TRANSPARENT OBJECT PERSISTENCE

DR AMMAR JOUKHADAR
INFORMATICS FACULTY
DAMASCUS UNIVERSITY

ABSTRACT

Object persistence is a major problem concerning enterprise applications, since Object Oriented Databases are not mature enough, and Relational Databases can't store objects. Therefore, a solution has to be provided to solve this issue.

A lot of solutions have been introduced, some are very heavy and require a huge infrastructure, and others are still in early stages.

This article makes an overview of the different techniques used to persist objects and propose a novel solution based on some advanced meta data in order to implement a transparent objects persistence service for distributed applications. This solution support storage, querying, and retrieval of objects from data stores. In addition, it fully supports OO (Object Oriented) features; including: aggregation and composition relations, both uni- and bi-directional, inheritance, and polymorphism.

Our solution is implemented for J2EE platform and it uses standards to be vendor-independent regarding the underlying data store, so applications can be transparently ported from one data store to another.

KEYWORDS

Transparent Persistence, OO Storage, Storage independence, inheritance, polymorphism, Meta Modeling, Java, J2EE.

1. CURRENT PERSISTENCE ISSUES

1.1 UNDERSTANDING OBJECT PERSISTENCE

For years, the supposedly straightforward task of loading and storing data has unnecessarily complicated the developments of applications. However, a lot of techniques have been introduced to assist developers solving this problem.

In Java (and other object oriented programming languages) an object is an instance of a class. As such it has state (its attribute values) and behavior (its methods). The collection of all class definitions that comprise an application is known as the application's object model. These classes perform a variety of functions: some render user interfaces; some manage system resources; some represent application events. However, within each object model there is usually a distinct set of objects that are direct abstractions of business concepts – typically with names to which non-technical people would ascribe meaning. In an order processing application these may be "Customer," "Order," and "Product." For a financial application they might be "Client," "Account," "Credit Entry," and "Debit Entry." In each case these objects are modeling the business domain in which the specific application will operate, and thus they are collectively referred to as the domain object model.

The domain object model is of particular importance to application designers. It is these objects that represent the primary state and the behavior available to the application. They will be the focus of many design workshops, since they represent the concepts which the application's target user community understand, and in which they have specific expertise. Perhaps most importantly, it is these objects that typically need to be stored (somewhere and somehow) between invocations of the application and shared between multiple simultaneous users.

The storage of these objects, beyond the lifetime of the Java Virtual Machine (JVM) in which they were instantiated is called *Object Persistence.*

There are, of course, other classes beyond those which fit naturally into the domain object model, which may require persistence services (e.g. log messages). Object persistence is by no means restricted to the domain object model, but it is here that we find the majority of classes for which persistence must be provided.

1.2 Current Techniques for Persistence

Persistence requires the storage of object state for future retrieval. Various underlying mechanisms are in use in the industry, but by far the most common approach is to use a relational database management system (RDBMS) accessed through a combination of JDBC and SQL. Alternative mechanisms include file system-based storage and object database management systems (ODBMS). A persistence infrastructure is often layered on top of the data store, examples being Entity Beans and Enterprise Application Integration (EAI) frameworks.

1.2.1 Relational Databases

RDBMS technology has been widely adopted in the last 15 years because of its freeform definition of data (rows and columns), flexibility of ad hoc queries, and transactional reliability (begin, rollback, commit). Due to extensive standardization efforts in the RDBMS market, all such databases can be invoked using the SQL. Although variations exist in the SQL dialects used by various databases, support for the SQL-92 standard is relatively widespread.

Java applications using relational databases for persistence typically invoke the database by passing SQL commands to the database server through an API called Java Database Connectivity (JDBC). SQL statements are constructed as string objects, which are then passed to the database server for compilation and execution.

Use of JDBC for the persistence of objects, although widespread, presents a number of difficulties. Firstly, the developer must know SQL and use it to implement every manipulation of persistent data. Secondly, the developer must map object attributes to the columns of one or more tables. This mapping is often non-intuitive, and is required because of the so-called “impedance mismatch” between the notions of an object and a database row. Thirdly, once implemented, the relative lack of portability offered by SQL may restrict the persistence code from working unaltered against an alternative RDBMS implementation, thereby locking the application into one vendor’s technology. Finally, the weak type-checking and deferred compilation of SQL statements means that many errors cannot be detected at compilation time, although this can be mitigated when tools such as SQL/J are used.

1.2.2 File System

File systems are usually considered to be lightweight storage solutions. A file system is capable of storing data in files of a user-defined format, but does not inherently support transactions or automatic data integrity functions.

The one advantage that file system do provide is that they require little by way of supporting services beyond the operating system itself. As such they are commonly used for persistence within embedded applications where system resources are constrained (e.g. the contact list on your mobile phone). However, they are generally not considered appropriate for business-critical transactional information.

1.2.3 Object Oriented Databases

OODBMS are storage environments for objects. The internal representation in which each object is held is hidden from the application developer, who instead uses an API for persisting and retrieving objects. Although they can be extremely efficient at such activity, OODBMS have historically suffered from a lack of ad hoc query capabilities, or inefficiencies where such capabilities do exist. The lack of well-implemented standards for the invocation of persistence services, and the inevitable lock-in of an application to a proprietary vendor's product, have also constrained the adoption of this technology. The ODBMG did put together a standard API for accessing object databases, but this has done relatively little to improve the industry's uptake of object database technology.

1.3 Object, Object-Relational, or Relational?

Object Oriented database integrate database technology with the object-oriented paradigm. Object orientation was originally introduced within the field of programming languages and has become very popular as a paradigm for the organization and design of software systems. Object databases were originally developed in the mid eighties [PAO-

99], in response to application demands for which the relational model was found to be inadequate.

In object oriented databases, each entity of the real world is represented by an object. Classical examples are:

- Electronic components, designed using a Computer Aided Design (CAD) system;
- Spatial or geographic data, such as geometric figures or maps, managed by Geographic Information System (GIS).

These kinds of objects differ greatly from each other and are managed by specialized applications and systems. A common requirement of all of these applications is that of organizing of the data as complex and unitary objects. This demand is not satisfied by the relational model, in which each 'real world object' is distributed among a number of tables. To view an object in its entirety requires the execution of complex queries that reconstruct the various components of an object from tables in the database, by using joins. Object Oriented databases represent real world objects by means of data objects with complex structure and with rich semantic relationships [PAO-99].

The most relevant features introduced by object oriented databases are:

- The use of inheritance, overloading, and late binding, as defined in the context of object-oriented programming languages.
- The integration of data with the operations (or 'methods') that are used for accessing and modifying objects [IDC-97].

There are two approaches for the introduction of objects into databases. Object-Oriented Database Systems (OODBMSs) have taken the revolutionary approach, extending the DBMSs based on the characteristics of object-oriented programming languages. Object-Relational Database Systems (ORDBMSs) have on the other hand assumed the evolutionary approach, by integrating the object concept into the relational model. It should be noted that the two approaches, which are appeared to be in sharp conflict the beginning of nineties [IDC-97], have recently turned out to be convergent [PAO-99].

On the other hand, relational database products have been under development and used much longer than object oriented database products. The RDBMSs are more mature products. The more mature products have been fine-tuned for optimized performance (albeit on a very limited set of data types) and provide a very rich set of functionality, including support of advanced features like parallel processing, replication, high availability, security, and distribution.

There are a wide variety of tools and applications that support the RDBMSs and work with SQL. Ostensibly, the OORDBMSs should be able to take advantage of this support because they are extensions of the RDBMSs their vendors have been marketing for years.

OODBMS products are now also maturing, some with nearly a decade of experience in production applications and often with more advanced functionality than their

RDBMS competitors. In addition, it will be difficult for OORDMSs to be both extensible and retain their legacy advantages.

Ultimately, as ORDBMSs evolve to support more of the OODBMS-like capabilities, they will do so based on new extensions to the products, thereby becoming the new, untried products compared with the OODBMS that are already mature in those areas, with DBMS engines natively designed for objects. Furthermore, although the ORDBMSs struggle with and try to extend inherited architectures and product implementations that assume only the tabular relational model, OODBMSs benefit from foundations built directly to support objects.

The other advantage that RDBMSs and the SQL-based ORDBMSs have is the availability of experienced developers and the plethora of SQL-based developer tools, books, and consultants. OODBMSs won't be in a similar position in the next few years [PAO-99]. SQL is the most universal database language. As a result of the investments made by organizations during the last 15 years, most developers are familiar with SQL and have tools with which to develop. Accommodating SQL or related access mechanisms like ODBC or JDBC minimizes the cost of adoption of proposed extensions or new database capabilities.

1.3.1 The Dilemma

The previously mentioned comparison shows that albeit the big effort done in the field of ORBMS and OODBMS; those products are not mature enough to be used in real-life critical applications, e.g., banking systems, financial tracking, ...etc. thus, most world leaders in the market of enterprise systems prefer using RDBMS [STF-97].

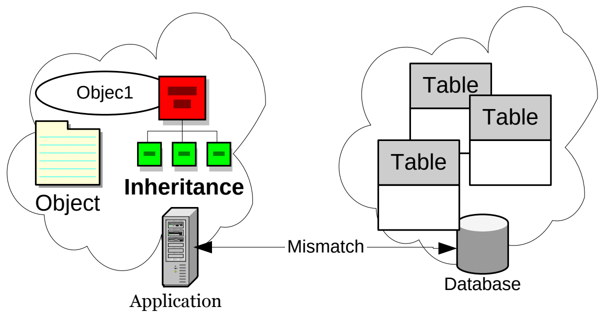


Figure 1 Mixing Different Paradigms

However, there is a huge and largely unnecessary productivity and quality decrease as a result of developing in different paradigms simultaneously like mixing object-oriented development and relational databases – see the figure. The work-around is to introduce a dedicated object-relational mapping layer that transparently maps objects to relational data. Developers using this approach experience a tremendous increase in productivity, escape database vendor lock-in, and simplify maintenance.

On the down side, it requires substantial effort to create an object persistence layer manually [SEB-99].

1.4 THE SOLUTION- OBJECT/RELATIONAL MAPPING

Object/Relational mapping is the process of transforming between object and relational modeling approaches and between the systems that support these approaches. Doing a good job at object/relational mapping requires a solid understanding of object modeling and relational modeling, how they are similar, an how they are different.

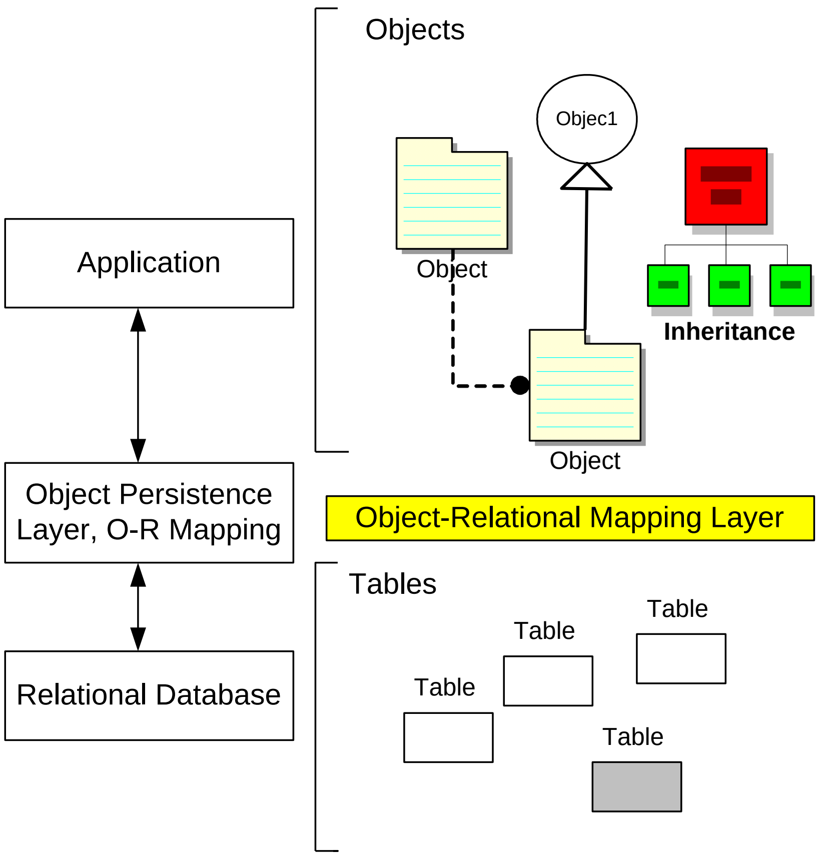


Figure 2 Object Relational Mapping Layer

1.4.1 Design Goals of a transparent O/R mapping layer

There are three important goals to be achieved when implementing an object relational mapping layer, these goals are: decreased coupling, increased cohesion, and increased abstraction.

By routing all data access through an encapsulating layer you decrease the coupling between the application and the storage solution. In other words, your application becomes independent of the underlying data store, usually a relational database, but it can be other storages like file system.

Low coupling is achieved by delivering transparent persistence services[SEB-99]. This allows your application to become agnostic to the "physical" mechanisms of the storage solution, which can be changed in the future without affecting the functionality of the application.

Typically the object persistence layer exposes a full set of access methods and properties that are mapped to the particular underlying storage solution. If you need to change the storage solution, just update the mapping code.

Even seemingly small changes, such as database upgrades, can be a big problem with storage solution specific code scattered all over the application. With the specifics confined to the dedicated object persistence layer, upgrades become a much smoother operation. And if the object persistence layer commits to use standard SQL statements, there will be no changes at all when upgrading or even replacing the underlying relational database system.

Cohesion is about doing one thing great rather than several things poorly. By focusing the persistence code to one separate layer, bugs and performance bottlenecks are easier to isolate and address. Also, the consequences of changes can be predicted with greater certainty.

The object persistence layer should only CRUD (Create, Read, Update, Delete) data - and that is it! You really want to avoid business rules, communication interfaces, or GUI-elements originating from or performing in this part of the application. The reasons are the same that have been driving structured programming for the past two decades: reuse and maintainability.

An object persistence layer that only CRUDs data, but does it great, is more versatile than one that has integrated business rules and other "great" features.

Finally, but most important, the object persistence layer increases the level of abstraction by hiding the complexity of the underlying data model. You can use advanced object-oriented concepts such as inheritance, polymorphism, and complex relationships without having to worry about how it is implemented.

Abstraction is one of the most important factors driving software quality and developer productivity. High-level programming languages and development environments have really empowered developers to create fantastic applications without explicit knowledge about the internals. This is really important because software developers create value by solving business problems, not by patching inefficiencies in the underlying technology.

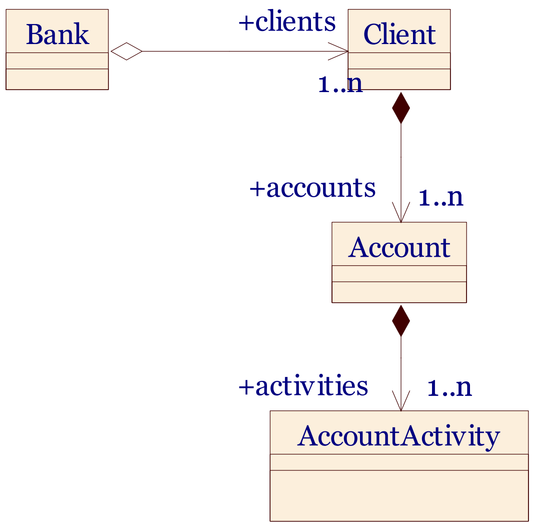
There is a little bit performance penality with abstraction, but the benefits of increased productivity and quality far outweigh the drawbacks.

In addition to the above mentioned goals there are some necessary features to be achieved when implementing an object persistence layer, these are:

- **Object Identity**: An object identifier (OID) is a mean of uniquely identifying a particular object. OIDs are automatically generated. The OID of an object never changes, even across application executions. The concept of OIDs makes it easier to control the storage of objects (e.g., not based on value) and to build links between objects (e.g., they are based on the never changing OID). Complex objects often include references to other objects, directly or indirectly stored as OIDs. When an object is deleted, its OID may or may not be reused. Reuse of OIDs reduces the chance of running out of unique OIDs but introduces the potential for invalid object access due to dangling references. A dangling reference occurs if an object is deleted, and some other object retains the deleted object's OID, typically as an inter-object reference. This second object may later use the OID of the deleted object with unpredictable results. The OID may be marked as invalid or may have been re-assigned [DOD-97].

- **Type**: a type is the specification of an interface that objects will support. An object implements a type if it provides the interface described by the type. All objects of the same type can be interacted with through the same interface. An object can implement multiple types at the same time. Thus a single object may be handled in different ways, and the mapping layer should preserve this capability.

- **Relations and Object Closure**: types can be related with other types, which specifies that the objects of one type can by linked to objects of the other type. Having a relation provides the ability to traverse from one object to the other objects involved in the relation. Object closure is very important, to show its importance let's take a look to the following example, imagine a simple banking



object model where a *Client* has references to many *Account* objects, and each *Account* has references to many *AcountActivity* objects. Given a particular client, the closure of instances include all the *Account* instances referenced by the *Client*, plus all of the *AccountActivity*s referenced by each *Account*. A group of objects that reference each other is caked an *object graph*. Object graphs can be fairly large, particularly when considering the graph of objects reachable from the *Bank* object, which presumably holds references to every *Client*.

- **Concurrency Management**: Databases provide concurrency control mechanisms to ensure that concurrent access to data does not yield inconsistencies in the database or in applications due to invalid assumptions made by seeing partially updated data. The problems of lost updates and uncommitted dependencies are well documented in the database literature. Relational databases solve this problem by providing a transaction mechanism that ensures atomicity and serializability. Atomicity ensures that within a given logical update to the database, either all physical updates are made or none are made. This ensures the database is always in a logically consistent state, with the DB being moved from one consistent state to the next via a transaction. Serializability ensures that running transactions concurrently yields the same result as if they had been run in some serial (i.e., sequential) order. Relational databases typically provide a pessimistic concurrency control mechanism. The pessimistic strategy allows multiple processes to read data as long as none update it. Updates must be made in isolation, with no other processes reading or updating the data. This concurrency model is sufficient for applications that have short transactions, so that applications are not delayed for long periods due to access conflicts. For applications being targeted by OODBMS (e.g., multi-person design applications), the assumption of short transactions is no longer valid. Optimistic concurrency control mechanisms are based on the assumptions that access conflicts will rarely occur. Under this scenario, all accesses are allowed to proceed and, at transaction commit time, conflicts are resolved. [DOD-97]

– **Transaction:** Transactions are the mechanism used to implement concurrency and recovery. Within a transaction, data from anywhere in the (distributed) database must be accessible.

1.4.2 Java's Other Solutions

1.4.2.1 Enterprise Java Beans EJB– Entity Beans

EJB is a part of the J2EE platform. J2EE is a specification for application server technology supporting the middle tiers of an application architecture. J2EE specifically addresses two tiers: the web tier and the EJB tier. The EJB tier contains components that are transactional, scalable, secure and which facilitate the encapsulation of and access to data entities.

The EJB specification enables server-side application component developers to focus on the application logic their components will provide. The developers are abstracted from issues such as transactions and security, and code does not normally have to be written to interface with these services.

Entity beans, a part of the J2EE EJB specification, were designed to present a remote interface to data entities. This allows remote clients to have direct access to the entity bean and thence the data store. Entity beans usually obtain their data from a relational database.

The class of an entity bean identify the type of data it can provide to the client, so a product entity bean would provide product data. Each particular instance of an entity bean is used by a client is associated with a primary key identifying the particular data (e.g. the particular product) that the bean encapsulates.

Entity beans must have their transactions managed by the container (CMT). However, they may choose whether to implement persistence management programmatically with bean-managed persistence (BMP) or declaratively with container-managed persistence (CMP). All entity beans must implement methods for the creation, loading, storing, and removal of data from the data store. If the bean uses BMP, these methods will contain the code required to perform the corresponding operations on the data store. If the bean uses CMP, these methods are merely callback methods that, although present, are usually empty. Instead the deployment descriptor is complemented with sufficient information for the container to undertake the persistence of data on behalf of the component.

Although other parts of EJB specification, like session bean and message-driven bean, are widely successful, a variety of design flaws in the entity bean model hinder its suitability for the representation of persistent data [ROB-03].

Some of these flaws have been addressed in the EJB 2.0 specification (e.g. new local interfaces providing an alternative to the slower remote interface preciously available). However, the semantic differences between local (pass by reference) and remote (pass by value) invocation introduce further issues. Other concerning aspects of the entity beans remain (e.g. the lack of meaningful support for inheritance). Additionally, the persistence and query functions of entity beans must usually be coded by hand (in SQL with JDBC) or described by hand (in Enterprise JavaBean Query Language (EJBQL), which stems from SQL). Finally, the concurrency issues endemic in EJB's threading model [ROB-03], combined with the capability for gross inefficiency when manipulating large data sets, mean that entity beans have regularly failed to meet applications' requirements for object persistence [ROB-03].

Here is a list of some Entity Bean disadvantages [GEN-02]:

- Forces the use of a heavy component mechanism for fine grained business objects.
- More complex, hence, limiting developer productivity.
- More difficult to achieve good performance.
- Inheritance not supported.
- Cannot be used for persistence in non-application server environments.
- There is no dynamic query mechanism to lookup entity beans (finders are specified at compile time).
- It is not easy to write unit tests for beans as it is not possible to use them outside of the application server.
- No support for automatic primary key generation.
- Only relational databases are supported.

1.4.2.2 Java Data Object (JDO)

The Java Data Objects (JDO) specification was developed under the Java Community Process (JCP) as JSR-000012 with Craig Russell from Sun Microsystems as the specification lead. Work started in 1999 and version 1.0 was released in May 2002. JDO provides for transparent persistence for Java objects with an API that is independent of the underlying data store. There are no special interfaces to implement and it is easy to persist plain old Java classes. The query language (JDOQL) uses a Java like syntax so developers only have to know Java. These features provide improved developer productivity and portability across data stores and JDO implementations. The JDO specification supports different deployment environments with a common API. An implementation may support managed (i.e. application server) deployment and unmanaged (i.e. 2 tier) deployment or both. Another implementation might be designed for a small footprint environments such as a cellular phone or PDA. The developer API remains the same in all cases.

Application programmers can use JDO to directly store their Java domain model instances into the data store, without having to use database-specific code.

Here is a list of some benefits that JDO provides:

- **Transparent Persistence**: JDO's greatest advantage is that is allows us to concentrate on developing a good class Model, other than developing a relational class model.
- **Database Independence**: Different type of data sources or JDO implementation can be swapped out in a deployed system. e.g, relational database to a OODBMS or XML file.
- **Ease of use**: it uses only java classes, no other knowledge is needed.
- **High performance**: JDO offers a lightweight solution, if you don't want to take on the weight of EJB.

– **Integration with EJB**: Venders can use JDO to implement Bean Managed Persistence functionality of the EJB container.

On the other hand, JDO does not have a wide industry acceptance till now for the following reasons:

1. JDO is immature, the 1.0 spec just came out. (7.12.2002)
2. Class enhancement: JDO tools enhance class bytecode by adding to it the `PersistenceCapable` interface and any code specific to a particular JDO implementation.

1.5 SUMMARY

Three different types of databases are in use to store objects, each has its benefits and drawbacks, but the most popular are relational databases, since they are the most mature, and well studied and tested. But OO is the norm in programming these days, this leads to a mismatch between the application and the data store in terms of data modeling and representation. The solution is to introduce the so-called Object/Relational Mapping layer between the application and the database. This layer works like an adapter that transforms information between the two heterogeneous systems. Current tools allows to map objects basing on some poor information such as field mapping or Java introspection. Other information are also very important and allow full transparency for the user.

In the following sections we introduce a solution for fully transparent object persistence.

2. OUR SOLUTION

2.1 MAIN REQUIREMENTS

Persistence is one of the most critical issues concerning the development of distributed applications such as Web applications. The mais goals of our requirements is to enhance productivity, performance, and to be vendor independent. Our requirements include – but not limited to:

∃ Object persistency must be transparent: we have to transparently handle the mapping of our object instances to the underlying data store. That includes mapping of inheritance, multiple multiplicity aggregation and composition, both uni- and bi-directional relations between objects. This needs information that are not available in standard JAVA introspection mechanism. This is why we need a special user defined meta data.

∃ Underlying data store transparency: we have to be able to use a number of different data storage paradigms, including (but not limited to) relational databases, file systems and XML documents. The access to these storages must be independent of the storage type and vendor so applications can be ported to any supported data store.

∃ Separating business logic form object persistence. A transaction is open only for synchronization but not during the execution of business logic.

∃ Lazy loading: some times we are not sure which part of a retrieved object we need in order to achieve the business logic. In such a case it is better not to load the entire object but to load its parts progressively when the user access them. This is known as lazy loading and it is very useful to decrease the size of the fetched data and to enhance the overall system performance.

∃ Business domain Object Query Language: we have to supply a Query Language based on business attributes not on storage ones such as primary and foreign keys. The association between objects in DB must be transparent for user in order to decrease the size of request.

∃ We have to handle concurrent access on application level not on storage level.

∃ OOP support: we have to support all OOP features and especially inheritance and polymorphisme.

∃ Should work in non-managed environment, i.e. no application server is required

2.2 SYSTEM DESCRIPTION

The main components of the system are object query language, meta data, persistence layer, data service layers and finally the storage. When the system receives a request from the user using our object query language, it passes this request to the persistence layer. The persistence layer uses Meta data in order to check integrity and business rules and to decompose the composite object into a set of simple ones and to generate a simple request for each simple object. The generated request has no semantics and do not contain any object notion. Then the persistence layer send the set of simple requests to the corresponding data service (xml, DB, file system, ..). the data service layer takes in charge the transformation of these simple requests into a storage related requests. If the initial request needs a response, the data service transform this response into an intermediate format that the persistence layer understand. The persistence layer transforms this intermediate format to a business object and returns this object to the system. The following figure illustrate the process.

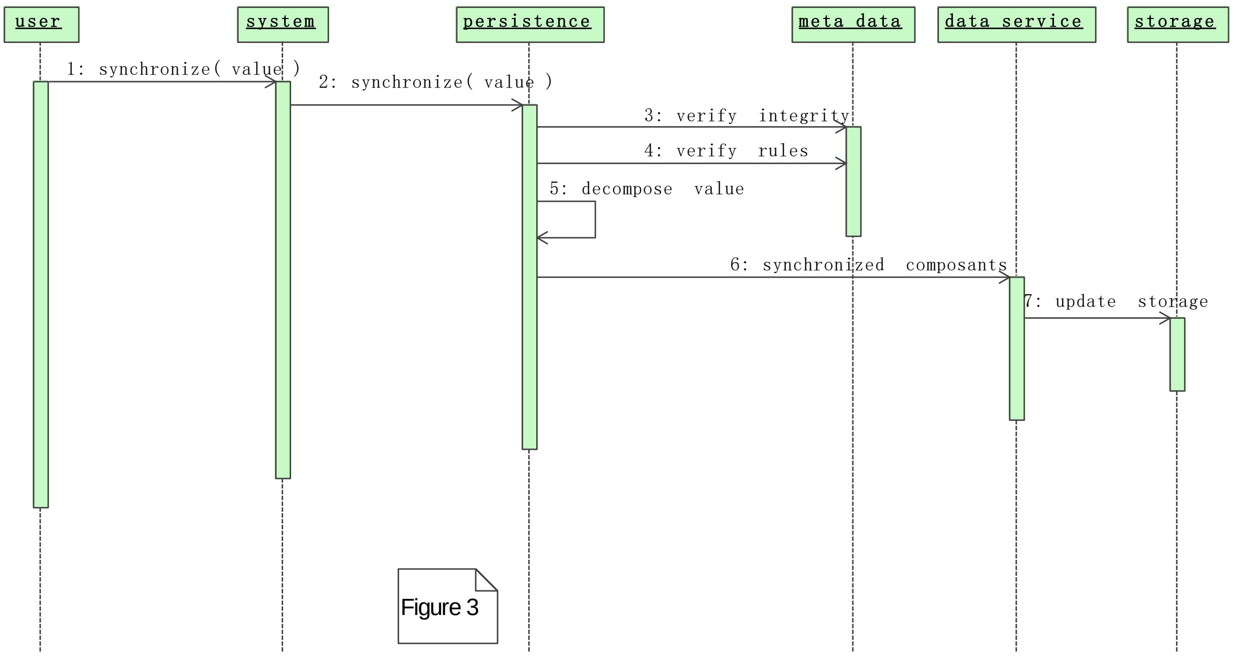


Figure 3

2.3 COMPONENTS DETAILS

Here is a detailed description of the role of each component in our system.

2.3.1 Meta Data

Mapping is usually based on some meta information given by the user such as the correspondence between business domain attributes/their types and fields/types in a given storage. These information may be supplied manually by the user in the case of normal data base or generated basing on JAVA introspection mechanism which the case of JDO. In all cases no semantic is added to the storage and it still the responsibility of the user to specify how to join tables and how to interpret results.

This is why we introduced a more advanced meta data that contains a more precise information that Java introspection does not offer such as:

- relation multiplicity: relation between objects may have different multiplicities such as 1, 0..1, 4, 4..6, n, etc.
- relation style: composition relation denotes a physical relationship between two or more objects (they form actually one object). Aggregation relation denotes a logical relationship between a set of objects such as club, forest, etc. Association relation denotes a logical relationship between two objects such as ownership, friendship, etc.
- syntax criteria: such as mandatory / optional, value domain, size, etc.
- semantic criteria: code/ normal field, some conditions on a set of attributes such as sum(a,b) < 100, etc.
- Collection element types: Java introspection does not specify the real element type of a collection. This is the responsibility of user to manage the type during processing.
- Collection type: 1..n relations may have one of three types: list, set or map.

Our meta data supports also classical information such as:

- field types
- inheritance
- role of each attributes

Basing on some kind of enriched UML diagram (see figure 3), our meta data can be automatically generated.

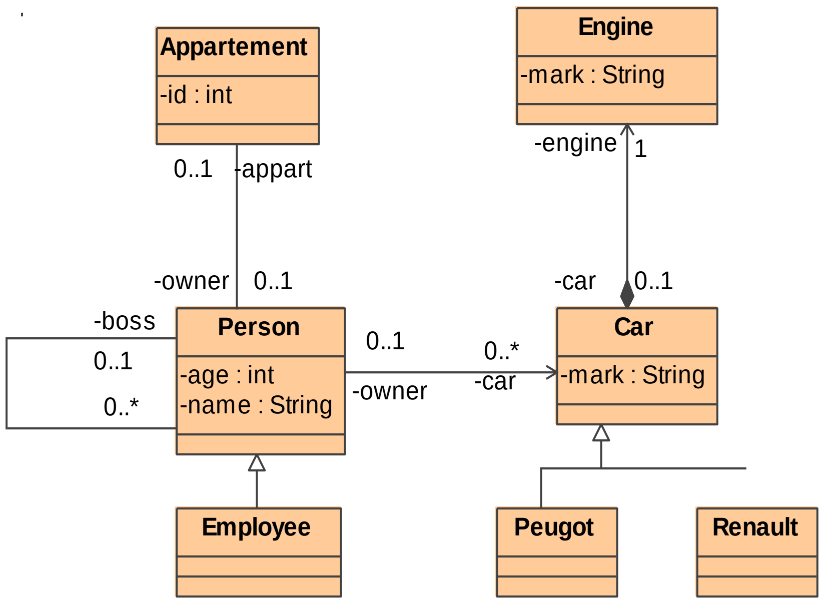


Figure 4 Simple Domain Object Model

These meta data are used by the persistence service in order to map java composite objects into storage structure (tables for example) and the inverse. The storage structure still transparent for the end user.

2.3.2 Persistence Service,

This component play the role of interpreter between business domain objects and the different storages. It verifies the integrity and business rules of a Java business objects during synchronization operation (create/update), divides it into simpler structure such as tables and fields basing on meta data and then passes it to the data service that corresponds to the final storage. During reading operation, this service reconstructs the JAVA object basing on a retrieved structure.

This decomposition takes into account relation between object and maps these relation to the storage. So user does not need to take these relations into accounts. The most complicated part of the mapping is to support inheritance.

There are three approaches that allow us to support inheritance:

1- mapping the inheritance tree into one table

2- mapping an inheritance branche from a tree into one table

3- mapping each class into one table so mapping one object into several tables: this method allows us to support inheritance and to enhance scalability.

This service takes as input a request written in our simple query language (see next section).

2.3.3 Data Service,

This service takes as input the primitive meaningless structure constructed by the persistence service and generates basing on this structure and in a straightforward manner the query relative to the related storage.

The following diagram shows the structure built to achieve the requirements mentioned earlier.

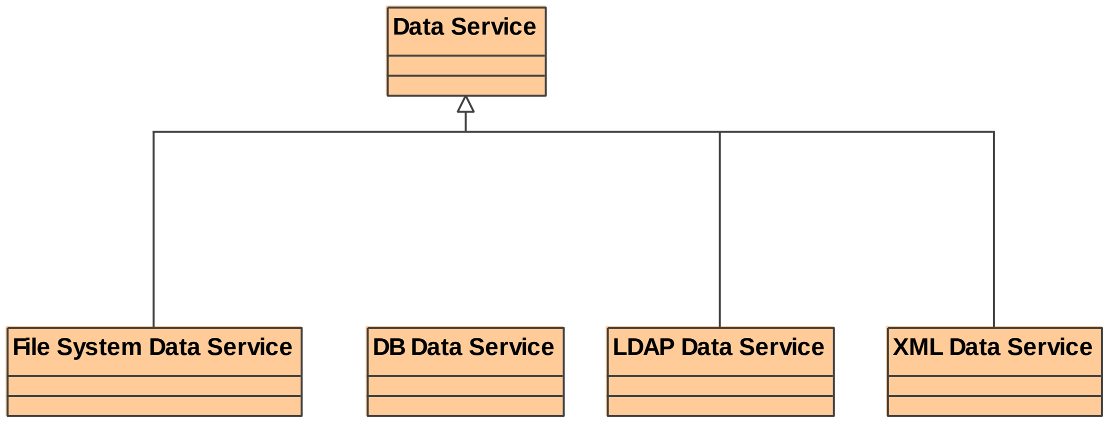


Figure 3 Data Service Structure

2.3.4 Object Query Mechanism

We have developed a proprietary query language that is independent of the storage and that depends only on the business domain objects and vocabulary. We replaced the conjunction by the full role name. Let us take the figure 4 as exemple. The following queries are possible on the object car:

Get mark, engine.mark from Car where owner.age<20

Get owner.name from Car where owner.boss.name=omar & owner.appart.id=120

Such requests correspond to a very long and complicated SQL requests because the conjunction between objects is made automatically at persistence level. The user indicates the conjunction by using the '.' but he does not specify how the conjunction must be realized.

Here is the formal BNF description of our object query language:

```
<Criterion>        ::=   <Criterion> & <Criterion>    |
                         <Criterion> '|' <Criterion> |
                         (<Criterion>)               |
                         <Atomic Criterion>

<Atomic Criterion>::=    <role>          |
                         <UnaryOp> <role> |
                         <role> <Op> <Val>
```

```
<role>                    ::=   <String >| <role> '.' <String>
<Op>                ::= '='|'~'|'<>'|'>'|'<'|'<='|'>='|'#'
<UnaryOp>           ::= '!'
<Val>               ::= <Constant> | <role>
```

2.3.5 Sequencer

It is important when dealing with several storages to handle sequences our self and not to leave this task to the storage itself. This has two advantages: we still independent from the storage itself, and we can deal with two storages at the same times without any conflict between objects ids.

provides auto-incremented counter to be used when serial numbers are required. Our system can manage several sequencers at the same time, for example, there is always a default sequencer that is used to create Object Identifier for our objects, Yet, users can request to create their own business-related sequencers.

2.3.2 Distibuted Transaction Service

Our system should be able to solve the problem of transactions. A transaction is a set of related operations. The system should execute all these operation or non of theme. In such a case we talk about ACID properties which are Atomicity, Coherence, Integrity, and Durability. We use optimistic strategy to solve the concurrent access problem and we are basing on J2EE API in order to support distributed transactions.

2.4 SUMMARY

In this article we exposed the problem of object persistence. We presented different solutions to solve this problem such as OODB, JDO, entity beans, and ORDB. We showed that actually a good solution is to map objects into relational data bases.

We designed a transparent persistence system that solves this problem and enhance user productivity by using some kind of rich meta data. These meta data can be deduced from a UML class diagram.

REFERENCES

[RSE-00] , Database Programming with JDBC and Java , 2nd Edition , O'REILLY , 2000, ISBN: 1-56592-616-1

[PAO-99] , Database Systems; concepts, languages and architectures, Paolo Atzeni, McGraw-Hill, 1999, ISBN: 007-709500-6

[IDC-97] , Object Database vs. Object-Relational Databases, Steve McClure, IDC Bulletin #14821E - August 1997.

[SEB-99] , O/R Mapped Object Persistence Is the Boon, By Sebastian Ware and Mats Helander, Web Site: http://www.15seconds.com/issue/020805.htm.

[STF-97], Database Applications with Objects and Rules, Addison Wesley Longman 1997, ISBN: 0-201-40369-2

[ROB-03], Java Data Object JDO, Robin M. Roos, Addison-Wesley, 2003, ISBN: 0-321-12380-8.

[GEN-02] , JDO Genie Manual , Copyright © 2003 by Hemisphere Technologies (http://www.hemtech.co.za)

[FUS-97], Foundations of Object Relational Mapping, Mark L. Fussel, v0.2, mlg-970703, www.chimu.com

[DOD-97], Object-Oriented Database Management Systems, Gregory McFarland, Andres Rudmik, and David Lange. Air Force Research Laboratory, Department of Defense, 1997.