INGENIERÍA Y DESARROLLO

Applying aspect oriented technology to relational data bases: The replication case

Tecnología aplicada a aspectos en base de datos relacionales: El caso de replicación

Fernando Asteasuain*
Adeel Javed**

* Investigador - Docente - Becario Doctoral. Conicet de Universidad de Buenos Aires, FCE y N, departamento de computación. Licenciado en Ciencias de la Computación, Universidad de Bahía Blanca. Investigador - Docente - Becario Doctoral Conicet de UBA-FCEyN, Dpto de Computación. *fasteasuain@dc.uba.ar*

Correspondencia: Oficina 12, Departamento de Computación - Facultad de Ciencias Exactas y Naturales. Universidad de Buenos Aires, Pabellón I - Ciudad Universitaria (C1428EGA) - Buenos Aires (Argentina).

** M.Sc. Student, University of Central Punjab (Lahores, Pakistan), Senior Software Engineer - Technical Lead | BPM Practice, Techlogix Pakistan , MS Computer Sciences (Pvt) Ltd. *adeeljaved@techlogix.com*

Número 25
Enero-Junio, 2009
ISSN: 0122-3461

Abstract

Due to aspect-oriented mechanisms explosion, their concepts arrive to distributed systems tackling concepts as security, persistence, or synchronization, especially in middleware approaches. Regarding distributed databases, the research has been focused mainly on object-oriented databases. Based on the great impact of these works, we introduce an aspect-oriented framework for relational data bases, incorporating a fundamental concept as replication as an aspect, achieving a truly independent replication layer. A conceptual model for replication is defined, which guided the implementation of our framework called Sigma.

Key words: Aspects, data bases, replication.

Resumen

Debido a la explosión de los mecanismos orientados a aspectos, sus conceptos han llegado a los sistemas distribuidos, atacando conceptos como seguridad, persistencia, o sincronización, especialmente en frameworks orientados a middleware. En este contexto, la investigación en base de datos distribuidas está enfocada principalmente en base de datos orientadas a objetos. Basándonos en el alto impacto de estos trabajos, introducimos en este trabajo un framework orientado a aspectos para base de datos relacionales, incorporando un concepto fundamental como lo es el concepto de replicación, obteniendo una verdadera capa independiente de replicación. Se presenta un modelo conceptual, el cual guió la implementación del framework propuesto de Sigma.

Palabras claves: Aspectos, base de datos, replicación.

Fecha de recepción: 4 de diciembre de 2008
Fecha de aceptación: 14 de febrero de 2009

1. INTRODUCTION

In the beginning, Aspect-oriented technology was applied only at the implementation stage, but with the advent of new languages and more powerful modularity capabilities that correctly abstracts crosscutting concerns, this initial situation changed very quickly, and aspect-oriented concepts were translated to other development stages like requirement engineering and design [1], verification and formal approaches [2], [3] as well as new platforms and tools arises [4]. This aspect-oriented explosion also reaches middleware frameworks used for large distributed systems [5], [6], [7], where aspects are used to abstract inherent concerns such as persistence, transactional communication, security, quality of service, or synchronization. Another interesting application involving large distributed systems is database management, so it is natural to conclude that aspect-oriented technology could provide great help in their development too. In

this context, Rashid et al. investigated this issue ([8], [9], [10], [11], [12], [13]) and successfully applied aspect-oriented mechanisms in order to achieve separations of concerns in database systems while focusing on object-oriented databases and modeling persistence as an aspect. In this work, we build on top of these proposals introducing a Java framework called **Sigma** for *Relational DataBases* where a core requirement as **replication** is encapsulated within an aspect allowing database designers to build a database independent replication layer.

With the introduction of new technologies in the database community, distributed databases became a reality. In particular, database replication, which is defined as *the process of copying and maintenance of data on multiple servers*, gained transcendence. Every major database vendor now supplies a replication solution in one way or the other. Due to replication's crosscutting nature, implementing its functionality in an independent, customizable and separate fashion will certainly make database development and evolution much easier. To our best knowledge, replication, although mentioned as a candidate requirement to be implemented as an aspect, remains unexplored in a database context.

The rest of the paper is structured as follows: The next section introduces replication as a database concept, and analyzes two different models for handling replication. Section 2.2 presents a conceptual model, where replication is considered as a main architectural component. Section three presents our framework which implements the conceptual model, and the remaining sections conclude our work.

2. REPLICATION MODELS

Replication is a key process for achieving databases' successful behavior, since its functionality helps to guarantee data consistency, and allows the database engine to keep working in case of network failures. In case of distributed databases, data distributed among different nodes in the network must be correctly synchronized to ensure data consistency. This involves copying and maintaining every data manipulation from one location (the node where the data manipulation took place) to the other nodes in the network that are to be updated. This means that network configuration and node communication greatly impacts on replication performance, as expected.

Replication techniques can be modeled either on top of a database engine, on a separate layer, or can be provided internally, as a fixed mechanism. Although our work is focused on the first option, in this section we briefly discuss two internal models mainly for comparison reasons in order to achieve a more complete analysis.

2.1. **Internal models**

In the next subsections, two internal models are described, using object oriented patterns and the master worker architecture.

OO patterns

Probably the most widely adopted way of decoupling collaboration among objects is through the object oriented patterns philosophy [14]. Perhaps the pattern that best adapts to replication features is the *publish-subscribe* design pattern (figure 1), commonly used in object-oriented software systems. This pattern behaves as follows: various subscriber objects can register with a publisher object to receive asynchronous notification callbacks when information is published via the publisher object.

In a replication context, every data manipulation is *published*, and every node that was *subscribed* to that event, receives the replicated data. In this model, the database manager must define the data to be replicated, the node that will be in charge of publishing events, the nodes subscribed to each event, the distribution mechanism and how long after an event was published the subscribers receives the notification. This last item is relevant to performance issues. This publish-subscribe metaphor is used in Microsoft SQL server 6.5, among others. Although the configuration seems relatively easy, the database is not always robust enough to manage complex and frequent modifications [15]:

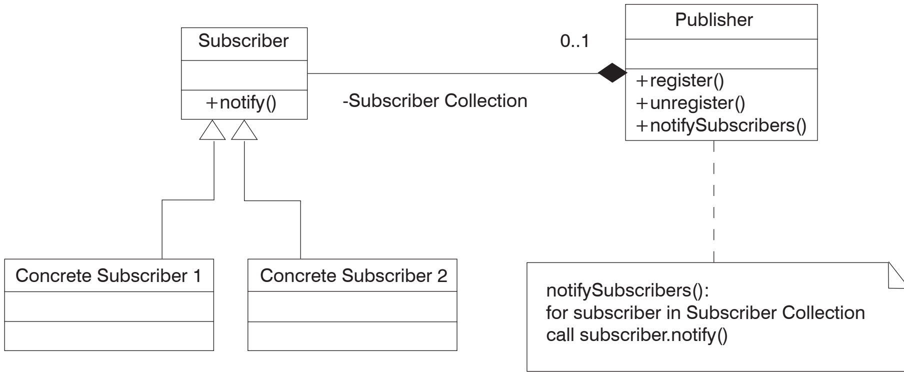

Figure 1: A UML scheme describing the Publish Subscribe Pattern

Master/Worker Paradigm

The master/worker paradigm (figure 2) is a typical architecture within distributed systems. Under this centralized scheme, a distinguished node is designated as a *master*, and the rest play *workers* role. The master makes all the decisions, and distributes information among the workers, who process it and eventually return the processed information to the master. Next, the master gathers all the information from workers and produces the final result. This situation can include several iterations until the final result is obtained.

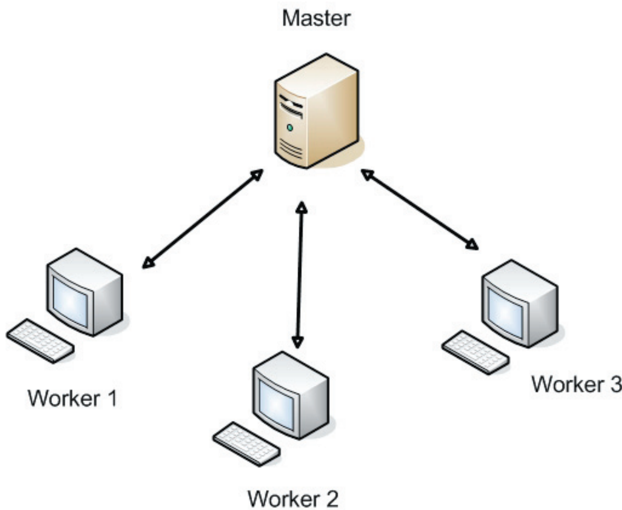

Figure 2: A Scheme for the Master Worker Architecture.

Applied to databases, a master node controls every replication decision, and distributes replicated data to the workers, so that every node manages the same data. This architecture follows a one-way, asynchronous replication, and currently is being used in MySQL.

The master/worker architecture is rather simple to configure, but it suffers the inherent disadvantages of centralized schemes. Every worker receives and sends data to the master, which causes a communication bottleneck.

Discussion

Two models for replication have been briefly presented. For a more complete and detailed comparison the reader is referred to [16].

In many cases these default replication techniques are more than enough for database systems, where the replication requirement is not so crucial, or the system dimensions fit under some replication default model, but these solutions are not an answer to all problems. Both models discussed earlier suffer from scalability problems, or communication bottlenecks, but the main disadvantage is that replication is not considered as a *first class citizen* in the system. As a result, its features are fixed, and the designer is forced to fit data and databases structures under the replication model.

If replication is to be handled as a first class entity, it must be modeled on top of a database engine, which is covered in the next subsection.

2.2. **Replication as a first class citizen**

Under this vision, an independent replication layer is introduced, providing much more flexibility. Even if replication constrains changes, to apply these new requirements to the database framework is easier, since replication is modeled in an independent and separate way.

The replication's status upgrade requires incorporating replication as a main *architectural* component, interacting and communicating with the database engine in a bidirectional flow. This is illustrated in figure 3.

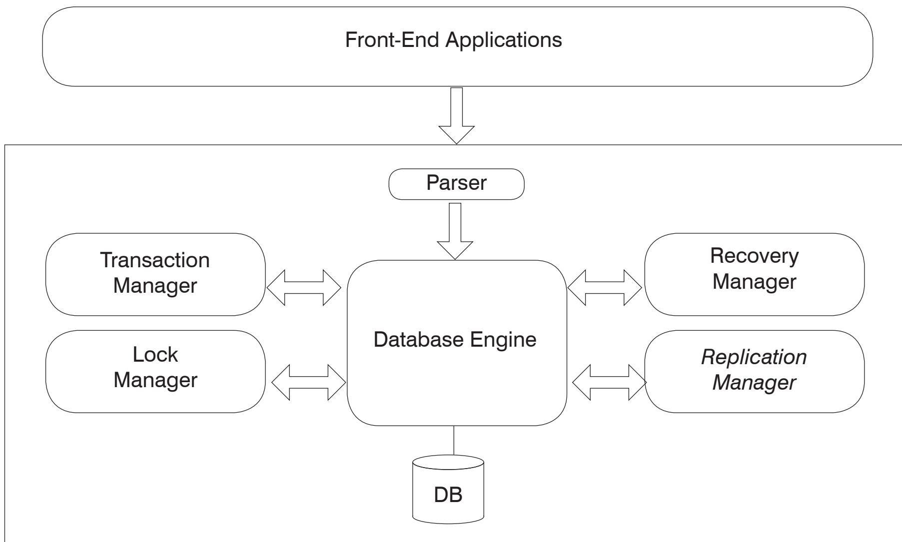

Figure 3: A simplified DataBase Engine architectural description with replication as a first class citizen

The configuration for the replication component includes knowing which nodes represent servers, connections to the database, and the data structure and the operations to be replicated. Each of these responsibilities are further described in the next section.

Model responsibilities

The three main responsibilities for the replication component are described next.

Connection

Connecting and disconnecting from the database is a basic feature for a replication component. The fact that the replication component is modeled as a separate component, on top of the database engine, helps to ensure reusability requirements with respect to specific issues such as location of the database or drivers used since these items can be described at a high level, and then become instantiated at the concrete system (similar to abstract and concrete classes or methods in OO world).

Servers

The distributed database consists of multiple nodes connected according to some configuration. One or multiple nodes will play the server role. In general, there is a main server, against which database operations data are performed, and one or more secondary servers, which maintain replicated data. The replication component must "capture" somehow operations performed against the main server, and replicate them in secondary servers. When this situation is not possible (for example, a server is down) the operation is performed against any server (primary or secondary) and then logged (all pending queries are maintained in a log) so that it can be replicated later on.

The server’s configuration is suitable to many changes (such as adding/ removing servers), so it is important to take this fact into account while considering the database system evolution. This results in keeping the server’s configuration apart from the database system, in a totally transparent way. Another benefit obtained through this indirection is that the replication component can elaborate the best routing algorithm for the replicated information, alleviating the database engine from extra work.

Data and operations to be replicated

Not every data is to be replicated, and the same happens with database operations. Having the entire database replicated is ideal, but not possible even for small or medium databases. A good replication strategy implies selecting appropriated data and operations to be replicated. Statistics, history of the data base, catalog information, and others are main inputs for the process of selecting data and operations that are to be replicated. Once defined, the replication component ìmonitorsî database execution, and applies replication features according to the configurations.

Highlights

As it can be seen, the information needed in the three cases is configurable externally so that the database system remains unaware of replication behavior, achieving flexibility and reusability requirements besides easing system evolution.

3. PROOF-OF-CONCEPT IMPLEMENTATION

In this section we describe our implementation for the replication model presented before, introducing our framework called ***Sigma***. We present in fact two implementations.

The first one has been developed using object-oriented technology in the Java programming language and the second one using aspect-oriented technology, in AspectJ, a Java extension to AOP mechanisms, and one of the most popular and widely known aspect-oriented programming language. Sigma is focused on *relational databases* using the MySQL Connector/J 3.1 JDBC driver.

After both implementations are introduced, we conclude the section analyzing which implementation satisfies better the responsibilities presented in the conceptual model.

3.1. Sigma object-oriented implementation

The object-oriented implementation consists of the following components:

- an interface *IDAO*, describing the methods to be implemented to interact with database, representing the Data Access Layer.

- a DBManager class, implementing key features involving connection to database, servers configuration and logs for replication.

- a main class UserDAO, which implements the IDAO interface, and access to the database through a private field manager of type DBManager.

Figure 4 shows the interaction between these components.

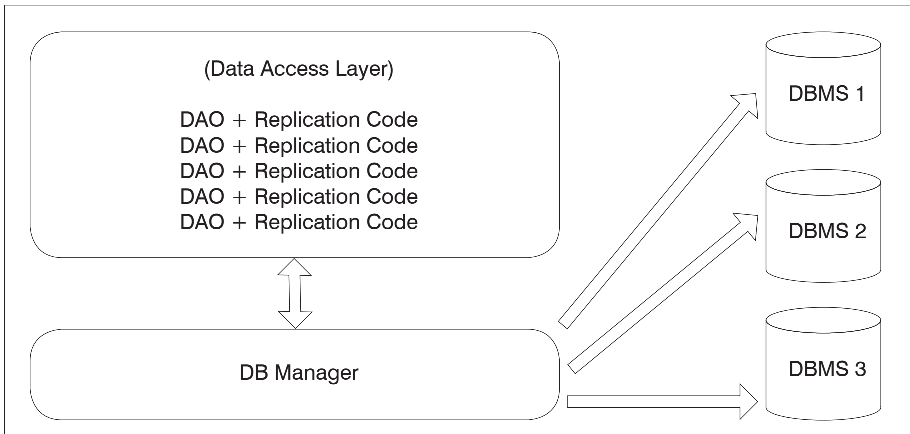

Figure 4: Scheme for the object-oriented implementation.

Due to its crosscutting nature, replication code is present all over the system. For example, after an operation is performed, it is necessary to include extra code to replicate the new information obtained. This code is repeated in each of these operations, *insert, remove, modify*, etc. This extra code represents an ìalienî code since it is not related to operation itself but to replication. Listing 1 shows this situation within the insert method.

```
1:public void insert(Object object) throws Exception
2:{
3:     UserObject user = (UserObject) object;
4:          query = "INSERT INTO Users(name,phone,email,country) VALUES('" +
          user.getName() + "','"
5:          +  user.getPhone() + "','" + user.getEmail() + "','" + user.getCountry()
          + "')";
6:          try
7:          {
8:               //get available connection from DBManager
9:               Statement stmt = manager.getConnection().createStatement();
10:              //request DBManager to execute all the pending queries (if any) using
                 this connection
11:              manager.executePendingQueries();
12:              //execute the insert query
13:              stmt.executeUpdate(query);
14:    }
15:    catch(Exception ex)
16:    {
```

```
17:            //exception handling code goes here
18:    }
19:    finally
20:    {
21:            //request DBManager to replicate this query on the other servers too
22:            manager.replicateQuery(query);
23:    }
24:}
```

Listing 1: Replication as a crosscutting concern.

Besides the code for the insert method, code for replication is also present: line 11 shows the invocation to the method *executePendingQueries* from the DBManager class, which executes all the pending queries (if any) using this connection. In the finally block (lines 19-23) the method replicateQuery from the DBManager class is invoked. This method will replicate the operation just performed before in all the others servers. In case where replication is not possible, an entry is added in the pending query log.

Although implemented on top of a database engine, replication itself does not constitute an independent replication layer. Changes in server configuration, database connection or in the replication policy imply checking all over the system for possible modifications. This problem is known as the ìtyranny of the dominant decomposition” [17]: the system behavior is ruled by a dominant factor, causing structural problems for the remaining factors of the system. In this case, replication code is widespread through all the system. Taking this into account, we conclude that the object-oriented version did not behave as specified in the conceptual model.

3.2. **Sigma aspect-oriented implementation**

This version was implemented using Eclipse 3.2, and the AJDT 1.4 plug-in for the AspectJ language. Modeling replication as an **aspect** allows defining an independent replication layer, achieving all the objectives required in the conceptual model. The implementation is described next.

Replication as an Aspect

Since replication is implemented as an aspect, it is implemented in a separate and independent way. First of all, a pointcut captures all the operations and information that will be replicated. An abstract pointcut can be defined, and then implemented in concrete pointcuts, obtaining the possibility of reusing the replication aspect in other environments. Given this pointcut, a *before* advice is introduced, specifying that before every operation is performed pending queries are executed, thus synchronizing all the information on the servers. Similarly, an after advice is also introduced, which replicates the operation in all the other servers or log the queries if secondary servers are down. The aspect skeleton is illustrated in listing 2.

```
pointcut replicatedOperations() : execution <enumerate all replicable operations>;

before() replicated Operations() {
<establish connection to database>
<Execute pending queries and synchronize servers>
}

after() replicated Operations() {
<replicate query in secondarys servers>
}
```

Listing 2: Replication Aspect Skeleton

Connectivity to the database is also included within the aspect, in an aspect method *createConnections*. Up to now, the replication component modeled as an aspect includes the first and the third responsibilities, namely connection and data and operations to be replicated. Regarding servers configuration, it is modeled also within the aspect through private fields[1], completing all the responsibilities required in the conceptual model.

[1] In future versions of Sigma, these configurations will be implemented in xml files.

Interaction

Aspect interaction with the other components is shown in figure 5. The DBManager class is no longer required, since all its behavior is now encapsulated within the replication aspect, including database connection and server configuration.

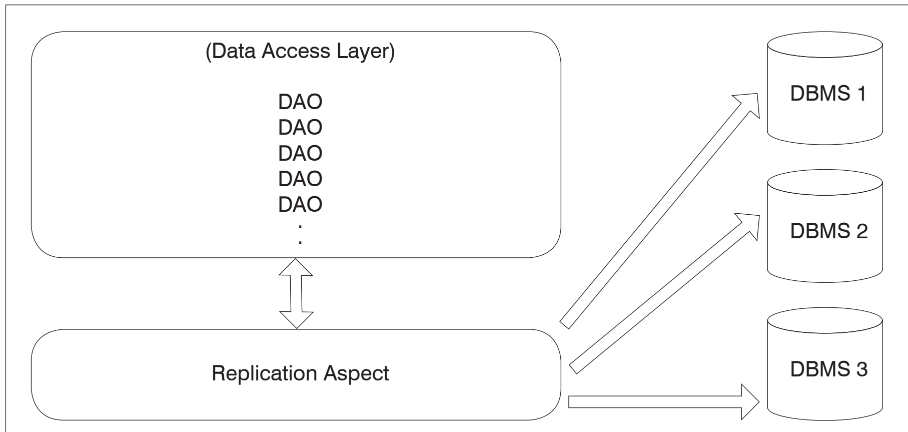

Figure 5: Scheme for the aspect-oriented implementation.

The replication code is no longer distributed through the system. We can now revisit the code for insert method, shown in listing 3. The code involves only actions regarding an insert event.

```
public void insert(Object object) throws Exception
{
        UserObject user = (UserObject) object;
        query = "INSERT INTO Users(name,phone,email,country) VALUES('" + user.
getName() + "','"
          + user.getPhone() + "','" + user.getEmail() + "','" + user.getCountry() + "')";
        Statement stmt = connection.createStatement();
        stmt.executeUpdate(query);
}
```

Listing 3: The insert method revisited.

Besides modularity advantages, the aspect-oriented implementation allows introducing a separate replication layer. Reusability can be managed specifying abstract pointcuts, and evolution becomes easier, since replication code is isolated. Connection to database, server configuration, and data and operations to be replicated can be specified at a higher level, totally independent from the rest of the application. Due to all the reasons exposed previously, we can conclude that the aspect-oriented version clearly satisfied the conceptual model presented.

Fragility and oblivious discussion

Despite the use of abstract pointcuts, the replication aspect cannot be completely unaware of base code evolution. Because AspectJ mechanisms rely on the code syntax's, it suffers from the *pointcut fragility* problem [18]: when base code evolves, aspects must evolve too. However, this fragility is inherent to AspectJ, and not to our replication model. Our conceptual model could be implemented instead under other mechanisms tackling fragility problem as SetPoint [19], [20], [21], CAESAR [22], Aspectual Views [23], among others. AspectJ was chosen due to its wide acceptance, besides being one of the most popular aspect-oriented programming languages.

Regarding obliviousness [24], the base code does not have to be prepared in anyway, since execution points that the replication aspect is interested in are very reachable, which greatly facilitates inserting aspect behavior.

4. RELATED & FUTURE WORK

As said before, aspect-oriented technology has been applied successfully in databases [8-13], specially considering object-oriented databases and modeling persistence as an aspect. In this work, we build on the top of these proposals introducing a Java framework called **Sigma** for *Relational DataBases* where a core feature as replication is implemented over a database independent replication layer.

Replication has been implemented as an aspect in other distributed contexts, as component-based programming [25] and dynamic flow control [26]. To our

best knowledge, replication, although mentioned as candidate requirement to be implemented as an aspect, was unexplored in a database context.

We will continue improving our framework, incorporating configurations details into xml files, and adding more functionality, such as incorporating a transaction manager as an aspect. We also would like to analyze performance issues and compare our framework with other successful frameworks like Spring [27].

5. CONCLUSION

In distributed databases, replication is a key concept in order to maintain data consistence, and a secure mechanism in cases of servers' failure. Internal models for replication provided in most engines lack flexibility and suffer from other problems as bottleneck communications. Taking this into account, we propose a *conceptual* model where replication is considered as a main architectural component, built on the top of databases engines which allows introducing a separate and independent replication layer. Two implementations are presented, an object oriented one and an aspect oriented one. Whereas the object oriented versions suffers from the consequences of not managing correctly crosscutting concerns, the aspect-oriented version adapted perfectly to the conceptual model.

REFERENCES

[1] R. Chitchyan, A. Rashid, P. Sawyer, A. Garcia, M. Pinto Alarcon, et al., "Survey of aspect-oriented analysis and design approaches". *Technical Report AOSD-Europe-ULANC-9*, AOSD-Europe, May 2005.

[2] S. Katz, "A Survey of Verification and Static Analysis for Aspects". *Technical Report AOSD-Europe*. July 2005.

[3] R. Douence and D. Le Botlan, "Towards a Taxonomy of AOP Semantics". *Technical Report AOSD-Europe*. July 2005.

[4] J. Brichau and M. Haupt, "Survey of aspect-oriented languages and execution models". *Technical Report AOSD-Europe- VUB-01, AOSD-Europe*, May 2005.

[5] Neil Loughran et al. "Survey of aspect-oriented middleware". *Technical Report AOSD-Europe- ULANC-10,* AOSD-Europe, June 2005.

[6] A. Colyer, G. Blair, and A. Rashid, "Managing complexity in middleware." *Workshop on Aspects, Components and Patterns for Infrastructure Software (held with AOSD 2003)*, Boston, USA.

[7] A. Colyer and A. Clement, "Largescale AOSD for middleware". *Proceedings of the 3rd international conference on Aspect-oriented software development*, pp 56-65, 2004.

[8] R. Green and A. Rashid, "An Aspect-Oriented Framework for Scheme Evolution in Object-Oriented Databases", *1st Workshop on Aspects, Components, and Patterns for Infrastructure Software (held with AOSD 2002).*

[9] A. Rashid, and R. Chitchyan. "Persistence as an Aspect", *2nd International Conference on Aspect-Oriented Software Development. ACM*, pp 120-129, 2003.

[10] A. Rashid, A. and P. Sawyer, "Object Database Evolution using Separation of Concerns", *ACM SIGMOD Record, vol 29, no 4*, pp 26-33, 2000.

[11] A. Rashid, "Aspect-Oriented Scheme Evolution in Object Databases: A Comparative Case Study", *Workshop on Unanticipated Software Evolution (held in conjunction with European Conference on Object-Oriented Programming ECOOP 2002.*

[12] A. Rashid and E. Pulvermueller. "From Object-Oriented to Aspect-Oriented Databases". *11th International Conference on Database and Expert Systems Applications (DEXA). Springer-Verlag Lecture Notes in Computer Science, vol 1873*, pp 125-134, 2000.

[13] A. Rashid and N. Leidenfrost, "Supporting Flexible Object Database Evolution with Aspects", *International Conference on Generative Programming and Component Engineering (GPCE 2004).*

[14] E. Gamma et al.,"Design Patterns Elements of Reusable Object-Oriented Software", *Addison-Wesley*, 1994.

[15] C.Thompson, "Comparing Three Leading DBMS vendor's approaches to Replication". Available: http://www.dbmsmag.com/9705d15.html

[16] International Business Machines (IBM), "OGSA-DAI replication". Available: http://www.aiai.ed.ac.uk/~ychen/ibm_ogsadai/replication-technologies.html

[17] P. Tarr et al., "N degrees of separation: Multi-dimensional separation of concerns", *In Proc. 21st Int'l Conf. Software Engineering (ICSE'1999)*, pp 107 – 119, IEEE Computer Society Press, May 1999.

[18] C.Koppen and M.Stoerzer. "Pcdiff: Attacking the fragile pointcut problem", *EIWAS 2004.*

[19] R.Altman, A.Cyment and N.Kicillof. "On the need for SetPoints", *EIWAS 2005.*

[20] A. Cyment et al.,"Improving AOP systems' evolvability by decoupling advices from base code", *3rd ECOOP'2006 Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE)*, 2006.

[21] A. Cyment, N. Kicillof and F. Asteasuain, "Enhancing model-based AOP with behavior representation", *2nd ECOOP'2006 Workshop on Models and Aspects (MDD&AOSD)*, 2006.

[22] M. Mezini and K. Ostermann, "Untangling crosscutting models with CAESAR. In Aspect-Oriented Software Development", *Addison-Wesley, Boston, 2005,* pages 165–199.

[23] A.Kellens et al., "Managing the Evolution of Aspect-Oriented Software with Model-based Pointcuts", *ECOOP 2006.*

[24] R.Filman and D.Friedman, " Aspect-oriented programming is quantification and obliviousness", *Advanced Separation of Concerns. OOPSLA 2000.*

[25] Sara Bouchenak et al., "Can Aspects Be Injected? Experience with Replication and Protection", *International Symposium on Distributed Objects and Applications (DOA 2003)*, Catania, Italy, November 2003.

[26] J. Fabry, "Replication as an Aspect", *ECOOP Workshops*, 1998.

[27] Spring Official Site: http://www.springframework.org/