# A Middleware Framework for the Persistence and Querying of Java Objects

Mourad Alia[1,2], Sébastien Chassande-Barrioz[1], Pascal Déchamboux[1], Catherine Hamon[1], Alexandre Lefebvre[1]

[1]France Télécom R&D, DTL/ASR, 28 chemin du Vieux Chêne, B.P. 98, 38243 Meylan CEDEX, France
[2]LSR, B.P. 72, 38402 Saint-Martin-d'Hères CEDEX, France
{alia.mourad, sebastien.chassandebarrioz, pascal.dechamboux, catherine.hamon, alexandre.lefebvre}@rd.francetelecom.com

**Abstract.** This paper presents the adaptable and flexible architecture of a middleware framework for the persistence and querying of Java objects. The framework is composed of two sub-frameworks, each responsible for one aspect: persistence and queries. The persistence framework considers two kinds of objects: Memory instances (MI), which represent Java objects holding the data to be made persistent, and Data Store instances (DSI), which represent data items stored within data stores. It thus concentrates on the binding chain between a DSI and an MI, providing the management of the structural projection of persistent objects to a particular data store when performing I/Os. The query framework makes it possible to express, optimize and evaluate queries over heterogeneous data stores and in particular over the persistence framework objects. Query expression is independent of any query language and can be mapped to several standards. The middleware presented in this paper has been integrated in several contexts, thus validating its adaptability and flexibility.

## 1 Introduction

The problem of object persistence has been the subject of much research and industrial work over the last few years. The literature distinguishes two degrees of object persistence: *transparency* and *orthogonality*. Transparent persistence [2] makes a minor distinction between transient and persistent objects. This means that the programmer has some degree of persistence control, such as opening/closing transaction boundaries. *Orthogonal* persistence [1] [2] [3] supposes that the persistence property is independent of the object type: the programmer does not specify the objects that will be persistent and all objects are potentially persistent, as in the object model of ODMG [6]. Further details about approaches for adding persistence are detailed in [17] in the case of the Java programming language.

Implementing orthogonal persistence has been identified as a difficult task, even though it has been done. Transparent persistence is implemented and used primarily in the industry. This trend is strongly driven by the fact that most enterprise data are stored in relational databases. The use of complex underlying object-to-relational mapping techniques leads to a layered application server design. In this context, transparent persistence is more appropriate. Several standards for transparent persistence have been developed, such as CORBA Persistent State Service (COS PSS) [7], Java Data Objects (JDO) [32] and EJB-CMP Entity Beans [31], for accessing persistent objects.

We address the problem of building persistence solutions for applications which manipulate persistent objects, whether these applications follow the transparent persistence or the orthogonal persistence approach. Programming such applications implies managing *storage instances* (e.g. database tuples) and *memory instances* (i.e. real-world Java objects). Programmers are then faced with managing both kinds of instances, that is, organizing transfers between the associated memory levels, accommodating formats and types which usually differ between these levels, and also translating references.

Storage instances may be stored in databases, file systems, ERP systems or mainframe transaction processing systems. These data stores are all referred to as Data Stores (DS) which can also support transactions and can potentially be federated (objects within one DS can refer to objects within another DS). The object persistence standards, such as those cited above, are referred to as Memory Instance Managers (MIM). Applications which manage persistent objects are considered to lie on an MIM. Another common requirement is that of querying the persistent objects, that is, providing associative access to these objects (usually by expressing a query), as opposed to direct access starting from a reference.

In this paper, we propose a middleware solution for tackling this problem independently of existing standards (MIMs) and persistent approaches, with the aim of ensuring that such a framework nevertheless be usable in the context of these standards and approaches. Thus, the middleware must be *adaptable* and *flexible* in order to allow pluggable implementations of DSs into MIMs: from the DS point of view, it must be possible to extend the middleware to incorporate different types of DSs (downward adaptability); from the MIM point of view, the interface exposed by the middleware must allow the implementation of different MIMs persistence approaches (upward adaptability). In particular, the proposed middleware framework makes it possible to implement both transparent persistence and orthogonal persistence.

These properties are transposed into the following goals for the middleware presented in this paper:

- Independence from the data store type (RDBMS, OODBMS, directories, flat files, etc).
- Independence from the memory object life cycle.
- Openness to other non-functional aspects, such as concurrency control, caching support and consistency control.
- Openness to DS federation and distribution.
- Independence from the query language.

We adopt the framework approach for designing the architecture of the middleware. A framework is defined as *"a reusable conception of one or part of one system that is represented by a set of abstract classes and the way their instances interact"* [16]. In order to be usable in the context of standards, the framework must be extended into so-called "personalities". In our case, the framework approach results in the definition of APIs for the persistence of objects in a DS, the retrieval of objects from a DS, interactions with the MIM layer, or the expression and evaluation of queries. Typical personalities include the above-cited standards (CORBA PSS, EJB CMP and JDO). An expected benefit of the framework approach is that of software reusability: the persistence and query frameworks can be re-used to implement different standards, as we have done with EJB [21] and JDO [24], thus reducing redundant code production.

This article presents the architecture of the persistence and query middleware framework. The architectural concepts are in the line of the ISO Reference Model of Open Distributed Processing [13] [14]. The paper is organized as follows. Section 2 positions our work with regard to related work. The overall architecture and principles are presented in Section 3. Sections 4 and 5 detail the persistence and query frameworks, respectively. Section 6 presents the implementation and usage of the framework, validating our approach. Section 7 concludes and presents future work.

## 2 Related Works

This work can be positioned with regard to many other works in two research domains: object persistence and data integration systems.

### 2.1 Object Persistence

Object persistence has been the focus of much work representing different approaches and viewpoints. The Pjava project [4] proposes *orthogonal* Java object persistence without changing the Java syntax. Data, metadata (classes) and code (methods) are made persistent by modifying the Java virtual machine.

Other approaches involve the definition of standard interfaces for the *transparent* persistence of objects into data stores. The CORBA Persistent State Service [7] interposes a CORBA-based abstraction layer between a server and its persistent storage. The persistent information is represented as *objects* stored in *storage homes*. Conceptually, a data store is a set of typed storage home objects. PSS users have to define objects and storage homes either using PSDL (Persistent State Description Language), which describes the persistent data, or directly in a programmatic way. In the case of PSDL, a compiler generates code in a target programming language, and in particular Java. JDO, Java Data Objects [32], is another example of transparent persistence for Java objects by *reachability*[1]. It consists of a simple set of interfaces

[1] Persistence by reachability is defined as follows: a given object that can be reached by following references starting from a persistent object becomes itself persistent.

which enable persistence capability for applicative objects. JDO instances are accessible via a Persistence Manager which represents a session with a data store. A simple method *makePersistent(obj)* of a Persistence Manager allows object *obj* to become persistent. JDO also provides a simple query language, JDOQL, for selecting instances of persistent objects. In the J2EE platform [31], Container-Managed Persistence (CMP) is responsible for handling the persistence of the entity bean states (their fields) and their interrogation at run-time. In the entity beans deployment descriptor, the user describes the abstract schema[2] which defines the beans persistent fields and relationships. The associated query language, EJB QL, is a SQL92-like query language with navigational expressions over the abstract schema. All these standards impose a way of managing the object's life cycle and are considered as Memory Instance Managers (see Section 3).

As a comparison, our framework is a lower-level middleware interface allowing the implementation of all these standards.

In the same spirit as the work presented here, many products and tools support the mapping of Java objects to persistent storage, and especially to relational database systems. Amongst the most representative of them are TopLink [19], originally from the Object People, PowerTier [27] from Persistence Software and the open source project OJB [26]. They are usually provided as autonomous systems and cover a wide range of functions (e.g. cache management or concurrency policies). However, such systems follow the black-box principle, hiding most of their internals. As a result, extending such systems for handling new functionalities such as distribution (distributed references between persistent objects) or handling new data store models is usually very difficult. These limitations motivate the downward adaptability and the flexibility of our framework.

The PerDiS project [9] treats the problem of distributed and shared persistent objects for distributed collaborative engineering applications with shared memory, caching and security management. Objects are inter-referenced by pointers and are nested into clusters within memories. Unlike the framework presented here, PerDiS imposes persistence by reachability and a particular object life cycle: objects reachable from persistent roots are made persistent and others are automatically garbage-collected. For a transactional access to data stores, applications use the SDAI standard interface (Standard Data Access Interface) [15], which is at the same level as JDBC or ODBC gateways.

### 2.2 Data Integration Systems

The goal of the data integration systems is to provide uniform access (querying) over heterogeneous data sources. Most of these systems [10] [12] [28] follow the mediators/wrappers architecture [33]. The mediator interacts with several exported data

---

[2] An abstract schema is a virtual schema which is independent of the physical data store schema.

store schema. It uses wrappers to interact with related data stores. There exist simple relational wrappers (such as JDBC or ODBC), but also object wrappers that hold and manage schemas (fat wrappers).

Compared with this architecture, our framework is more generic and can be used to generate wrappers for object mediation systems, as in [29]. The query framework can then be used by mediators to retrieve data from multiple data sources, accessed through the generated wrappers.

As the middleware needs to process queries, we reuse the results of research in query processing (query optimization and evaluation), whether in a centralized environment or in a distributed environment over multiple data sources [10] [11] [28].

## 3 Model and Principles

Our approach is to provide a middleware layer by "opening the black box" and following the separation of concerns principle. The middleware is composed of two sub-frameworks: the persistence framework and the query framework. It is these frameworks which make the middleware flexible and adaptable. In practice, it is also possible to use these two frameworks separately.

The persistence framework focuses on the I/O between the data store and the memory objects by identifying the persistent items using persistent object identifiers. This is achieved by interposing mediation objects, or *binding objects*[3], which represent Data Store Instances (DSI).

In addition to accesses through identifiers, the query framework is responsible for associative access. It thus manipulates collections of persistent objects and makes it possible to express and process MIM queries in their related query languages.

Figure 1 illustrates the interactions between the middleware layer and both the data stores and applicative layers. Persistent data of memory objects are projected into binding objects managed by the middleware, which are further mapped to data within data stores according to meta information. Persistent data of memory objects are typically attributes (fields) of a class. Objects managed by the persistence framework, typically binding objects, follow an object model presented in Section 4.2. This makes it possible to adapt the persistence object model to the MIM object model.

The interaction between binding objects and memory objects is achieved through objects called Memory Instances (MI), which hold the persistent fields. In order to access data stores, gateways such as JDBC or ODBC are used.

In the following, we present more explicit definitions of the general concepts related to each layer which will guide us in the presentation of the framework.

---

[3] A *binding object* is a computational object which holds a binding between other computational objects. Binding objects are subject to special provisions (RM ODP) [14].

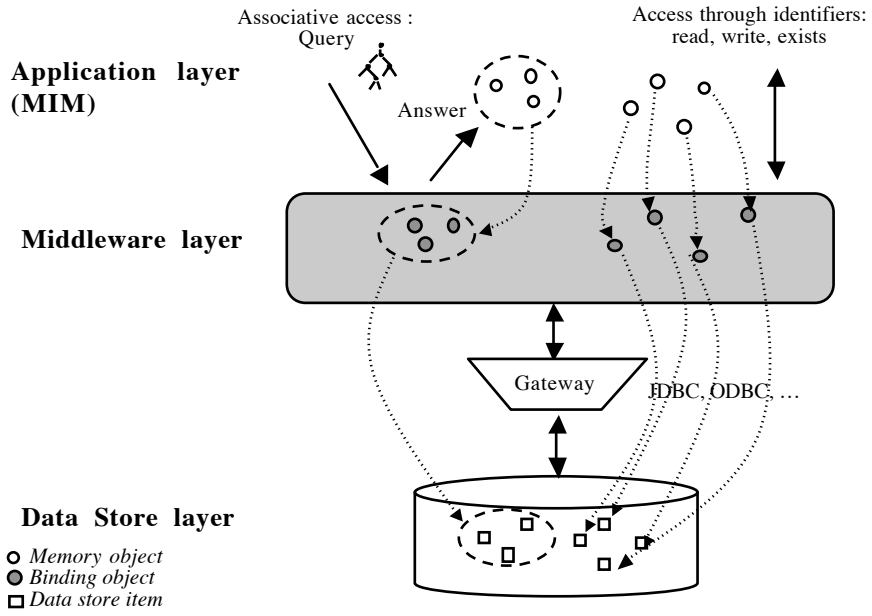


**Figure 1:** The persistence and query middleware – the "big picture".

**Data Store (DS).** A DS provides the infrastructure for storing persistent information. A DS displays an interface which allows a client to manipulate its persistent data locally or remotely. Examples of DS include file systems, relational database systems, object database systems or directories.

**Data Store Instance (DSI).** A DSI is a data item stored within a DS. Such an item is identified by a persistent name within the middleware (see Section 4.3). A data item could be, e.g. a row of a relational database table, an object within an object database class or a file.

**Memory Instance (MI).** An MI is an object which holds the variables of the memory object to be made persistent. It can cooperate with a binding object in order to load/store its variables from/to the DS. From the persistence framework point of view, the MI is an Accessor (see Section 4.1). It could, for example, be the *PersistenceCapable* object in a JDO implementation of an MIM.

**Memory Instance Manager (MIM).** An MIM is the software (application) layer which manages the memory instances corresponding to the data store instances. This layer usually provides a high level of transparency with respect to the management of this projection (e.g. it hides load/store actions). It also defines the life cycle of an MI. Examples of MIMs include JDO, Corba PSS or CMP EJB implementations.

The separation between the MIs and the binding objects offers a freedom of choice for implementing various object life cycles, and allows several implementations. This is the key point which differentiates our approach from the related work mentioned in Section 2. In order to use the persistence framework, the user simply implements the MI and links it to the binding object managed by the framework, as explained in the next section.

# 4 Persistence Framework

## 4.1 Bindings and Accessors for Storage Synchronization

The basic architecture principle that governs the persistence framework consists in interposing *binding objects* between an MI and its associated DSI. They provide typed synchronizations (i.e. I/Os) between the MI and the DSI. There are two main synchronization actions: *read* and *write*. They are typed, as they support a particular structure for each persistent object class to be stored, as well as the way to map this structure to the associated DSI. Thus, a binding object is the Java object where the mapping occurs when performing I/O operations.

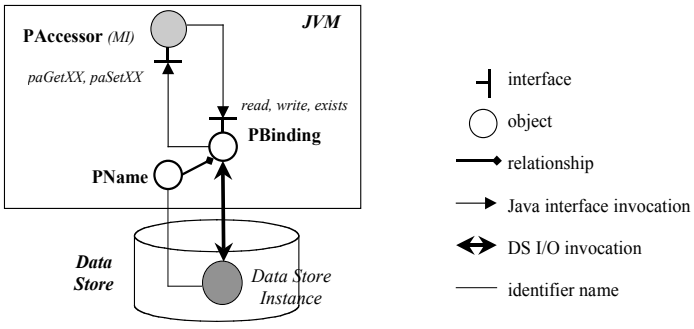


**Figure 2:** Binding mediation for storing Java persistent objects.

Figure 2 illustrates the persistence chain between the DSI and the MI through three interfaces: *PBinding*, *PAccessor* and *PName*. Before being able to perform synchronizations, a binding object, represented by *PBinding* interface, must be assigned a persistent identifier, represented by the *PName* interface (see Section 4.3). This identifier is a Java object which designates the DSI to which the binding object

is bound. Symmetrically, the binding object must also be assigned an accessor object, represented by the *PAccessor* interface, in order to have access to the state variables (or fields) of the MI to which it is bound. Once both assignments are done, the persistence chain is fully functional.

To make an object persistent, the *write* method is called on the *PBinding* object; the *PAccessor* object is used to read from the memory the values to be stored into the data store, by calling *paGetXXX* methods. Inversely, to read a persistent object from the data store, the *read* method is called on the *PBinding* object; the *PAccessor* object is used to write into memory the values obtained from the data store, by calling *paSetXXX* methods.

The user of the persistence framework is responsible for providing the implementation of the *PAccessor* interface, since the management of objects in memory is outside the scope of the persistence framework itself. The *PAccessor* interface is composed of field-specific setter and getter methods (i.e. *paSetXXX* and *paGetXXX* for the field named *XXX*). The *paGetXXX* method returns the value of field named *xxx*. The translation table between the middleware object model types and their Java counterparts determines the type of this value. For example, it can return a Java *int* value. The *paSetXXX* method performs the symmetrical action by assigning a truly typed value to the *xxx* field.

**Analysis.** These architecture principles satisfy the upward (MIM) and downward (DS) adaptability requirements:

1. The persistence chain is open to different MIM strategies. Synchronization points may occur when demarcating transaction boundaries, which can be done at the MIM level or in upper layers: the MIM layer decides when to call the read and write methods, and in which transactional contexts (the *read* and *write* methods of the *PBinding* offer an argument for propagating the connection). Thus the framework is independent of the transactional behaviour.
2. The binding objects support a unique *PBinding* interface, independent of the type of data store. Thus, this abstract interface provides adaptability and portability. While supporting this interface, a binding object hides the means to access the DS. For relational databases, it can use SQL statements, submitted through JDBC, in order to read/write Java persistent fields.

Moreover, the use of the object interposition approach does not require modifications to the Java Virtual Machine in order to support the persistence of objects.

In terms of implementation, the framework does not impose a way of composing the *PBinding*, *PAccessor* and memory objects. The only constraint is that the *PBinding* object must have a link to an object implementing the *PAccessor* interface. As a result, the user can freely compose the objects. By way of example, there is a single instance if the memory instance class directly implements *PAccessor* and the *PBinding* class extends this class. Thus, the framework offers great flexibility in terms of memory object architecture, as illustrated in Section 6.

**Example.** This example shows the binding and accessor objects in the case of a simple *Product* class with persistent fields *name* and *price*. Class *ProductAccessor* implements the getter and setter methods. Class *ProductBinding* implements the *read* and *write* methods.

```
public interface ProductAccessor extends PAccessor {

  //Accessors to the name field
  public void paSetName(String val) throws PException;
  public String paGetName() throws PException;

  //Accessors to the price field
  public void paSetPrice(float val) throws PException;
  public float paGetPrice() throws PException;
  …
}

public abstract class ProductBinding implements PBinding {
  //Writes an object into the data store – uses the PAccessor
  //paGetXXX methods to get the memory values to be stored
  public void write(Object conn, PAccessor pa) {
  …
  }
  //Reads an object from the data store – uses the PAccessor
  //psSetXXX methods to put the obtained values into memory
  public void read(Object conn, PAccessor pa)
          …
  }
  …
}
```

### 4.2 Internal Object Model

The basic principle of the persistence framework is to use binding objects between data stores and memory instances. Bindings perform the relevant mapping of Java structures to persistent structures which are specific to a particular DS, as illustrated in Figure 3. In order to do so, the persistence framework must be aware of the type of objects that are stored and also adaptable to MIM object models.

For this purpose, the persistence framework uses an object model to specify the types of entities it can store. This object model has been designed to be as close as possible to the Java object model and represents the structural part of objects. All entities described by this model are objects of persistent classes, composed of fields. There are three kinds of persistent classes: abstract classes, classes and generic

classes[4]. There is no notion of value, dependent object, or second-class object in the model (this should be provided by a higher-level layer such as the MIM).

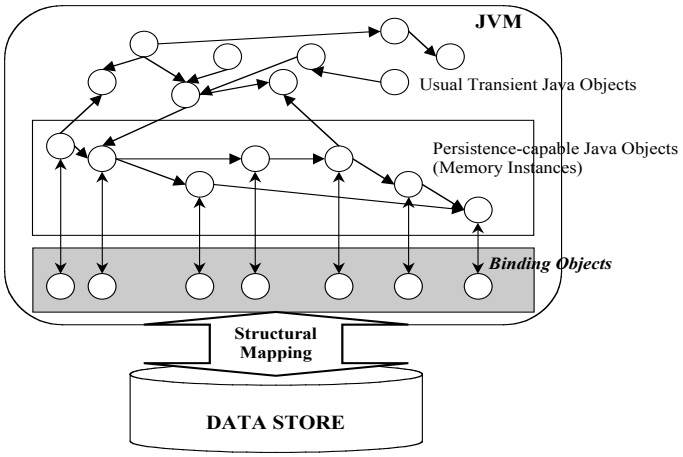


**Figure 3**: Structural mapping performed by bindings (representing persistent objects of the framework model).

**Classes and Fields.** A persistent class defines a persistent object. It is composed of a set of fields described by a *field name* and a *field type*.

A field type belongs to one of the following three kinds:

- A *primitive type*, which is essentially one of those defined by Java.
- A *persistent class*, which means that the value of the field is a persistent name (representing the persistent identifier – see Section 4.3) that references a persistent object.
- A *persistent generic class*, which means that the value of the field is a persistent name that references a collection object (see below).

As in Java, a persistent class may be abstract, in which case no DSI can be created for this class. Abstract classes are used to factorize definitions between different non-abstract classes.

A class is always declared as belonging to a package, which is equivalent to the package concept in Java.

Persistent objects are always created within a persistent class, be it generic or not.

A class may inherit from other classes. Multiple-inheritance is supported among abstract classes, as well as between a non-abstract class and abstract classes (i.e. a class may derive several abstract classes). Only single-inheritance is supported among non-abstract classes (i.e. a class may be derived from at most one other class).

---

[4] The term "generic class" used in this article is not to be confused with the generic parametric classes of object languages.

**Collections and Generic Classes.** A collection object is defined by a *generic class*. Generic classes have been designed to support different types of collections (e.g. lists, trees, maps).

Thus, a collection object is composed of a set of *indexed elements*. An indexed element has a value, which is the value of the element, and possibly several indexes, usually depending on the structure represented by that collection

As for the fields of a class, a generic class defines the type of its elements. The type of index values is restricted to scalar types (byte, char, short, int, or long) or the String type.

**Persistent Names for Object Identification.** Objects of non-abstract classes and generic classes are identified by a persistent name (see Section 4.3 for more details). The storage structure of the persistent name is declared in the class definition.

Such names can be managed by the underlying DS (e.g. an OODBMS usually associates persistent names to objects transparently), or by the framework users independently of the underlying DS. In this latter case, names are part of the object structure and must be defined as such (e.g. a primary key within an RDBMS). In the case of a DS managed name, a name may be defined as a value of an abstract type. Otherwise, a name may be defined as a basic type (byte, char, short, int, long, or string), or as a composite name. A composite name is a list of fields whose type is either one of the basic ones presented above, or is linked to one of the persistent object fields.

**Analysis.** The persistent object model is independent of any persistence model. Possible persistence models include "persistence by reachability" (objects are made persistent as soon as they are reachable from a "root" persistent object), "class-attached persistence" (all objects of a particular class are persistent), or "explicit per object persistence" (objects are explicitly made persistent using a dedicated command, such as "makePersist"). Our framework can support all these models, although it is the role of the upper layers using the framework to implement them. Conversely, the framework must be as neutral as possible with respect to the model supported by the underlying DS.

**Example.** The example below shows the XML descriptor of the object model for the class *Product* with two fields, *name* and *price*. The persistent name of the class *Product* is composed of one field, *name*.

```
<persistence>
  <package>invoice</package>
  <class abstract="FALSE" name="Product">
    <field name="name">
      <primitive-type type="string"/>
    </field>
    <field name="price">
      <primitive-type type="float"/>
    </field>
```

```
    …
    <name-def name="">
      <field-ref field-name="name"/>
    </name-def>
  </class>
</persistence>
```

### 4.3 Persistent Object Identifiers and References Management

The persistence framework has to manage persistent object identifiers, assuming that each DSI representing a memory persistent object (i.e. an MI) has at least one persistent identifier, i.e. an object allowing its identification within its DS. Also, according to the object model, the persistence framework must deal with references between objects. In order to manage these, two naming concepts are used: *names* and *naming contexts*, borrowed from RM ODP [13].

**Names and Naming Contexts.** A *name* is an object that identifies a persistent object within a particular naming context (i.e. the name is valid in a particular naming context). A *naming context* is an object that associates an entity to each name it manages. A *name* is strongly dependent on the underlying DS. It may be composed of application-related information, such as primary keys in RDBMS. It may be a system-managed identifier (independent of user data), such as in OODBMS. The objective of the framework is not to make any assumption about the way names are managed. This means that the framework can support any kind of name.

Conceptually, these two concepts allow us to organize Data Store Instances into sets (i.e. naming contexts). These sets can also share Data Store Instances through their names. A naming context can, for example, represent instances of a given class or generic class, or tuples of a related data store. In naming concepts [ISO ODP part2], a name represents both an object identifier and a reference to this object.

The example of Figure 4 describes a complex situation where DSIs are stored within two Data Stores, *DS1* and *DS2* (a DSI always belongs to only one DS). It also shows fives naming contexts, *PNC1* to *PNC5*. A given DSI may have names valid in several naming contexts. For example, the DSIs of *PNC3* also have names within *PNC5*. Furthermore, a DSI may have names that are valid beyond its DS. This is the case for the DSIs of *PNC2* that also have names within *PNC4*, which federates the DSIs of two DSs.

Naming management essentially consists in three operations on naming contexts:

1. *export*: on a given naming context, this operation creates an association between an entity and a name within this naming context; the entity is either a persistent object or another name valid in another naming context. The entity is a parameter of this operation that yields the name designating it within this naming context. Performing the export operation again, by exporting the yielded name to another

naming context, creates a so-called naming chain[5]. The behaviour of this operation depends on the naming context semantics. For example, it can look up an existing association involving this entity; then it yields the existing associated name or creates a new one, if no association exist. Or it can systematically create a new name, having the entity designated through different names within this naming context. In either case, name creation always occurs when calling `export`.

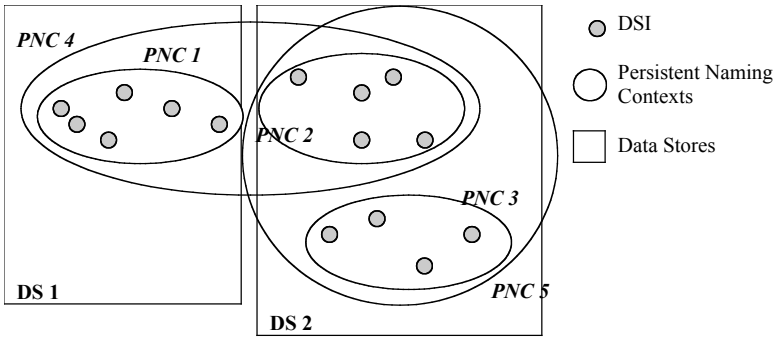


**Figure 4**: Data stores (DS), Data Store Instances (DSI) and Persistent Naming Contexts.

2. *resolve*: this operation is the reverse operation to export. It retrieves the entity that has been previously exported. It is used to look up the entity designated by a particular name within this naming context. The name is a parameter of this operation which yields the associated entity, if any exists. If this name is part of a naming chain, the operation yields the preceding name to the one passed as parameter.
3. *unexport*: this operation is used to remove an association within the current naming context. It takes a name as parameter and removes the association with the entity it designates within this naming context, if any exists.

In our framework, an entity represents a *PBinding* object (which represents a DSI) and a name represents a *PName* object. Naming context objects implement the *PNamingContext* interface. In order to manage binding objects and *PName* objects, two other kinds of objects are introduced: *binders* and *class mappings*.

In order to allow their storage in the DS, another important functionality is the encoding/decoding of names.

---

5 A naming chain is also called a naming graph in RM ODP [13]. It is defined as a directed graph where each vertex denotes a naming context, and where each edge denotes an association between a name appearing in the source naming context and the target naming context.

**Binders and Class Mappings.** The association between *PBinding* objects and *PName* objects is maintained by a *binder* object which implements the *PBinder* interface. Within a binder, there is always a unique persistent name that gives access to its associated DSI. Thus, a binder is a particular kind of naming context (the *PBinder* interface inherits from the *PNamingContext* interface).

Class mappings perform the creation of bindings. Each class has a class mapping associated to it. This is a factory producing binding objects (implementing *PBinding*). The management of *PNames* is delegated to a binder, also associated to the class. A class mapping object implements the *PClassMapping* interface and can be seen as the starting point for the management of the object instances of a related persistent class (*PBinding* and *PName* objects).

There are two cases when a class mapping creates a binding:

- A DSI already exists and a binding is requested by an MI in order to synchronise its values with it. This corresponds to a *bind* operation which associates a name to this binding, the binding being thus activated.
- No DSI exists. In this case, a binding is requested in order to create a new DSI. This corresponds to an *export* operation on this binding which creates a new name.

**Example (continued).** In addition to the class *Product* of the previous example, consider an additional class *InvoiceItem* representing the purchase of a given quantity of a product. This class has a field, which is a reference to a *Product* object. Figure 5 shows the corresponding naming chain managing the reference between an *InvoiceItem* object and the associated *Product* object. *pn1* is the persistent name of a *Product* object, managed by the binder associated to the *Product* class, *nc1*. Name *pn2* is the persistent name of the *Product* referenced from the *InvoiceItem*. Resolving *pn1* within *nc1* would yield *pn1* again, since *nc1* is the binder, and we are at the end of the naming chain. The following relationships hold:

```
pn2 = nc2.export(pn1)
pn1 = nc2.resolve(pn2)
```

The binding object corresponding to the *Product* object is obtained by calling `nc1.bind(pn1)`.

**Analysis.** Since the binder maintains the association between bindings and persistent name objects, it is always the end of a naming chain. This is why resolving a name within a binder always yields the final name related to its binding object. Thus, a binder can also be seen as the correct entry point to introduce caching mechanisms.

The use of names and naming contexts allows the introduction of any kind of reference semantics between DSIs, be they stored into co-located or into distributed DSs, activating MIs of referrer and referee into the same process or into distributed ones. In the case of distribution, this approach is the same as the one used in the Jonathan framework [8] used to implement open and flexible Object Request Brokers.

The scope of a reference can be larger than a simple persistent class. It may cover several persistent classes within a DS: this is the case when naming contexts are used to deal with polymorphism. It may also cover several persistent classes within several DSs: this is the case when using naming contexts to federate DSs, such as *PN4* in Figure 4. As the framework argues for openness with respect to the federation of DSs, naming contexts are the feature which guarantees this openness. Indeed, users of the framework can assign (specialised) naming contexts to each reference field of a particular persistent class.

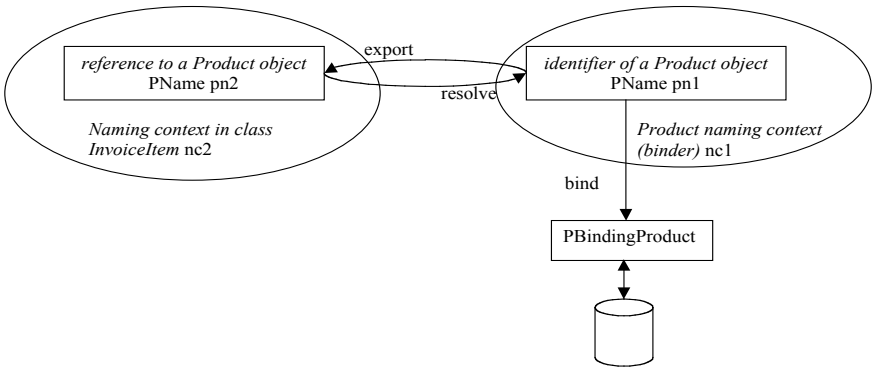


**Figure 5**: Naming chain.

### 4.4 The Persistence Framework at Work: Mappers

The *mapper* is the root object which gives access to persistence functions. It is usually tied to one DS for which it manages synchronization between MIs and DSIs. Several mappers can coexist simultaneously within a JVM. A mapper gives access to binders which manage bindings. One binder is attached to each persistent class defined within the persistence framework.

The mapper also manages meta information for mapping the object schema to the corresponding DS. The mapping meta information describes the structural projection of objects in the object model onto the DS model. Associated to each class, and for each kind of data store within which the persistent object are stored, the mapping meta information contains *mapping definitions*. Such mapping definitions are similar to the ones described in [5]. The mapping information depends heavily on the type of data store. The architecture of the framework ensures that it can be extended to take into account new types of data store.

As an example, for relational database mapping, several mapping rules can be used in order to allow legacy database integration to be supported. A given class is mapped to a main table. If a class is projected onto more than one table, several external tables are possible, reachable through join conditions. Moreover, in the case of class references, it is possible to store the reference either in the class table, or as backward

references in the table of the referenced class (in this latter case, the table is "collocated"). Finally, there are several possibilities for defining the mapping of generic classes, with or without an additional join table.

### 4.5 Type Management

Persistent objects are defined by persistent classes, which are composed of typed fields (see Section 4.2 for more details about the object model). When assigning a reference to a persistent object as the value of a field of another persistent object, type verification may occur. Even if Java enforces some kind of static typing, there are many situations where typing is poor and the relation between the Java typing mechanism and persistence typing is not always straightforward. This is especially true when dealing with generic classes. Furthermore, in the case of the federation of DSs, typing should be enforced in a larger context, potentially involving distributed JVMs.

In order to be able to perform type checking, typing information must be carried along with persistent names. Type verification can then occur when names are assigned as references, that is when a binding is going to store them. It is verified against the type associated to the field of the corresponding class: the type of the reference must satisfy the "isa" relationship which is enforced for the type of this field.

Types are closely related to names and are organized around type spaces. One type space is associated to each mapper. As naming contexts do not always share type spaces, when a name is exported into another naming context of another mapper, its type must be present in the type space of this destination naming context. If it is not present, it must be defined within this space, which means that its complete definition must be imported from the type space of the original naming context into the destination naming context. An object type in the framework is fully defined by a class name and the super classes of this class, recursively.

A type space can be combined with the object model containing the descriptions of all classes, as it defines a sub-part of the object model.

## 5 Query Framework

As objects are made persistent within data stores, the query framework must be able to interrogate such data stores. Thus, the problematic of the query framework is similar to that of query management in data integration systems [10] [12] [28]. In such systems, a global query expressed on an integrated schema is decomposed into subqueries corresponding to the underlying data stores.

In our case, the query framework must be able to fulfil the two following objectives. First, it must be possible to express queries directly on a DS. In this case, the framework must be aware of the schema exported by the DS. Second, it must be possible to express queries on the persistent object schema of the persistence

framework. The expression of such queries is independent of the underlying DS onto which persistent data projected. This ensures that a given query remains valid even when data is migrated from one DS to another DS, provided that the object schema remains the same: only the mapping meta information is required to change. This second case is closer to the data integration systems: the persistent object schema can be seen as the integrated schema.

If queries are expressed directly on the DS, only values are returned as results. This is the case when querying directly relational databases. In the second case, the interaction of the query framework with the persistence framework makes it possible to obtain object references as query results, since the persistence framework manages persistent object names.

The purpose of queries is to select persistent data based on semantic criteria (e.g. products in the 100 to 200 price range, invoices for a given customer in a given time frame). The query framework makes it possible to express, optimize and evaluate queries over persistent data.

In the following sections, we detail each component of the query framework.

### 5.1 Query Expression

Rather than imposing yet another query language, the query framework offers a programmatic way to express queries. This approach guarantees adaptability to MIM query languages. This has been validated by the integration of the query framework in several systems (see Section 6). A query is expressed by the programmatic construction of an algebraic tree. This tree can be seen as an internal pivot representation of the corresponding initial MIM query.

As the query framework deals with objects, the algebra supports at least the algebraic operations described in [28] [30], which, in addition to the relational algebra (join, selection, projection), include collection manipulation operators, such as grouping (nesting), ungrouping (unnesting) or flattening on collections of tuples.

The algebraic tree is called a *query tree*, implementing the *QueryTree* interface. To each *QueryTree* object is attached a tuple structure, described as a list of typed named *fields* (the equivalent of the SELECT clause in SQL). For the root of the query tree, the tuple structure represents the type of the query results.

In a query tree, a *query node* represents an operation of the query algebra. The *QueryNode* interface inherits from the *QueryTree* interface. Several classes implement *QueryNode*, such as *Join*, *Selection*, *Nest*, *Unnest* or *Union*.

Data sources are represented as *query leaves* of the query tree. The *QueryLeaf* interface inherits from the *QueryTree* interface. The query framework contains, for each type of data store, query leaves corresponding to data of the data store. For example, a relational database query leaf can represent a relational table or a SQL query and its fields correspond to the SELECT part of the SQL query; an object database query leaf typically represents all instances of a given class. Other query leaves include persistent class extents (see below).

Fields of a query node can be defined in three ways.

1. A field can be *propagated* from another query tree: the latter query tree becomes a child of the current query node.
2. A field can be *calculated* from an expression applied to fields of other query nodes (see below for a detailed description of expressions).
3. A field can be the result of a nesting (grouping) operation: in this case, the nested field is a collection of tuples. The algebraic operation of the corresponding query node must be a *Nest*.

For a given query node, the set of query trees reachable through propagated or calculated fields constitutes the children of the current query node (the equivalent of the FROM clause in SQL).

A query node can also be attached a *query filter* (the equivalent of the WHERE clause in SQL). The query filter makes it possible to select data from the children query trees. The query filter is expressed as a well-formed Boolean and/or arithmetic *expression* over operands and operators. Operands of expressions can be constants, parameters, field operands on fields propagated from children nodes, or other expressions. Operators can be arithmetic operators (plus, minus, etc), logical operators (and, or) or string manipulation operators (concat, etc). As query trees, query filters are constructed programmatically. Query filters are trees where nodes are operators and leaves are operands. This enables extensibility of the query framework in order to introduce easily new types of operators.

**Integration with the Persistence Framework.** Integration with the persistence framework is first done through *extent* query leaves (the *ClassExtent* interfaces inherits from the *QueryLeaf* interface): an extent conceptually represent all objects of a given persistent class, independently of its mapping to a particular DS. The tuple structure associated to an extent contains one field for each field of the persistent class, plus the persistent name (*PName*) of the persistent class. The persistent name of a class can be manipulated as any other field (e.g. it can be projected or it can be part of a query filter).

Another important integration aspect concerns the support of path expressions. Navigation through the reference fields of persistent classes is done using the navigator operator. It makes it possible to construct a field operand for a field reached by a path expression (e.g. *invoiceItem.product.price*).

**Query Expression on the Example.** Consider that the example contains the additional class *Supplier*, with the field *supplierName* mapped onto the table *SE_SUPPLIER* with the column *SNAME*. Consider that the class *Products* has the additional field *supplier* of type reference to a *Supplier*. Consider the query *"Retrieve the product object identifiers (its PName) for products in the 100-200 price range supplied by "MySupplier"*.

The corresponding query tree is illustrated in Figure 6 below.

The code expressing this query looks like:

```
//creation of the two query leaves on Product and Supplier
ClassExtent pExtent = new ClassExtent("invoice.Product");
```

```
ClassExtent sExtent = new ClassExtent("invoice.Supplier");
//creation of the query tree root
QueryNode myQuery = new JoinProject();
//projection of PName(Product) into the result: pExtent becomes a
//child of myQuery
myQuery.addPropagatedField(pExtent.getField("PName");
//query filter
Expression filter = new And(
  new And(
    new Greater(new FieldOperand(pExtent.getField("price")),
        new VariableOperand(100)),
    new Lower(new FieldOperand(pExtent.getField("price")),
        new VariableOperand(200))
    ),
  new And(
    new Equal(new FieldOperand(sExtent.getField("supplierName"),
      new VariableOperand("mySupplier")),
    new Equal(new FieldOperand(sExtent.getField("PName"),
      new FieldOperand(pExtent.getField("supplier"))
  )
);
//assigning the filter to myQuery – sExtent becomes another child
myQuery.setFilter(filter);
```

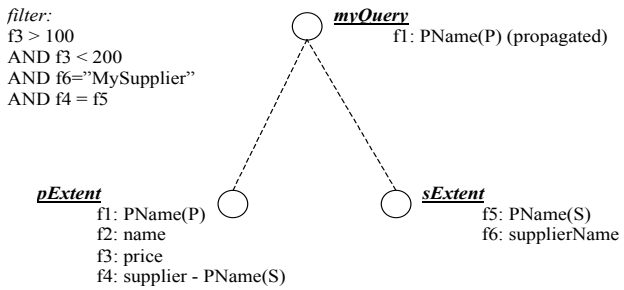


**Figure 6**: Query expression: example of a query tree.

## 5.2 Query Optimization

Query optimization is an important step in query processing and is classical in database systems [11]. The approach used in the optimizer of the query framework is to use rewrite rules which transform a query tree in another query tree.

The three main tasks of the query optimizer are:

1. If appropriate, delegate query evaluation to the DS.

a. The query optimizer first rewrites extents into the corresponding data store query leaves using mapping meta information. An important aspect is the management of persistent names: when rewriting an extent into a DS query leaf, the persistent name, which is a field of the extent, is transformed into a calculated field on the corresponding fields of the DS query leaves (e.g. the primary key in the case of a relational database mapping). For example, in the case of mapping to a relational database, the query leaves of class extents are rewritten into relational database query leaves. The query framework typically contains one such rewrite rule per data store type.
b. Then, depending on the data store evaluation capabilities, query leaves corresponding to the same data store are moved within the query tree so that they can be close to each other. A first generic rewrite rule is responsible for this task: *GroupSameDBRule*. Then, the query optimizer collapses these query leaves into a single query leaf in order to delegate the evaluation to the underlying DS. Currently, the framework contains one such rewrite rule per DS type. For more information about optimization depending on data store evaluation capabilities, see the work done in [10] [28].

For example, in the relational DS case, several query leaves on the same database can be grouped into one single relational database query leaf, with a single SQL query. On the other hand, in the case of simple file storage (one object per file), the file system does not have the capacity to evaluate queries, and the corresponding query leaves cannot be collapsed.

The DS query generated by the optimizer depends on the DS type. Even within the same DS type, various adaptors may be necessary to accommodate language differences (for example in the SQL case).

2. Apply "classical" optimization rules on the resulting query tree. This involves optimizations such as pushing selections down, removing useless projection nodes, which do not contain any filter.
3. Finally, the query optimizer produces the final query execution plan by choosing the appropriate join evaluation algorithms (hash join, nested join, etc). This can be done using statistical data on the data store (selectivity factors, histograms, etc).

The query optimization module contains a set of rewrite rules, each rule taking a query tree as input and producing a query tree as output. Configuring the query optimizer consists in choosing the rules to apply and the order in which they should be applied.

**Query Optimization on the Example.** If both tables reside on the same DS, the query tree of Figure 6 is collapsed into a query tree containing a single relational query leaf, and a query node for constructing the *Product* persistent name, as illustrated in Figure 7 below. The optimizer performs tasks 1.a and 1.b above to produce this optimized query tree.

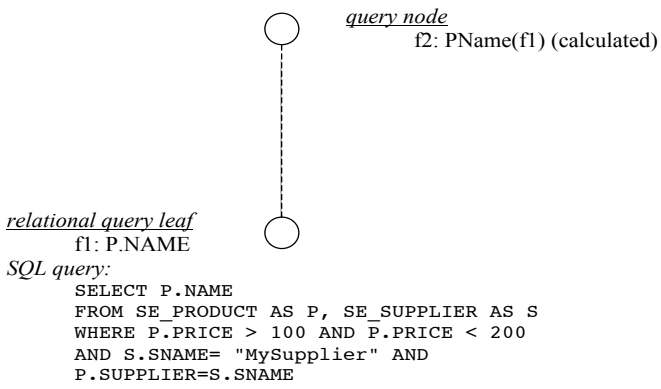


**Figure 7:** Example of an optimized query.

### 5.3 Query Evaluation

The query tree resulting from the optimization stage can now be evaluated. The evaluation process iterates over the collection of tuples issued from the query leaves. It works as a pipelined evaluation, with data flowing between query nodes.

For a given query tree, a global query evaluator first parses the optimized query tree and:

1. compiles the query filters of each query node by constructing typed buffer structures. This process avoids creating new objects when evaluating filters.
2. assigns a local query evaluator to each query node;
3. creates intermediate data structures for storing the tuple collections corresponding to each query node evaluation;
4. links each query leaf with its appropriate data store gateway;
5. coordinates the evaluation on the local query node evaluators. Each query node evaluator evaluates the corresponding compiled query filter over data coming from its children query node evaluators.

**Prefetching.** The persistence and query middleware contains an important optimization function. It consists of *prefetching* object fields at query evaluation time. A typical usage of the persistence and query framework is that the user first submits a query, asking for a set of persistent names of objects answering a given condition; later, the user interacts with the persistence framework to load the corresponding objects one by one into memory.

A naïve sequence of execution consists in performing a first access to the DS during query evaluation, in order to retrieve the persistent names, and then as many accesses to the DS as there are objects to be loaded in order to retrieve the field values.

The prefetching optimization gathers enough data from the DS at query evaluation time (i.e. the field data of the objects, and not just their persistent name) in order to be able to load the complete MI when requested without any further access to the DS.

### 5.4 Analysis

The adoption of a programmatic approach to express queries, together with the separation of the query optimizer and the query evaluator, ensure the high adaptability of the query framework at several levels:

1. Extensions of the optimizer. Given the rule-based structure of the optimizer, its extension is simply done by adding new rewrite rules.
2. Support of a new data store type. This is possible by adding the query leaves corresponding to this new DS type, and adding the rewrite rules responsible for transforming extents into the corresponding query leaves. Moreover, if the DS has some query evaluation capacity, the corresponding rewrite rule responsible for collapsing query leaves must be added.
3. Algebra and evaluation algorithms. The query framework has been designed to make it possible to introduce easily new algebraic operators (we have experienced it with aggregation), and new join evaluation algorithms (new query node evaluators).
4. Finally, new operators for expressions can also be added easily.

## 6 Implementation and Validation

The persistence and query middleware framework has been fully implemented in Java in the context of the ObjectWeb open source middleware consortium [25], respectively as the JORM [22] and MEDOR [23] projects.

In terms of mapping, legacy relational databases are supported with complex mapping rules. A simple mapping to files as well as a prototype mapping to a simple object database have also been implemented.

The implementation of mappings to several data store types has validated the extensibility of the framework. This includes the support of mapping definitions, generation of bindings for persistence and query management (rewrite rules and query leaves).

The middleware framework presented in this article has also been coupled with other middleware components related to persistence in order to integrate caching, concurrency and transaction management. This integration has shown that the persistence and query framework is indeed adaptable with regard to these other technical features.

The persistence and query framework has been used to implement transparent persistence in the JOnAS J2EE server [21] and the Speedo JDO implementation [24]. Both systems generate the object model and mapping meta information from their

respective descriptors. In both cases, the corresponding binding classes are generated. Flexibility in terms of MIM is illustrated by the two following different approaches for integrating the generated binding classes:

1. In the case of JOnAS, the integration is done at the code generation level with inheritance: the generated binding class extends the JOnAS context switch. A container object implements the *PAccessor* interface.
2. In the case of Speedo, an additional proxy class is generated for each user class. This proxy class implements the necessary JDO interfaces imposed by the specification, as well as the *PAccessor* interface, and extends the generated binding class. Finally, the original user class is enhanced using byte code manipulation and is merged with the binding and proxy classes.

Regarding queries, the initial JDOQL or EJB QL query is transformed into the corresponding query tree. The query framework optimizes and runs the query, and the query results are loaded into the corresponding application server objects.

These two experiments have proved the re-usability of our persistence and query framework in different contexts, with different object life cycle management policies, and different implementation solutions.

Performance tests are under way in order to compare the persistence and query framework with other commercial products in the contexts of J2EE (JOnAS) and JDO (Speedo).

## 7 Conclusion and Future Work

This paper has presented an adaptable and flexible middleware framework for the persistence and querying of Java objects. The framework's downward and upward adaptability and its independence from other persistence aspects, such as caching and transactions, have clearly been demonstrated throughout this paper. The framework has been fully implemented and has been used in several contexts (JDO, EJB), thus validating its re-usability, as well as our initial objectives.

Regarding its extension to other data stores (new mappers), the possibility of the Lightweight Directory Access Protocol LDAP [37] as a data store is currently being investigated. Another extension (or personality) concerns the implementation of the Universal Description, Discovery and Integration protocol, UDDI [18], which is being carried out as an internal France Telecom project.

We are currently working on the integration of the persistence and query framework with other middleware aspects, such as distributed caching.

The persistence and query framework will be reengineered using the Fractal component model, and more precisely its Julia reference implementation [20]. Using Fractal should improve the integration and extensibility of the framework, simplify the configuration of the naming facilities, and provide a better structure for the rule-based query optimizer.

## Acknowledgements

The authors would like to thank the anonymous referees for their many constructive and encouraging comments.

## References

1. M.P. Atkinson, P.J. Bailey, K. Chisholm, W.P. Cockshott, R. Morrison: "An Approach to Persistent Programming". Computing Journal 26(4): 360-365, 1983.
2. M.P. Atkinson and R. Morrison. "Orthogonal persistent object systems". VLDB Journal, 4(3), 1995.
3. M. Atkinson, M. Jordan. "Providing Orthogonal Persistence for Java". Lecture Notes in Computer Science, Vol 1445, 1998.
4. M.P. Atkinson, M.J. Jordan, S. Spence. " Design Issues for Persistent Java: a type-safe object-oriented orthogonally". In Proceedings of the 7th Workshop on Persistent Object Systems, Cape May (NJ), USA, 1996.
5. M. Baldonado, C.-C.K Chang, L. Gravano, A. Paepcke, "The Stanford Digital Library Metadata Architecture". Int. J. Digit. Libr. 1 (1997) 108–121.
6. R.G.G. Cattell, D.K. Barry, M. Berler, J. Eastman, D. Jordan, C. Russel, O. Shadow, T. Stanienda, and F. Velez. The Object Data Standard: ODMG 3.0. Morgan Kaufmann Publishers, 2000.
7. The CORBA Persistence State Service Specification. http://www.omg.org/
8. B. Dumant, F. Horn, F. D. Tran, J.-B. Stefani. "Jonathan: an Open Distributed Processing Environment in Java". IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing, The Lake District, U.K., September 1998.
9. P. Ferreira, M. Shapiro, X. Blondel, O. Fambon, J. Garcia, S. Kloosterman, N. Richer, M. Roberts, F. Sandakly, G. Coulouris, J. Dollimore, P. Guedes, D. Hagimont, S. Krakowiak. "PerDiS: design, implementation, and use of a PERsistent Distributed Store". Technical report, QMW TR 752, CSTB ILC/98-1392, INRIA RR 3525, INESC RT/5/98, October 1998.
10. H. Garcia-Molina, Y. Papakanstantinou, Q. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, V. Vassalos, J. Widom "The TSIMIS Approach to Mediation: Data Models and Languages". Journal of the intelligent Information Systems (JIIS). 1997.
11. L.M. Haas, J.C. Freytag, G.M. Lohman, and H. Pirahesh. "Extensible Query Processing in Starburst". In Proceedings of ACM SIGMOD International Conference on Management of Data, pages 377--388, Portland, Oregon, May 1989.
12. L.M. Haas, R.J. Miller, B. Niswonger, M. Tork Roth, P.M Schwarz, E. L. Wimmers. "Transforming Heterogeneous Data with Database Middleware: Beyond Integration". IEEE Data Engineering Bulletin, vol 22, number 1, pages 31-36, 1999.
13. ISO. ITU/ISO Reference Model of Open Distributed Processing – Part 2. Foundations, International Standard ISO/IEC 10746-2, ITU-T Recommendation X.902, 1995.
14. ISO. ITU/ISO Reference Model of Open Distributed Processing – Part 2. Architecture, International Standard ISO/IEC 10746-3, ITU-T Recommendation X.903, 1995.
15. ISO 10303-22; Industrial automation system and integration – Product data representation and exchange – Part 22. Implementation methods: Standard Data Access Interface specification. 1996.

16. R. E. Johnson. “Framework = (components + patterns): How framework compare to other object-oriented reuse techniques”. Communications of the ACM, 40(10):39-42, October 1997.
17. J.E.B. Moss, A.L. Hosking, “Approaches to Adding Persistence to Java”, in First international Workshop on Persistence and Java, Drymen, Scotland, September 1996.
18. Oasis, The Universal Description, Discovery and Integration (UDDI), http://www.uddi.org/
19. The Object People. TopLink: Java object-to-relational persistence architecture. http://www.objectpeople.com/
20. ObjectWeb Consortium. The Fractal component model and framework, http://fractal.objectweb.org
21. ObjectWeb Consortium. JOnAS: Java Open Application Server, http://jonas.objectweb.org
22. ObjectWeb Consortium. JORM: Java Object Repository Mapping, http://jorm.objectweb.org
23. ObjectWeb Consortium. MEDOR: Middleware Enabling Distributed Object Requests, http://medor.objectweb.org
24. ObjectWeb Consortium. Speedo: JDO implementation. http://speedo.objectweb.org
25. Objectweb: Consortium for the promotion and the development of open source middleware. http://www.objectweb.org
26. OJB: Object Relational Bridge http://db.apache.org/ojb/
27. Persistence Software. Persistence PowerTier for J2EE http://www.persistence.com/products/powertier/index.php
28. A. Tomasic, L. Rashid, and P. Valduriez. “Scaling Heterogeneous Databases and the Design of DISCO”. In Proc. 16th ICDCS Conf., Hong Kong, 1996.
29. M.T. Roth, P. Schwarz. “Don’t Scrap It, Wrap It! A Wrapper Architecture fur Legacy Data Sources”. In Proc. of the 23rd VLDB Conference, Athens, Greece, 1997.
30. G. M. Shaw, S. B. Zdonik A Query Algebra for Object-Oriented Databases. In Proceedings of the Sixth International Conference on Data Engineering, February 5-9, 1990, Los Angeles, California, USA; pp 154-162.
31. Sun Microsystems. Java 2 Enterprise Edition Specification. http://java.sun.com/j2ee
32. Sun Microsystems. Java Data Objects Specification. http://java.sun.com/products/jdo
33. G. Wiederhold, “Mediators in the Architecture of Future Information Systems”, IEEE Computer, pp. 38-49, March 1992.