# A Reflective Persistence Middleware over an Object Oriented Database Engine

Ortín Soler, F. Martínez Prieto, A. B. Álvarez Gutiérrez, D., and Cueva Lovelle, J. M.
{ortin, belen, darioa, cueva}@pinon.ccu.uniovi.es
Department of Computer Science, University of Oviedo
Calvo Sotelo 33007, Oviedo, SPAIN

**Abstract**

*Currently, using OODBMSs or persistence systems lacks flexibility due to the need of including additional code not related to the functionality of applications and learning of new APIs.*

*This paper explores the possibilities of an integral object-oriented system, based on an abstract machine with reflection. This permits the addition of the property of implicit persistence to the system, so that the user does not need to take special action to make objects persistent. A database engine is part of this persistence mechanism as an integral element of the system.*

*Computational reflection is defined as an inherent feature of the system. Due to this kind of reflection, both the database engine and the programming of applications are very flexible. Database engine properties can be dynamically modified in order to satisfy application requirements.*

*Reflectivity accounts for the design of a middleware that is able to achieve an implicit persistence system for the programmer in collaboration with the database engine. Transparent runtime selection of a level of persistence for the objects of an application without having to resort to additional code is now possible. The result is a very high-level object-oriented programming, very portable and with runtime flexibility.*

## 1. Introduction. Object-Oriented DBMS and Persistent Languages

### 1.1 Object-Oriented DBMS

Object-Oriented Database Management Systems (OODBMSs) were primarily motivated by new kinds of applications for which the Object-Oriented model was better suited for the semantics of the data to be stored. These systems have a common feature: using the OO model and integration with existing (persistent) languages [1]. Now "classic" examples of these systems are ObjectStore, $O_2$, GemStone, Poet, and Versant. Differences between systems were basically the election of the supported programming and query languages. This was also the cause of a limited portability. To alleviate this, the Object Database Management Group launched a standarization process (ODMG 1.0 and 2.0) [2, 3]. Nowadays, the majority of these systems claim to be ODMG compliant.

An evolution of relational systems towards extended relational systems is carried in parallel with these OODBMSs systems. These systems try to combine the OO model with the existing relational model. Changes in the storage and data managers of the relational DBMS are carried as needed. Examples of these systems are UniSQL and Persistence.

## 1.2 Persistent Programming Languages

Currently, and in co-existence with the above technologies, there is an increasing interest in incorporating support for persistent objects into (OO) programming languages.

Taking Java as an example, there are many different persistent Java flavours to choose. *PJava* (Persistent Java) [4] provides a persistent programming environment for the Java programming language based in a orthogonally persistent variant of the Java platform and machine.

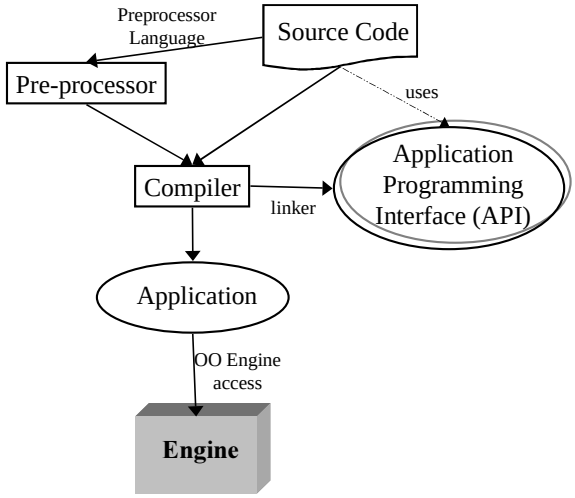


Figure 1 Overall structure for the inclusion of database functionality into a programming language

Other initiatives use persistent storage engines, such as PSE and PSE Pro [5] for Java (with a C++ version as well), that allow to store and retrieve objects en their native format. These persistent engines offer an API with varying learning curves and ease of use. These APIs endow the programming language with database functionality. Another example is Jeevan [6], a single user object oriented database for the Java platform, that provides a simple API of four classes and two interfaces. The Jeevan API provides for index specification and dynamic queries.

Other approaches translate Java objects into the relational model. Some rely on the programmer to translate into tables (StreamStore [7]). Others make the translation process transparent for the user (Java Blend [8]).

## 1.3 Adding Database Functionality to a Programming Language

Figure 1 shows the overall structure for the inclusion of database functionality into a programming language, which is commonly accomplished by means of an API, with a pre-processor used sometimes (i.e. extensions to the existing programming language).

A summary of the disadvantages of this approach follows:

1. User complexity. The user has to learn the introduced API and/or the extensions added to the programming language.
2. Legibility and maintainability suffers, as additional code, not related to the application logic ("intruder" code), has to be injected into the source code to have the added database functionality.

3. Portability suffers as well. There is a big dependence on the API used and its implementation.
4. Poor flexibility. Changes to database engine related aspects, such as adding a new indexing technique, are commonly made by changing and recompiling the source code.

### 1.4 Implicit Persistence and Incorporation of a Database Engine into a Reflective Integral Object Oriented System

We propose a different approach to the task of adding database functionality to programming languages, which is based on the notion of *implicit persistence*. This means that the user does not need to take special action to make objects persistent, no "intruder" code is needed, and so complexity, legibility and portability problems are not a concern. The system needs a persistence subsystem with a database engine to accomplish this.

To improve flexibility, a reflective architecture for this persistence subsystem is proposed, which will allow dynamic changes to persistence related aspects (for example, dynamic change of indexing techniques for a given application) without changing application code. This will also allow experienced users to directly use features of the database engine (explicit persistence). This is researched under the experimental system Oviedo3, which is an integral object-oriented system based on a reflective abstract machine. The persistence and database engine will be incorporated as an integral part of the system, which gives additional benefits.

The rest of the document is organized as follows. Section 2 justifies the developing of an Integral Object-Oriented System (IOOS), briefly describing the features of the system's object model and the abstract machine. Advantages of the structure of the system for the database engine that will be incorporated in the system is also described, as well as the different indexing mechanisms that will be considered. Section 3 is devoted to the reflection concept. A proposal for endowing an abstract machine with structural reflection is presented. In the next section persistence and design of the database engine for the IOOS based on the reflection capabilities of a reflective abstract machine is explored. Both explicit and implicit persistence is treated. Advantages of this system such as flexibility, uniformity, etc. are mentioned in section 5. Section 6 deals with related work in the field of reflective systems and OO databases and implicit persistence. Finally, some conclusions are drawn in section 7.

## 2. An Integral Object-Oriented System

The adoption of the object-oriented paradigm is not done in an integral way in all the system components. There are languages, databases, user interfaces, etc. using the object-oriented paradigm, which have to change to another paradigm to interact with other elements of the system like the operating system. They even can have different object models. This produces a serious impedance mismatch and interoperability problem, for the paradigm changes and/or object translations made depending on which element to work with. The result is a proliferation of additional software layers trying to alleviate these problems, but introducing in fact extra complexity in the system.

An approach in order to solve this problem is to move the OO support for the rest of the system to a common place into the operating system. Oviedo3 [9] is a research project that tries to build an experimental integral object-oriented system based on that foundation. All

components: user interfaces, applications, languages, compilers, databases... and the operating system itself share the same object-oriented paradigm.

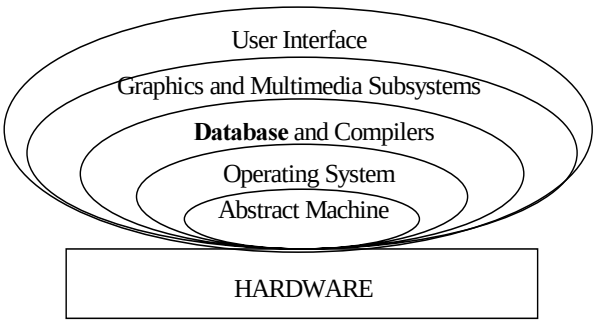


Figure 2 Components of the Oviedo3 System, shown in logical order of development

The system provides only one abstraction: objects. Objects can only create new objects from a class or send messages to others. One technique to structure an OO operating system aimed to support an integral OO system which offers many advantages is to use an OO abstract machine as the substrate of the OO operating system. This machine offers the basic object model and support to all objects of the rest of the system, and is given a reflective architecture for extra flexibility. The operating system functionality (as well as any subsystem functionality) is given by a set of user objects not different from any other object. Thus, an IOOS ideally provides a "world of objects" environment: a virtually infinite space where objects live indefinitely and exchange messages regardless of its location (Figure 3). The evolution of CORBA and Java is pointing towards this kind of heterogeneous distributed interoperable object environment.

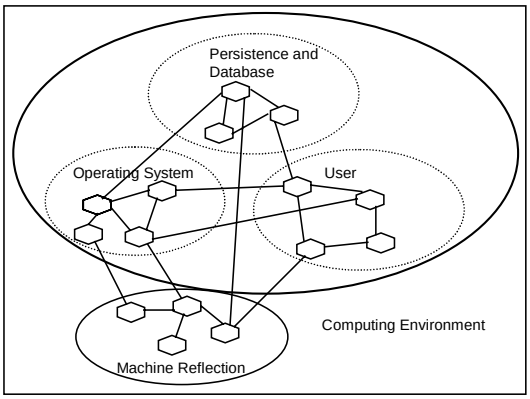


Figure 3 Computing environment composed of a set of homogeneous objects

## 2.1 System's Object Model and Abstract Machine

The machine ultimately provides the basic support for the common object model of the system, which intentionally follows the object model of the most popular OO methodologies. The goal is to take advantage of well-established OO concepts as well as carrying all the semantics of the object model (used in analysis and design phases) all the way to the implementation phase. The features the model includes are:

- Unique object identity (used by references);
- Encapsulation (access to an object only through methods);

- Classes (which are also used to derive types);
- Multiple inheritance (is-a) relationships;
- Aggregation (is-part-of) relationships;
- General association (related-to) relationships;
- Polymorphism and type checking including run-time checking;
- Exceptions;
- Protection, Concurrency, Distribution and Persistence (with database capabilities).

The machine language is a pure OO low level language. It allows class declaration, method definition and exception handling. Objects extending the functionality of the basic machine provide features not provided directly by the abstract machine.

## 2.2 The Database Engine of the Integral OO System

The OODB engine is not an individual element inside this integral system, like in conventional operating systems. It is an integral part of the computation environment. Database objects are objects just like other objets in the system (operating system or user ones), that provides database functionality: persistence, indexing, query processing, etc. Grouping them into a subsystem offers convenient access to them.

The existence of an object-oriented operating system and ultimately of an object-oriented abstract machine provides a set of benefits for the construction of the engine [10]:

- Seamless integration with the rest of the system. The OODBMS can be seen as playing the part of the file system in conventional operating systems, but with database capabilities. The database would not be used in isolation, but as a management system encompassing all objects in the system. For example, a user would find any object uniformly by querying the database.
- Facilitates its construction. The engine will on one side take advantage of the structural reflection of the machine, and on the other on some operating system capabilities (protection, concurrency, etc.), building on them incrementally by means of the object-orientation present in the system.
- Performance increase. It is not necessary to add new software layers to conventional operating systems to fill the semantic gap between operating system abstractions and objects. The integral system is already object-oriented.
- Productivity increase. Building database applications on this system is more productive, since it is not necessary for the programmer to change paradigms. Database and operating system share the same object-oriented paradigm, which is used uniformly in the system. For example, the same query language would be applied to database programming and user interface search tasks.

The engine is intended to be flexible and adaptative, using the extensibility offered by object-orientation. Work is being carried specially on the indexing mechanism, foundation for the query processing system. It will be used as a tested and an example of this flexibility.

## 2.3 Indexing Techniques

OO query languages have some special features to take into account: the existence of inheritance and aggregation hierarchies and the potential presence of method invocations. Thus, indexing mechanisms are needed that allow an efficient processing of queries under these circumstances. Many indexing techniques for OO models have been proposed, which can be classified into [11]:

a) *Structural.* These are based on object's attributes. Further divisions can be made: techniques that support for queries based on the inheritance hierarchy (Single Class(SC) and CH-Tree [12], H-Tree [13], etc.), techniques that support the aggregation hierarchy (Nested, Path y Multiindex [14], etc.), and techniques that support both aggregation and inheritance hierarchies (Nested Inherited [11], etc.).

b) *Behavioural.* These provide an efficient execution for queries that include method invocations. Method materialization [15] is one of these techniques.

## 2.4 Oviedo3 Indexing Mechanism

Depending on the most frequent type of query on a given class (or class hierarchy), some techniques are more efficient than others. However, most of the existing OODBMS use only a fixed subset of these indexing mechanism, and the user has neither the option of selecting her preferred indexing mechanism, nor the chance of adding new schemes.

The proposed database engine has the following basic features:

- Different indexing schemes. The system allows different indexing schemes. Initially, SC, CH-Trees and Path Index are considered in the first prototype that is being developed. However, the indexing mechanism allows for the easy addition of new schemes.

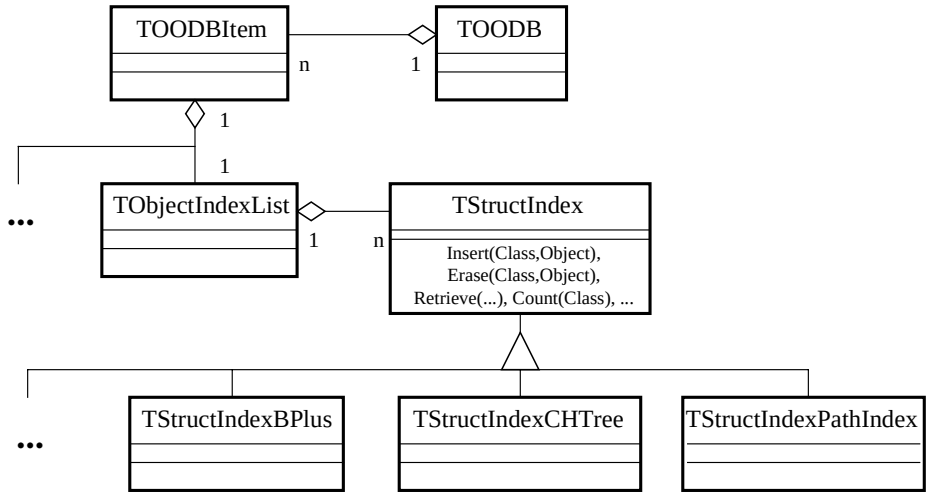


Figure 4: Indexing mechanism class diagram.

- Selection of the indexing mechanism. The system allows the selection of the indexing mechanism deemed as the most appropriate depending on the type of query made to the

class (or class hierarchy) in question. The inheritance and polymorphism allow easily those selections (with the classes TObjectIndexList and TStructIndex).

- Data type independence. It implies that the indexing can be performed over any data type (not simple types only). In order to accomplish this, the mechanism allows to use user-defined comparison operators.

The following code is an example of the method *NewIndex* in the *TobjectIndexList* (Fig. 4):

```
/ * Different indexes used in a Class
~
Persistent CLASS TObjectIndexList

AGGREGATION
  / * Index List
  Persistent IndexList:TList;
  / * Selected Index
  Persistent Selected:Integer;

METHODS
  / ...

/ * Add a new Index
NewIndex(anIndex:TStructIndex):Bool
  REFS
    bRes:Bool;
    Number:Integer;
  INSTANCES
    One:Integer(1);
    Zero:Integer(0);
```

```
CODE
~
~
  / * Add the new Index
~
  IndexList.Add(anIndex);
  / * Number=size(IndexList)
~
  IndexList.GetSize():Number;
~
  Number.Equal(One):bRes; / Is number equal to 0 ?
~
  Delete Number; / The object number is deleted
  JFD bRes,End; / Jump if false

  / * If it's the first one, it gets selected
  Selected.Set(Cero);

 End:
  Exit;
  ENDCODE

ENDCLASS
```

## 3. Reflection in the Integral Object-Oriented System

Reflection is the capability of a computational system to "reason about and act upon itself" [16] and adjust itself to changing conditions. The computational domain of a reflective system is the structure and the computations of the system itself. Two kinds of reflection can be observed: structural and computational reflection [17].

- Structural Reflection: is the most obvious and still the most developed form of reflection. It concerns the infinitary status of some data structures defined by reflexive domains [18]. The Java Reflection API [19] is an example of Structural Reflection.
- Computational or Behavioral Reflection: Is the ability for a process to describe, analyse and modify itself while running.

The Integral Object-Oriented System warrants portability by using the binary code of an abstract machine [20]. To build an IOOS on that machine, the design of the machine is very important. The first prototype of this OO abstract machine had inheritance, polymorphism, exception handling and multithreading [21]. Later on, properties were added, such as a distribution [22] and protection [23].

One option to augment the functionality of the machine is to add new instructions to the instruction set. A correspondent interpretation of these instructions has to be defined, to implement the new desired functionality. This implies a modification of the machine (interpreter).

A more flexible alternative uses the reflection concept, introducing structural reflection into the machine. The object is defined as the computation unit, and a set of primitives (basically method invocations) are defined upon objects. Objects have structural reflection, so its structure is always known. An object's properties can be freely modified, except properties defined as primitive.

This design allows to dynamically introduce any property that can be expressed in terms of primitives. Instead of modifying the machine (interpreter), additional functionality will be coded into binary code of the machine, implemented upon its structural reflection. This kind of design for abstract machines is also adopted by Smalltalk-80 [24], and especially by ObjVLisp [25].

### 3.1 Computational Reflection

However, most of the power of the system is due to computational reflection. Smith [26] proposes an interpreter tower to obtain a system with this property. We will define just two levels on this tower:

- The execution of an interpreter for a language **L**, expressed by binary code of the virtual machine **B.**
- The execution of a user program expressed in the language **L** by the interpreter.

To make the system expressed in the language **L** computationally reflective, two things are needed:

1. A "jump" [1] from the computation of **L** to the computation of **B** in the two-level tower must be possible.
2. Computation of **B** has to be structurally reflective in order to modify the computation state of **L**.

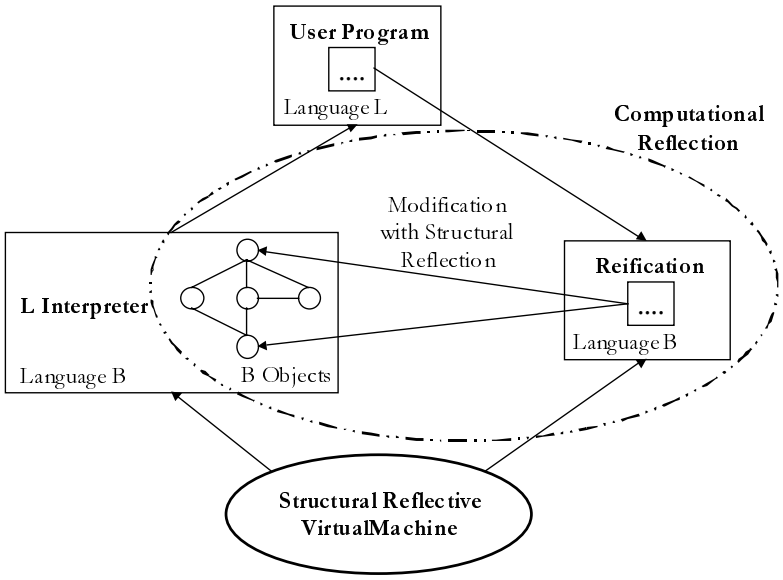


Figure 5 Computational reflection on an abstract machine with structural reflection

[1] This kind of "jump" is called reification [Maes87].

There are some techniques to build reflective systems, such as MOPs [27]. A MOP (Meta Object Protocol) is the interface exposed by some auxiliary objects (metaobjects) that offers the option to change existing objects. This interface is defined at compile-time, losing some flexibility for the sake of efficiency. Others like MetaXava are more flexible, instantiating this protocol at runtime, but less efficient. A higher degree of flexibility[2] is achieved by the before mentioned "jump" between different computation levels.

The rest of the paper will just use "reflection" for "computational reflection".

## 4. Achieving Persistence by means of Reflection

As mentioned in a previous section, the persistence system of the integral object-oriented system relies on a database engine. Only the interface is specified for better flexibility, leaving the actual implementation unspecified, as in other architectures such as CORBA [28]. This design opens the door for *open implementations* [29]. For example, many different implementations could be selected depending on a set of parameters, as in the case of the indexing mechanisms.

Portability is another feature of this kind of architecture. As in the Java platform [30], the binary code of the abstract machine is portable to different hardware platforms. With regard to portability of the application code, access to objects not directly part of the application has to be studied. OS objects and specially database engine objects are in this group. When migrating the application to a different hardware platform and version of the abstract machine, the database engine could have a different implementation, taking advantage of special features of the hardware system. In this former case, portability is also assured by the common specification of the interface of the engine.

The implementation of the engine will use the structural reflection offered by the machine. This eases the implementation of the engine, as all objects' properties are accessible at runtime. Creation, access, modification and deletion of objects and its properties are services provided by the machine itself by means of reflection.

Once the engine's interface is specified and implemented, applications can use it as an actual persistence system.

### 4.1 Applications Using Explicit Persistence

Experienced users which code applications willing to use the persistence property of the integral system can do so by accessing directly the services of the database engine through its interface. A reference to the engine has to be acquired before the application can use it, as the engine will be an object of the integral system [9].

In this *explicit persistence* the programmer decides when a specific service of the engine is to be used. This is usually more efficient than implicit persistence (next section), but the programmer is concerned with the burden of managing in a correct way all the persistent objects of the application.

---

[2] Flexibility usually hurts efficiency. In the first prototype of the system we will concentrate on flexibility, leaving optimizations for future versions.

As the engine is an integral part of the (operating) system, and being accessible from any part of the system[3], access to databases is from the programming language itself, as in the case of PSE Pro, Jeevan, etc.

### 4.2 Applications Using Implicit Persistence

With *implicit persistence*, applications do not have to take special actions to make objects persist. The system makes objects persistent transparently, without having to include additional "intruder" code besides the proper application logic. That is, there is no need to explicitly specify calls to the database engine, as in the case of explicit persistence [31]. The programmer does not have to take the responsibility of linking the application with the persistence system; the system will transparently do so. With this kind of persistence, an application could be created, debugged and tested, and later, at execution time, certain objects could be made persistent dynamically, even with different levels of persistence (with encryption, replication, logging of updates, different kinds of stable storage, etc.)

The language for applications (section 3) is interpreted by an interpreter program developed on the reflective abstract machine, being then a computationally reflective language. The middleware that accomplishes this transparent persistence for any object will be built taking advantage of this reflective property to implement different levels of persistence.

This middleware software will dynamically make computational modifications to the objects implementing an application, so that these objects will now automatically make the appropriate calls to the database engine (figure 6). The programmer will just have to select the desired persistence level for her applications objects (or use default levels selected by a system policy). An example of implementation of this middleware would modify the method invocation system, updating the object anytime its attributes change their values.

Updating of objects could be done using different persistence levels at various times, for example:

- Creation and deletion of objects
- Invocation of a given set of methods
- Update of the state of an object
- At regular intervals of time

[3] As a note, the abstract machine and the Integral System use distributed references. So, remote objects can be accessed just like local ones, and in particular remote persistence systems can be accessed.

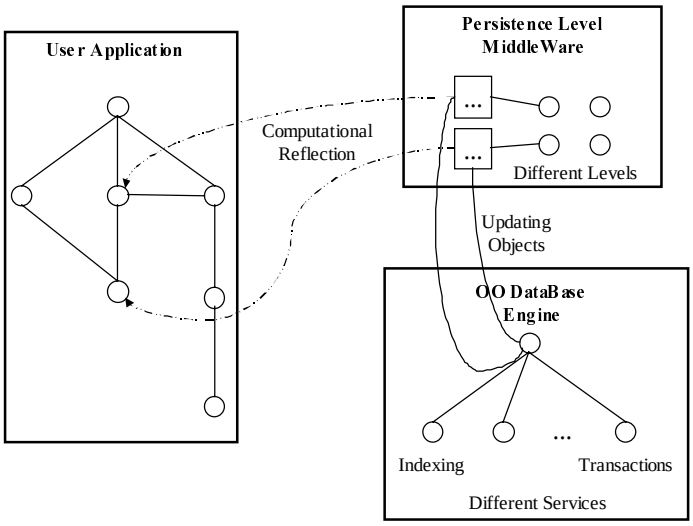


Figure 6 Overall structure of an implicit persistence reflective system

If there is a need to dynamically change the behaviour of persistence for the objects of a running application, for example, updating changes upon any changes to the state of an object, the system could proceed as follows. The message passing mechanism of user objects will be modified, making it now call the database engine to make the object persist when the invocation of a method results in changes to the state of the object. There are no changes to user code. User code is also not conscious that its functionality has been changed, this is completely transparent to it.

Obviously, the combination of the persistence level with activations of persistent updates will be less efficient when the number of updates increases. However, this transparent middleware takes work and responsibility out from the programmer, making applications easier to debug as well.

## 5. Advantages of this Architecture

The most important feature of this persistence subsystem is the higher degree of flexibility allowed by reflection. Some advantages derived from this property are mentioned below.

### 5.1 Uniform use of the Persistence System

The database engine is an integral part of the system. The engine services can be used either directly or transparently from any other part of the system: from the OS shell to user applications.

These services are also homogeneously used throughout the system. There is no need to make distinctions between data files, databases, executable files, etc. Persistent objects are the only abstraction of the system.

### 5.2 Flexible use

For applications using implicit persistence there are two different variables to take into account at runtime: the persistence level desired for objects and the different existing mechanisms or implementations for the services of the engine. A flexible and dynamic selection of this parameters is possible with this computational reflective system, so the balance between persistence functionality and updating frequency and efficiency can be tuned as needed, adjusting them along time.

For example, an application could use different indexing mechanisms for queries depending on the system load; even persistence could be disabled when faster execution is required, being re-enabled later.

Flexibility needs control: arbitrary changes to a running system by any user must not be allowed. The existence of a protection mechanism that can be used homogeneously to control message passing for the whole system [23] could be used in particular to control which reflective calls are allowed.

### 5.3 Parameter tuning

To find the optimum performance level of a computer system with limited resources is a complex task. This is applicable to the field of databases, trying to access higher data volume in the shortest time possible.

One particular case is related to search algorithms in databases that use different data structures. Some strategies perform better in some contexts but are less efficient in others. For example, nested index has the best retrieval performance, however multiindex has the best update performance [14].

A study of the impact of the variables that affect the performance of the engine has to be carried in order to select a given strategy for a context. The flexibility of the system facilitates these studies. The most important variables are:

- Classification of an application's objects.
- Existing persistence levels
- Different implementations for the engine's services.

Reflectivity makes very simple for an application to modify these variables and generate statistics in order to make a compromise. The system is a very suitable platform for the benchmarking of different indexing mechanisms, for example. Guidelines for the dynamic selection of indexing mechanisms depending on specific contexts will be produced after analyzing this data.

### 5.4 Higher programming level

Applications programming is simpler now with persistence implemented reflectively, as programmers do not have to deal with persistence[4] (being it implicit). At runtime, the user will just need (if desired) to identify which objects should persist and the persistence level desired. Moreover this decisions can be changed later. The abstraction level is consequently raised.

Another advantage of this raise of abstraction level achieved by using an integral database engine with a related reflective middleware is a big improvement when debugging, porting and maintaining applications.

## 6. Related work

There is some related work in the field of reflective programming languages. Most of these languages are based in the Meta Object Protocol, being 3-KRS [16] a pioneer. Some current examples are OpenC++ [32] and MetaXava [33].

These languages give *a priori* reflective executions by generating compile-time additional code or by extending the instruction set of a virtual machine. But a running application can not be modified unless source code is recompiled. These MOPs generate more efficient code but losing the flexibility of modifying running applications present in our system.

There are also reflective object-oriented operating systems such as Apertos [34]. Every object is associated with an execution environment called metaspace and composed of

[4] Although not treated here, similar considerations can be made about distribution and protection in the integral system.

metaobjects. Communications with the metaspace is possible through a set of metaobjects with these specific functions.

There are many OODMBSs, in the form of independent servers (O2, ObjectStore, etc.) or in the form of complements for programming languages (PSE, PSE Pro, etc.). However, these systems usually work like black boxes with poor flexibility. There are no provisions for the dynamic addition of new indexing mechanisms or for applications to select a given indexing mechanism, etc. On the other hand, these systems are conceived as independent elements, not as an integral part of a global system like the one proposed in this paper.

OODMBSs and persistent systems in general have three different possibilities to determine which objects are to persist: *By type* (an object may be made persistent when it is created, based on its type, persistent types versus transient types, as in Objectivity/DB), *by explicit call* (the user may explicitly specify persistence of an object, as in ObjectStore), and *by reference* (determine persistence of objects by reachability from certain globally known persistent root objects, as in GemStone). Nevertheless, a less explored alternative is the utilization of implicit persistence by means of a reflective mechanism in the sense in which it is proposed in this paper. That is, systems that allow the dynamic change at runtime of the level of persistence of an object, which the consequent increase in simplicity for the programmer.

To explore the concept of implicit persistence, a first prototype used a design in which persistence was implemented by adding functionality to the base code of the abstract machine [31]. Although applications programming in fact took advantage of this transparent persistence, this designed lacked flexibility in the way described before. Implementing implicit persistence with the reflective design proposed in this paper retains the benefits of implicit persistence while giving more flexibility.

## 7. Conclusions

Currently, programming an object-oriented application using OODBMSs implies the knowledgements of different APIs and the inclusion of additional code. An alternative approach uses the concept of implicit persistence, in which the user does not need to take special action to make objects persistent, no "intruder" code is needed, and so complexity, legibility and portability problems are not a concern.

An integral object-oriented system based on an abstract machine that provides basic support for objects in the system is used to implement implicit persistence. The seamless integration of a flexible database engine in this system taking the part of conventional file systems is a first step towards this, with additional benefits.

The use of an abstract machine with structural reflection upon which languages with computational reflection are built is the fundamental piece to achieve flexibility in the database engine and the persistence mechanism.

The flexibility in the database engine and the specification of its interface builds on the object-orientation and reflection capabilities of the machine. Experienced users can now use explicit persistence by calling directly services of the engine.

Implicit persistence is achieved by a reflective middleware that makes runtime computational changes to the application objects, doing (transparent) calls to the database engine to make objects persist. This accounts for flexibility, as no additional application code is needed,

changes can be dynamically made while the application is running and different levels of persistence can be selected for objects.

The system then has a number of benefits related to persistence such as a uniform use of the persistence system throughout the system. Flexible use of persistence by dynamic selection of parameters related to persistence is another benefit. This also makes the system a very good platform for benchmarking and/or parameter tuning.

The overall result is a very high-level object-oriented programming, very portable and with runtime flexibility.

## 8. References


[1] R. Cattell . *Object Data Management. Object Oriented and Extended Relational Database Systems* ( Revised Edition ). Addison Wesley, 1994.

[2] R. Cattell, T. Atwood, J. Duhl et. al. *The Object Database Standard:ODMG-93*. Morgan Kaufmann, 1994.

[3] R. Cattell, D. Barry, D. Bartels. *The Object Database Standard: ODMG 2.0.* Morgan Kaufmann, 1997.

[4] M. Atkinson, L. Daynès, M. Jordan, T. Printezis, S. Spence. *An Orthogonally Persistent Java. SIGMOD Record, Vol 25 Nº4. December,1996*

[5] Steven T. Abell. *Using Java with PSE*. Java White Paper. http://www.odi.com, March 1999.

[6] *Jeevan User's Guide*. http://www.w3apps.com/, March 1999.

[7] *StreamStore User's Guide*. http://www.bluestream.com/ss/default.htm, March 1999.

[8] *Java Blend: Integrating Java Objects with Enterprise Data*. White paper. http://java.sun.com, March 1999.

[9] Darío Álvarez Gutiérrez. *An object-oriented abstract machine as the substrate for an object-oriented operating system.* 11th European Conference on Object-Oriented Programming (ECOOP'97). Jyväskylä (Finland). June 1997.

[10] A.B. Martínez, D. Álvarez, J.M. Cueva, F. Ortín , J.A. Pérez. *Incorporating an Object-*Oriented *DBMS into an Integral Object-Oriented System.* World Multiconference on Systemics, Cybernetics and Informatics and International Conference on Information Systems, Florida, 1998.

[11] E. Bertino and P. Foscoli. *Index Organizations for Object-Oriented Database Systems*. IEEE Transactions on Knowledge and Data Engineering. Vol.7, 1995.

[12] W. Kim, K.C. Kim, A. Dale. *Indexing Techniques for Object-Oriented Databases.* En W. Kim y F.H. Lochovsky (ed) : Object-Oriented Concepts, Databases, and Applications. Addison-Wesley, 1989.

[13] C. Chin, B. Chin, H. Lu. *H-trees: A Dinamic Associative Search Index for OODB.* ACM SIGMOD, 1992.

[14] E. Bertino and W. Kim. *Indexing Techniques for Queries on Nested Objects*. IEEE Transactions on Knowledge and Data Engineering. Vol.1 nº2, 1989.

[15] A. Kemper, C. Kilger and G. Moerkotte. *Function Materialization in Object Bases: Design, Realization, and Evaluation.* IEEE Transactions on Knowledge and Data Engineering, 1994.

[16] Pattie Maes. *Computational Reflection*. Technical Report 87_2, Artificial Intelligence Laboratory, Vrieje Universiteit Brussel, 1987.

[17] Jacques Ferber. *Computational Reflection in class based Object-Oriented Languages.* Proceedings of the Conference on Object-Oriented Programmings, Systems, Languages, and Applications, OOPSLA'89, New Orleans, Oct. 1989, pp. 317-326.
[18] Jacques Ferber. *Coceptual reflection and actor languages*. Meta-Level Architectures and Reflection. P. Maes, D. Nardi (Editors). North-Holland, 1988.
[19] *Java Core Reflection. API and Specification.* JavaSoft. January 1997.
[20] D. Álvarez, *Complete Persistence for an Object-Oriented Operating System using an Abstract Machine with Reflective Architecture*, Ph. D. Thesis, University of Oviedo, Spain, March 1998.
[21] J.M. Cueva. *The Integral Object Oriented System Oviedo3*. II Jornadas sobre Tecnologías Orientadas a Objetos. Oviedo, 1996.
[22] F. Álvarez, L. Tajes, M. Díaz, D. Álvarez and J.M. Cueva**.** *Agra: The Object Distribution Subsystem of the SO4 Object-Oriented Operating System*. Proceedings of the PDPTA'98. International Conference on Parallel and Distributed Processing Techniques and Applications, pp. 255-258. CSREA Press. 1998.
[23] M.A. Díaz, D. Álvarez, A. García-Mendoza , F. Álvarez, L. Tajes and J.M. Cueva**.** *Merging Capabilities with the Object Model of an Object-Oriented Abstract Machine.* Proceedings of the ECOOP'98 Workshop on Distributed Object Security and the 4th Workshop on Mobile Object Systems, pp. 9-13. Inria Rhône-Alpes, Francia, Julio de 1998.
[24] Glenn Krasner. *Smalltalk-80, bits of history, words of advise.* Xerox Palo Alto Research Center. Addison Wesley 1984.
[25] Pierre Cointe. *The OvjVlisp Kernel: a Reflective Lisp Architecture to define a Uniform Object-Oriented System.* Meta-Level Architectures and Reflection. P. Maes, D. Nardi (Editors). North-Holland, 1998.
[26] B.C. Smith, *Reflection and Semantics in a Procedural Language*, MIT-LCS-TR-272, MIT, Cambridge, 1982.
[27] Gregor Kizcales, J. des Rivieres, D.G. Bobrow.*The Art of Meta-Object Protocol.* MIT Press 91.
[28] OMG. *CORBA: Architecture and Specification.* 1997.
[29] Chris Maeda, Arthur Lee, Gail Murphy, Gregor Kiczales. *Open Implementation and Design*. Proceedings Symposium on Software Reuse. May 1997.
[30] Douglas Kramer. *The Java™ Platform. A White Paper*. Sun JavaSoft. May 1996.
[31] F. Ortín, D. Álvarez, R. Izquierdo, A.B. Martínez, J.M. Cueva. *The Oviedo3 Persistence System*. III Jornadas de Tecnologías de Objetos. Sevilla, 1997.(in spanish).
[32] Shigeru Chiba. *A Metaobject Protocol for C++*. Proceedings of the Conference on Object-Oriented Programmings, Systems, Languages, and Applications, OOPSLA'95. pp. 285-299.
[33] Jügen Kleinöder, Michael Golm. *MetaJava: An Efficient Run-Time Meta Architecture for Java™.* TR-14-96-03. Computer Science Department, Friedrich-Alexander-University Erlangen-Nürnberg, Germany. June 1996.
[34] Y. Yokote. Kernel Structuring for Object-Oriented Operating Systems: The Apertos Approach. Workshop on Reflection and Meta-level Architectures at OOPSLA 93.