An Aspect-Oriented Framework for Orthogonal Persistence

Rui Humberto R. Pereira
ISCAP
Instituto Politécnico do Porto
Porto, Portugal
rhp@iscap.ipp.pt

J.Baltasar García Perez-Schofield
Departamento de Informática
Universidad de Vigo
Vigo, España
jbgarcia@uvigo.es

***Abstract*— The life cycle of software applications in general is very short and with extreme volatile requirements. Within these conditions programmers need development tools and techniques with an extreme level of productivity. We consider the code reuse as the most prominent approach to solve that problem. Our proposal uses the advantages provided by the Aspect-Oriented Programming in order to build a reusable framework capable to turn both programmer and application oblivious as far as data persistence is concerned, thus avoiding the need to write any line of code about that concern. Besides the benefits to productivity, the software quality increases.**

This paper describes the actual state of the art, identifying the main challenge to build a complete and reusable framework for Orthogonal Persistence in concurrent environments with support for transactions. The present work also includes a successfully developed prototype of that framework, capable of freeing the programmer of implementing any read or write data operations. This prototype is supported by an object oriented database and, in the future, will also use a relational database and have support for transactions.

Keywords: aspect, oriented, programming, orthogonal, persistence

I. INTRODUCTION

An Orthogonal Persistent system aims at giving [1-4] programming transparency, more productivity and a programming less prone to errors, in what the persistence aspect is concerned. This paper analyzes the requirements of a system on that paradigm and the viability of the implementation of that concern in a framework that is truly aspect-oriented [5] in the context of the Object Oriented Programming. A successful implementation of that concern as an aspect turns it possible to modularize completely the code responsible for storing and retrieving the object data on the secondary storage device, in a way that the program doesn't need to be prepared to take care of that work. Moreover, that implementation could be totally reused, as a generic framework, in any other program without any modifications. Despite the Orthogonal Persistence benefits, the programmer can still be oblivious about that concern.

Section II provides an overview of the aspect-oriented paradigm presenting the main concepts. In the following section are presented the motivations and challenges of applying an aspect-oriented programming in order to implement orthogonal persistence in object-oriented applications. On the section IV we present our prototype, a reusable aspect oriented framework for object Orthogonal Persistence in Java environment. The section V discusses some related work and the last two sections present our plans to future work and some conclusions obtained during the present work.

II. ASPECT-ORIENED PROGRAMMING

In general, any software application has one or more concerns: Logic, Presentation, Distribution, Persistence, etc. The programmer tries to modularize the code, with the best organization, in order to minimize the tangle of those concerns and maximize de code reusing. The procedures, the inheritance and the classes, in the Object Oriented Programming, as well as programming patterns provide a good level of code reusing. However, they are incapable of cross-cutting the application concerns, separating it completely.

The Aspect Oriented Programming (AOP) [5] consists in a programming technique that allows the separation of those concerns. In an object oriented context, a concern that is transversal to all objects could be segregated from those objects and putted in a specialized object called Aspect, while the remaining concerns, that are specific from each object, maintain themselves implemented in the object class.

The ability of quantification [6] of all join points in the objects code make it possible to turn those objects oblivious [6] about the concern aspects. This characteristic of Quantification, present in the AOP, allows the definition of crosscutting expressions that identify the points where the aspect code must be woven. By doing this, the object code doesn't need to be

prepared to use aspects. To this second characteristic present in the AOP is called Obliviousness [6]

III. DATA PERSISTENCE AS AN ASPECT

The persistence is a concern that is transversal to any component (objects, functions or procedures) in the majority of the software. Due to that, it is frequently considered as good example of crosscutting concern that can be aspectized.

Being the persistence one of the many aspects that a developer has to deal with, during software development, certainly it is not the most important one for the software project. The modularisation of the code responsible of data storage and retrieval, as an aspect, eliminates the work of writing any line of code by the application programmer transferring that concern to an underlined framework. The application, in this scenario, is developed without any need of explicitly write or read the persistent data from the storage device. The programmer only need to distinguish which objects are short lived, in memory, or long lived on a storage device, independently of its type. It is notorious the extreme level of code reutilization that can be achieved by this kind of framework. However, as you will see forward, in the related work, nowadays there are still no systems able to fully aspectize that crosscutting concern [7].

A. Orthogonality on object persistence

In the last years several researchers have study the data persistence theme introducing several new concepts as long the as the research was evolving. The persistence, as Atkinson et al [1] conceptualized, introduced the concept of Orthogonal Persistence, and later, for objects, by following the three principles that Atkinson and Morrison [8; 9] formulated, giving to the programmers total abstraction of their data on objects, allowing code reuse, the focus on application logic and data consistency. Those tree principles are desirable characteristics that any system should have in its persistent layer.

Persistence independence - The same code should be applicable for both transient objects and persistence objects. The advantages are obvious allowing the code reusability by the abstraction of the kind data object. This also indicates that the persistence framework semantics must not be changed in both cases. This principle is also known as transparent persistence.

Type orthogonality - All objects can be persistent or transient irrespective of their types, sizes or any other property. Any type of object, without exception, can be long-lived or transient.

Persistence Identification – The form of identifies and persist object is orthogonal, i.e., all objects are available from one, or more, common root [9], and all it's related, are accessed on same way. This compromise guaranties a uniform mechanism of retrieve the stored objects and its relations. This principle is also referred as reachability for many researchers [1] but this initial term was superseded for Transitive Persistence a more suggestive ODMG[1] term.

[1] http://www.odmg.org

This paradigm of persistence gives total transparency to the programming language while it interacts with the existent data on the underline repository, whatever be the model, Relational or Object Oriented, its technology or location. On this field, concepts like Safe Queries [10] and Native Queries [11], may help the better understanding of the orthogonalitly of the persistence and it potentialities.

However, all this carelessness given to the programmer puts several challenges to the management system database designers and many of them are not yet have been solved. In fact, this paradigm of data object persistence turns the problem very complex and, moreover, also introduces performance issues, special at system main memory management level. These issues motivated some authors like Cooper and Wise [12], to criticise the Orthogonal Persistence and advocate another alternative model less restrictive named as Type-Orthogonal Persistence opposing to the unrestricted model of Orthogonal Persistence presented by Atkinson and Morrison [9]. Analyzing the Cooper and Wise arguments we conclude that are essentially performance issues and not for restrictions made to the programmer or the language.

Despite of the many challenges posed by this form of object persistence, the advantages are tremendous not only at level of code reuse, as already mentioned, but also on others point of view like the data type safety checking, reducing the coding errors, promoting a better code organization, on the improving the applications refactoring processes, a practices very common on most modern agile methodologies of software developing, and also on solving the object-relational impedance mismatch when repository is a relational databases.

This form of persistence has several implementations, in some cases not totally compliant with the concept. PJama [3], OPJ [4], Visual Zero [2; 13] and Thor [14], are examples of those systems that implement Orthogonal Persistence. Some object oriented databases, as well some object-relational mapping tools, also implement some level of orthogonality.

B. Is AOP suitable for Orthogonal Persistence?

An AOP language allows quantify programmatic assertions over the code, at any point, that is oblivious to those assertions. This ability turns the AOP well suited to develop the orthogonal persistence concern for that system as an aspect. Since this kind of persistence advocate total transparency to application and programmer, two objects of the same class could have completely different persistence behaviours in multiple distinct contexts.

Regarding to the Type Orthgonality principle, all objects, of any class, could be persistent or transient. In the generally of the frameworks (like Hibernate, EJB or JDO), the persistence of an object is achieved by inheritance or by implementing special class or interfaces, that conditions completely that principle of orthogonality. Using AOP, because we can quantify any pointcut, on any type object, the weaver mechanism could weave the code responsible for persistence on the object. This avoids the requirement of that object extending any super class or implements any interface.

Considering the Persistence Identification principle, the fact of an object is long lived or transient depends only if it is

related directly or indirectly from a persistent root (an object). Since the aspect code have conditions to know the semantic of relationship among the objects, it is possible to achieve that principle by examining the data contained in objects and their relationships.

Considering the last two arguments, that any object can be persistent, despite of its type, and the orthogonal way of identification of those objects, so it is not needed to have a special care handling those persistent objects in comparison to transient ones. Since the aspect code can by itself distinguish a short-term from long-term object, by examining the relationships among objects up to the persistent root, the code contributed by the persistence aspect makes the logic orthogonal to the persistence concern. This way, the principle of persistence independence is achieved.

C. *Issues when aspectizing object persistence*

The aspectization of the persistence concern raises several implementation problems that need to be solved. As described early, at conceptual level it may seem easy, the persistence is a concern and the AOP is a programming technique used to implement concerns, but in practice isn't so easy.

During the analysis of the persistence concern to implement as an aspect, that provides a CRUD (Create, read, update and delete) group of four operations while meeting the ACID (Atomicity, Consistency, Isolation, Durability) properties on transactions, emerges a vast and diverse set of issues. From the importance of the role played by the Object Id, the capability of distinguishing transient and persistent objects instances of the same class, object creation versus object instantiation, object deletion versus object instance destruction, until the performance challenges are issues that have different contours depending of the underling type (relation/object) of persistent storage.

IV. FRAMEWORK FOR ORTHOGONAL PERSISTENCE

In our study we have developed a prototype [15] using AspectJ, a compile-time Java based AOP language [16]. This prototype, in order to provide object persistence services, uses db4objects a pure object database [17]. On next versions of this prototype, also it will be able to use a relational data base. The prototype conceptual basis modularizes, in a natural way, the system architecture on aspects turning the use of an object or relational database as system aspect as any other.

This prototype is actually a framework capable to be totally reused in any Java application, providing orthogonal persistence services to that applications in a oblivious [6] way. Those applications don't care about the persistence aspect because this framework takes care all about it.

Like in the Atkison and Morrison model [9], this framework provides a special object, that we have called PersistentRoot, that gives access to all persistent data objects, besides other low level services like caching. The persistent data object can be of any class since it source code it is available at compile moment, an AspectJ limitation.

The following code it is one of the demonstration program of this framework.

```
01 CPersistentRoot psRoot=new CPersistentRoot();
02 Degree computerScience=new Degree("01","Computer Science");
03 computerScience.setCourses(new Course[]{
       new Course("MAT","Mathematics"),
       new Course("AI","Artificial Intelligence"),
       new Course("OT","Object Technology")});
04 Student ana=new Student(1,"Ana",new Address(...),null,20);
05 Student rui=new Student(1,"Rui",new Address(...),null,25);
06 psRoot.setRootObject("enroll_Ana_MAT",
       new Enrollment(ana,computerScience.getCourses()[0]));
07 psRoot.setRootObject("enroll_Ana_AI",
       new Enrollment(ana,computerScience.getCourses()[1]));
08 psRoot.setRootObject("enroll_Ana_OT",
       new Enrollment(ana,computerScience.getCourses()[2]));
09 psRoot.setRootObject("enroll_Rui_AI",
       new Enrollment(rui,computerScience.getCourses()[1]));
10 rui.setAddress(null);
11 rui.setAddress(new Address(...));
```

Figure 1. Application example with an n-n relationship

At line 01, the persistent root it is created. In the next four lines several data objects are instantiated, by it constructors, and interrelated. At this moment, all those objects are transient. In the other next four lines those objects, and new others, are made persistent simply by the fact of be related with the persistent root object. At the line 10, the student address object it is deleted from the database by the fact it student object be persistent and it reference to the address object it is made null. Next it is created again in a new Address object. Obviously the line 10 isn't necessary.

No data object needs to call any method to achieve persistence. There is not a special class implementing a specific Java interface or extending a super class, as well. That behaviour is implemented as aspects coded in AspectJ.

The framework architecture, presented in figure 2, it is organized in three main components/layers: the PersistentRoot object, object persistence aspect and database persistence aspect. The first element acts as interface between the upper layer application and the framework itself. The second implements the object persistence aspect, independently of the type of the underline data storage. Intercepts all read and write operations on object fields and also manages the object cache. The third element, the database persistence aspect, takes care of all issues related with database connection and cache data persistence. At the lowest level it is the object database.

This framework prototype uses intensively the Java Reflection API to know the internal structure of the object and it fields. This is crucial to the object data hierarchy drilldown until the object primitive data types. The prototype provides support to object arrays. This particular type of object needs a specialized treatment since it must be disaggregated in all individuals objects of one or more types. This procedure isn't need in an object database, like the one used, it was an option made to achieve a normalized mechanism in the object persistence aspect to enable a system future evolution to a relational databases, and for cache performance issues allowing a granularity until the elementary object.

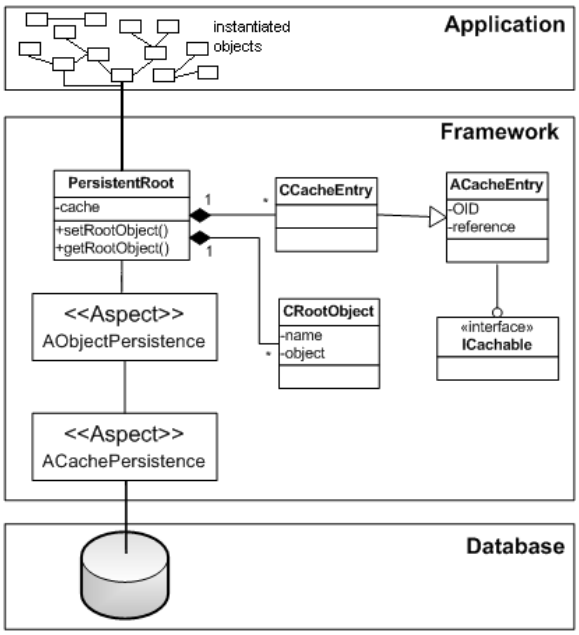

Figure 2. System and framework architecture

In order to the object persistence aspect distinguish between two objects, of a same class, to know which one is persistent or is transient, that is done by a search on the system cache and also on all related objects. An instantiated object that is referred in cache has an Object Id, so it is long-lived. This solution avoids any preparation at type, code or object level [7]. By this way, the prototype meets the Persistence independence and Type Orthogonality principles.

The use of the framework it is very simple. As already referred, the application objects source code it is needed for the woven and compiling process.

To write an application, just add the framework package and start write code with any hurry about persistence. To achieve final binary code, the process it is very simple and completely automated by the Integrated Development Environment (IDE)[2]. The following diagram illustrates all the phases since the source to the final executable in bytecode.

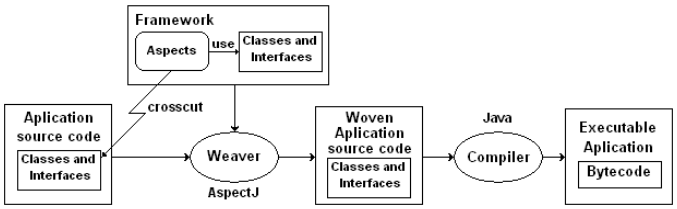

Figure 3. Executable generation

V. RELATED WORK

Soares et al [18; 19] present their experience while refactoring a web application, a Health Watcher system, modularizing all code related with distribution and persistence concerns in AspectJ aspects.

This experience demonstrates the viability and shows some benefits of apply aspects in real word applications while also indentified some drawbacks of the AspectJ. But the way that those two aspects were implemented, in our opinion, moves away from the essential idea of aspect-oriented programming by the fact of limiting the done work to a reorganization of the code, specific the application, transferring that code to specialized modules, the AspectJ aspects. This way of aspectization does only allow to reuse the specific code inside the application or on the several modules (client, server and other in future). Code reusability is limited to some interfaces and, on other hand, all SQL statements are hard-coded in the aspects. The distributed architecture, applying a Facade pattern, itself, already enables a good level of code reuse.

On specific concern of persistence, a data abstraction layer, based on business data collections that allow the application works transparently in two versions of data, persistent and non-persistent data, was implemented in aspects. This interesting idea it is a common programming pattern used in many applications, in this case was implemented on aspects that turns their work an important experience.

Another important work [20], that really presents the persistence as an aspect, describes an aspect-oriented framework for persistence. This solution, by using the reflection capabilities, on the presented case the Java Reflection API, and a specialized aspect for translation Object-Relational, frees the programmer of doing any mapping from objects in memory to their related tables on the relational data base.

In order to identify the persistent objects, this framework requires that all those objects must have a special and common class as their super class. The PersistentRoot, beside of providing deletion functionalities, also provides a mean to define a pointcut that quantify the join points where the persistence code must be woven. This super class can be totally reused in any other application, but this solution breaks the Atkinson and Morrison principle of the type Orthogonality [9] conditioning the applicability to objects that extend the PersistentRoot class.

The authors of this work also identified two issues difficult of aspectize in an oblivious way: the data retrieval, because it's declarative nature, and the deletion of objects because the difficult of know when an object is eliminated. In fact, the mechanisms of data retrieval have a declarative nature. But that nature also exists when we are searching objects in memory with large data collections. So the question is how to obtain a common mechanism capable of be applied both in persistent and non-persistent data. On this matter, the obliviousness about the semantic of the data, doesn't have sense, because that semantic is part of the application logic. Native Queries, in our opinion, already provide the needed transparency allowing a safe and common way to manipulate objects. A second issue is the difficulty in detection when the object is deleted, because in Java, and many other platforms, there is a garbage collector that do the job of eliminate the unneeded objects from the memory. So to resolve this problem, they introduce a field, it getter method and a method that allow an explicit object deletion invoking. This field and methods, implemented in the

[2] We have used Eclipse - http://eclipse.org

super class PersistentRoot, allows the aspect code to know when the object must be deleted from persistent store. Our prototype doesn't suffer of those two limitations as described above.

Al-Mansari et al [7] presented a complete solution for Orthogonal Persistence based on two new concepts: Path Expression Pointcuts (PEP) and Persisting Containers. The Path Expression Pointcuts [21] are a new pointcut construct that applies path expressions on object relationships in the same fashion that XPath do in a XML file to find a node. PEP are a powerful quantify mechanism that is capable of identify transversally all join points that match with a given object relationship path expression, providing the aspects with access to non-local object information. As explained above, this non-local information it is crucial to identify which object are directed or indirectly related with the Persistent Root [9]. The following expression exemplifies this concept.

```
pointcut trapUpdates(PersistentList pl,Object o):
set(* *) && target(o) && path(pl -/->o);
```

example extracted from: [7]

In this example, the path expression "pl -/-> o" match with all objects relations, with any length, from an PersistentList object pl to a object o. The pl object is the root and the object o is the aspect target object.

The Persisting Container [7] is a special object maintained by a aspect to provide persistent services. The idea is similar to spontaneous containers [22]. This container, besides play the role of Persistent Root, is capable of spontaneously give persistence capabilities to objects related with.

This work was the most relevant one of all the reviewed research projects and that goes closest to Orthogonal Persistence and Obliviousness. Don't require any preparation at type, code or object level [7] as happens in the previous presented works. But this solution does not yet have an efficient implementation of Path Expression Pointcuts. This issue is referred by the authors as future work. As already referred, on our prototype, we have used the information cache to obtain that object non-local information.

Transactions and failure has been referred by authors [7; 19; 20; 23; 24] as impossible to be totally aspectizable and turn the programmer oblivious about both issues. Kienzle and Guerraoui [24] made a detailed study about the aspectazition of those concerns and classify that in three levels of different ambition of aspectization:

- Transaction semantics – All semantic about the transaction is hidden. They consider as impossible to achieve.
- Transaction interfaces – On this approach, the transaction interfaces (begin, commit, abort, etc) are transferred from the functional parts to specific aspects. With this solution, a method can be made transactional by encapsulating it in a around advice surrounding the advice proceed statement with the transaction interfaces. This approach leads to some problems. The roll back operation must be done externally to the aspect turning it an intricate code where part of then is in the aspect and the failure treatment remain on the functional part of the program. The authors also refer this level of aspectization as impeditive to collaboration among threads, by the fact they can't enter in each other transactions. They also argue that makes no sense to turn all applications objects into transactional objects, only methods that have transactional behavior must be intercepted and aspectized.
- Transaction mechanisms – This less ambitious level, it is very close to our plans to implement transactions, despite in this case the recovery and undo mechanisms are implemented in the OPTIMA [24], the framework used in their experiences. Our approach extends this one, by providing a special Transaction Context object that implements all those mechanisms and give selectively transactional behavior to the method objects.

The system architecture presented by Soares et al. [19] naturally resolve most of the transaction problems by using a Facade pattern, moreover, the implemented aspects are too specific for the developed system allowing a very low level of code reuse.

VI. Future Work

The Al-Mansari et al [7] proposed solution puts a practical problem when developing a prototype because the essential mechanism, the Path Expressions Pointcut, does not yet have any implementation. So, the developed prototype presented in this work, implement an alternative mechanism that provides the same non-local information. So the future work will may pass to find a solution to get that precious information, using Path Expressions Pointcut. With this kind of expressions, the advice is called with much less often than our actual solution.

This work presented the PersistentRoot object that provides persistence services on the prototype. Those services, at current version, don't include any transaction capabilities. Future work will use important Kienzle and Guerraoui [24] work results. Our approach aspectizes the transaction mechanisms and provides an interface to the application interact with objects in a transactional context. A special object, that we have called TransactionContext, together with the existing PersistentRoot, will be able to save the transaction state, give transaction behaviour and failure treatment mechanisms, to any associated object in the same paradigm that PersistentRoot gives persistence services to any object related with it.

As already referred above, our prototype use an object oriented data base. Considering the actual importance of the relational databases at the performance level and because they have a considerable market share, the prototype must be able to use them as information repository in order apply our framework on a real life production system.

VII. Conclusions

The existence of an object root [9] (in the present work, the PersistentRoot), plays an important role to any system that really wants to be obliviousness about the persistence concern. This element works as interface between the upper layer of

applications and the underlying data store mechanism. It provides the needed persistence mechanisms enabling the conditions to aspectization of the persistence concern. Initalization and finalization of connection, data reads and writes can easily be wrapped by an advice that does the all job. Thus, it is possible to obtain a framework that is capable of supply orthogonal persistence services in an oblivious fashion.

In the specific case of failed transactions, we among other authors [24] consider that does not make sense to fully aspectize those concerns, because that is part of the application logic concern.

The developed prototype it is a generic and reusable framework and it is capable turn programmer and application oblivious about the persistence concern. We believe that freeing the programmer of taking care of that concern improves the quality of the code, besides of giving a valuable boost in productivity.

REFERENCES

[1] Atkinson MP, Bailey PJ, Chisholm KJ, Cockshott PW and Morrison R. "*An approach to persistent programming*," In The Computer Journal, pp. 360-365, 1983.

[2] García Perez-Schofield B, García Roselló E, Pérez Cota M and Ortín Soler F. "*Technical Report - PROWL, a computer language for prototype-based objectoriented programming.*,"Universidad de Vigo - Departamento de Informática, 2008.

[3] Atkinson MP, Daynès L, Jordan MJ, Printezis T and Spence S. "*An orthogonally persistent Java*," In SIGMOD Rec., pp. 68-75, 1996.

[4] Marquez A, Blackburn S, Mercer G and Zigman JN. "*Implementing orthogonally persistent Java*," In Pos-9: revised papers from the 9th international workshop on persistent object systems, pp. 247-261, 2001.

[5] Kiczales JG, Lamping J, Mendhekar A, Maeda C, Lopes C, Loingtier J and Irwin J. "*Aspect-Oriented Programming*," In European Conference on Object-Oriented Programming, pp. 220-242, 1997.

[6] Filman RE and Friedman DP. "*Aspect-oriented programming is quantification and obliviousness*," RIACS, 2000.

[7] Al-Mansari M, Hanenberg S and Unland R. "*Orthogonal persistence and AOP: a balancing act*," In Acp4is '07: proceedings of the 6th workshop on aspects, components, and patterns for infrastructure software, pp. 2,2007.

[8] Atkinson MP. "*Persistence and Java - A Balancing Act*," In Proceedings of Objects and Databases, International Symposium at ECOOP 2000. Sophia Antipolis, France, June 2000. Published as Lecture Notes in Computer Science, (Dittrich, KR et al Eds). Volume No. 1944., pp. 1-31, 2000.

[9] Atkinson M and Morrison R. "*Orthogonally persistent object systems*," In The VLDB Journal, pp. 319-402, 1995.

[10] Cook WR and Rai S. "*Safe query objects: statically typed objects as remotely executable queries*," In Icse '05: proceedings of the 27th international conference on software engineering, pp. 97-106,2005.

[11] Cook WR and Rosenberger C. "*Native Queries for persistent objects, A design white paper*," db4objects inc.,2005.

[12] Cooper T and Wise M. "*Critique of orthogonal persistence*," In Iwooos '96: proceedings of the 5th international workshop on object orientation in operating systems (iwooos '96), pp. 122, 1996.

[13] Perez-Schofield JBG, Rosello EG, Soler FO and Cota MP. "*Visual Zero: A persistent and interactive object-oriented programming environment*," In Journal of Visual Languages & Computing, pp. 380 - 398, 2008.

[14] Liskov B, Adya A, Castro M, Ghemawat S, Gruber R, Maheshwari U, Myers AC, Day M and Shrira L. "*Safe and efficient sharing of persistent objects in Thor*," In SIGMOD Rec., pp. 318-329,1996.

[15] { http://www.iscap.ipp.pt/~rhp/aof4oop }

[16] Kiczales G, Hilsdale E, Hugunin J, Kersten M, Palm J and Griswold WG. "*An overview of AspectJ*," In Ecoop '01: proceedings of the 15th european conference on object-oriented programming, pp. 327-353, 2001.

[17] Paterson J, Edlich S, Hörning H and Hörning R , "*The definitive guide to db4o*," ,Apress, 2006.

[18] Soares S, Laureano E and Borba P. "*Implementing distribution and persistence aspects with aspectJ*," In SIGPLAN Not., pp. 174-190, 2002.

[19] Soares S, Borba P and Laureano E. "*Distribution and persistence as aspects*," In Softw. Pract. Exper., pp. 711-759, 2006.

[20] Rashid A and Chitchyan R. "*Persistence as an aspect*," In Aosd '03: proceedings of the 2nd international conference on aspect-oriented software development, pp. 120-129, 2003.

[21] Al-Mansari M and Hanenberg S. "*Path Expression Pointcuts: abstracting over non-local object relationships in aspect-oriented languages*," In Node/gsem, pp. 81-96, 2006.

[22] Popovici A, Alonso G and Gross T. "*Spontaneous container services*," In Ecoop 2003 -- object-oriented programming, pp. 499-551, 2003.

[23] Soares S and Borba P. "*Towards reusable and modular aspect-oriented concurrency control*," In Sac '07: proceedings of the 2007 acm symposium on applied computing, pp. 1293-1294, 2007.

[24] Kienzle J and Guerraoui R. "*AOP: Does it make sense? The case of concurrency and failures*," In Ecoop '02: proceedings of the 16th european conference on object-oriented programming, pp. 37-61, 2002.