implementation is hidden and shared by every possible implementation of `OBJ_APP`.

C. *Permissions*

We implemented Java Permissions in our OBJ_APP to validate, through the privileged action, view access. Each time a view is needed, a method `getView()` in `OBJ_APP` is called (see code extract)

```
View currentView = (View)
views.get(neededView);
ViewPermission perm =
currentView.getPermission();
try { AccessController.checkPermission(perm);
} catch(Exception e){
  String message  = "Access denied to view " +
  view + " (" + e.getMessage() + ")";
  throw new VOPException(message);
}
return currentView;
```

Each view is composed with a view permission class (View Permission extends `BasicPermission` from Java API). Using the Security Access Controller, it's possible to check if the current client (in realty the subject from the « execute » method) has the permission to use the current view. A last detail should be added to complete our security issue. Where do we store clients and views permission validation? This information is saved in a security policy configuration file named « `vop.policy` ».

```
grant codebase "file:vop.jar", Principal
vop.securite.VOPPrincipal "C1" {

  permission vop.securite.ViewPermission "V1";

  permission vop.securite.ViewPermission "V3";

};
```

In this case, the security Manager grants privileged of views V_1 and V_3 to the authenticated client C_1. Every other access will throw an exception

This dynamic adaptation model of the application object OBJ_APP raises a new layer of work effort in view-oriented programming. Nevertheless, it has some advantages: i) The conflict between clients is decreased; ii) Java security model is easy and efficient to facilitate view security authentication; iii) It could be implemented with minor change in the client and OBJ_APP; iv) Many of the new classes could be generated with minimal effort; and finally v) views remain free of security implementation.

V. CONCLUSION

Our work addresses the problem of access privileges in view-based system in which several functional domains or aspects are supported within the same application. In particular, this paper addresses dynamic model for view oriented programming.

We have proposed a security view model to i) offer different functional aspect/views to different client programs, ii) handle view functional calls in a system, and ii) preserve the privileges of each client where each object offers a set of dynamic views. By supporting the access control of views, the clients can call any function and get the right answer dynamically. Indeed, views are code fragments, which provide the implementation of different functionalities for the same object domain and these views can be used as a units for distribution to improve performance issues.

Another issue is discussed: if one client has activated the shard view and the second client has deactivated this view, than the first one will be no longer able to use this view. For this reason, a view security technique can avoid such concurrent access issues. The presented approach is at an early stage of development and would certainly merit to be more detailed especially for validation aspects and manage views in a transparent way for clients.

ACKNOWLEDGMENT

This work was sponsored by the University of Quebec at Chicoutimi (Quebec), and by NSERC (Canada).

REFERENCES

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

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

[3] H. Mcheick, H. Mili, H. Msheik, A. Sioud, and A. Bouzouane, "AspectGC: an Aspect Garbage Collection for Object Lifecycle Management", *Proceedings of Third International Conference on Intelligent Computing and Information Systems (ICICIS07), Sponsord by ACM SIGART and SIGMIS*, pp.150-157, 15-18 mars 2007, Ain Shams University, Cairo, Egypt.

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

[5] H. Mcheick, Distribution d'objets en utilisant les techniques de développement orientées aspect, *thèse de doctorat*, 273 pages, Université de Montréal, Québec, Canada.

[6] H. Mcheick and E. Dallaire, SecurityViews: Security Object Model for View-Oriented Programming, *Proceedings of Mctech08/IEEE*, Montréal, Canada, Jan. 2008.

[7] W. Harrison and H. Ossher, "Subject-oriented programming: a critique of pure objects," *in Proc. of OOPSLA'93*, pp. 411-428.

[8] C. Horstmann. Big Java, third edition, John Wiley and Sons, 2007.

[9] http://java.sun.com/javase/6/docs/technotes/guides/security/ (Guide de sécurité Java 1.6, 2006).

[10] H. Mili, A. Mili, J. Dargham, O. Cherkaoui, and R. Godin, 1999. View Programming: Towards a Framework for Decentralized Development and Execution of OO Programs. Proceedings of TOOLS USA '99. USA, Aug. 1-5, Prentice-Hall, pp. 211-221.