An Approach for Solving Java™ Object Persistence Issues using RDBMS and other Data Sources

Adomas Svirskas[1,2], Jurgita Sakalauskaite[2]

[1]SAP Systems Integration AG, Business Software Products
D-71691 Freiberg am Neckar, Germany
adomas.svirskas@sap.com
[1] Department of Computer Science, Faculty of Mathematics and Informatics, Vilnius University
LT2600 Vilnius, Lithuania
{adomas.svirskas, jurgita.sakalauskaite}@maf.vu.lt

Abstract. This paper summarizes experience gained designing, developing, reusing and refining Java™ based persistence framework. Authors describe their approach for abstracting operations with data sources under Java interfaces. The paper discusses issues of object-relational mapping issues and presents a solution. The key decisions as connection management, independence from database structure, mass instantiation of persistent objects using object factories and load balancing are discussed. A short review of the related work in this area is provided.

1 Introduction

Perhaps one of the most important issues for many applications is convenient access to external data sources. Historically, the most common data source type is relational database, but there is considerable variety of other types of data sources, which are quite different in terms how data is stored and accessed. There are two main problem areas – the manipulation of the existing data, and the persistence of the run-time objects. These areas are often tightly coupled – tables in relational database are often used to store application entities and, on the other hand, some run-time entities are stored in a different kind of external storage - Lightweight Directory Access Protocol (LDAP) server, for example. Since there is a great deal of interaction with database, both for object persistence and data access, there is a need for the convenient persistence framework, abstract enough to be reusable independently of the particular application logic and the external data source type. This framework should provide interfaces to obtain a connection to a data store, create, retrieve, update and delete persistent entities. Implementation of this interface should also be independent of changes in the database layout.

We have designed persistence frameworks in Java™ and in CORBA™ and implemented them on top of JDBC - relational database middleware of choice for our applications. These frameworks are able to encapsulate data source-dependent implementation under uniform interface. Later we expanded the implementation of

our framework by adding classes for an interaction with LDAP services and we have plans to implement access to the XML-based storage using the same interface.

This paper is organized as follows: Section 2 identifies the main issues and reasons, which caused our framework development effort; Section 3 describes design and implementation in detail; Section 4 is dedicated for brief overview of related work; and Section 5 provides concluding remarks.

Usage of the relational database metadata allowed us to develop implementation, which is independent of RDBMS table structure. The Factory pattern is used to obtain the results of the queries against the databases. The set of application objects is produced based on SQL or other type of (LDAP, for example) queries. There is also a possibility to specify a set of the so-called data sinks for a factory. We can define data sink as an object, which is able to consume and process a single data item from the query result set. Each sink class implements uniform interface and encapsulates the logic of the data item conversion to some alternative representation. Sink is supplied with the data transformation object, which is in charge of the actual further presentation of the data. Thus, an item of the query result set can be passed to the array of sinks and directed to different destinations, if necessary.

Figure 1 shows part of UML class diagram of our persistence framework. We also use very simple connection pooling mechanism, which allows more efficient usage of database connections. Modern application servers provide connection pooling, some JDBC 2.0 drivers (e.g. Oracle) provide connection pooling at drivel level. In these cases, of course, one should use one of the mentioned mechanisms. Our simple pooling technique can be useful in other situations when there is no application server or JDBC driver support.

2 The issues

2.1 High-level functional requirements

It was our opinion that the following issues were crucial to deal with: uniform interfaces as much independent from data store type as possible (JDBC, LDAP and some other interfaces, such as Informix early-times proprietary Java™ Object Interface, were important to deal with); data store connection management – establishment, closing, transactions; persistent object lifecycle – creation, read, update and delete (CRUD) operations; mass creation objects from data stores usage of them or their attributes (e.g. search operations). Load balancing of database operations in distributed object environments emerged as an important task a little later.

2.2 Data source related issues

Despite the fact that the framework has been designed with the independence from data store type in mind, JDBC-compliant data stores captured the most of attention and efforts. While JDBC itself is quite convenient and simple set of interfaces, the actual implementations tend to deviate from the standard by providing proprietary

extensions in some cases or postponing certain important implementations (e.g. metadata) until better times. This led to quite cumbersome task of balancing between temptations to use the proprietary vendor extensions and the complexity of encapsulating little differences into our own abstractions. Particular areas of concern were: metadata (as mentioned earlier) and mappings of SQL data types to JDBC data types, processing of BLOB/CLOB data, unique incremental ID's. It was difficult to isolate our development efforts from vendor-specific decisions because of the fact that JDBC drivers from different vendors were not at the same stage of maturity, we had project-related priorities, etc. Our first JDBC-compliant DBMS was Informix Universal Server, thus many implementation decisions were influenced by this fact. The good thing was that since the beginning of our development we were exposed to the variety of JDBC drivers – we used two different implementations: we used one from Weblogic at first and switched to native Informix driver later. This diversity made us aware of the differences in JDBC URL's and other subtle differences. This laid out a good reusability foundation and, despite considerable bias to Informix side, making the framework Oracle-ready was manageable and proved basic validity of our design decisions.

.

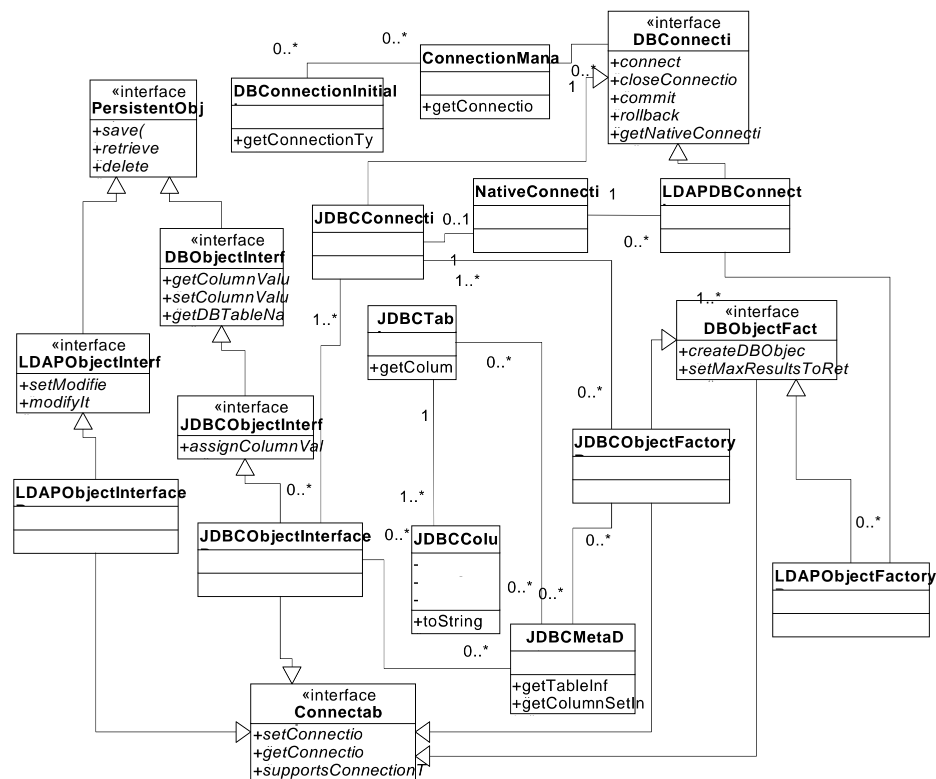

Fig. 1. UML class diagram of object persistence framework described in this paper

On the other hand, the decision to use a uniform set of interfaces for JDBC and non-JDBC data stores proved itself, there were no big problems despite the fact that LDAP, for example, and JDBC are quite different in nature. The big advantage was unified procedure of mass creation of objects from these types of data stores by using the factory pattern. Each type of factories differed form each other in terms of specifying search criteria, but the procedure of obtaining results was identical. LDAP implementation also provided the same support for CRUD operations as the one of JDBC.

3 Proposed solutions

One of the main tasks was to find the right level and granularity of abstractions for our persistence framework. We needed to shield ourselves from tedious task of dealing with underlying storage system details, yet we kept in mind simplicity and efficiency. All our abstractions fall into these main categories: connection management, persistent business objects, object factories. Below we describe our design decisions and explain usage of our framework.

3.1 Connection management

Before doing anything else with a data store, an application must establish a connection to it. This action is quite tedious because it is time consuming, resource intensive and depends on a set of parameters such as network address, port, data store name, etc., which should be specified in storage-type and vendor dependent form (e.g. JDBC URL). That said, it is clear that one would like to encapsulate connection handling into reusable components for the convenience of data store related applications.

We have introduced the *ConnectionManager* class, which is responsible for creating a data store connection. This class has the only method *getConnection*, which creates a connection based on the information contained in its parameter of *DBConnectionInitializer* type. The method mentioned above arranges connection related properties into the right sequence and format depending on the underlying data storage mechanism. For example, JDBC URL's for different types of databases have different format, LDAP does not have URL at all, etc.

The result of an invocation of the *getConnection* method is an instance of the *DBConnection* type. The main decisions to introduce this interface were the need to have uniform interface for different types of connections and the idea to have a connection, which would re-bind itself to the data store in case of connection loss (network-related errors, restart of data store manager, etc.). The interface mentioned above provides methods to establish a connection, close it and begin/end transactions. Of course, it was not our goal to mimic all methods from all possible connection types in our interface in our *DBConnection* interface, so we introduced the *getNativeConnection* method to get access to the underlying data store connection. Applications can use this feature when necessary, however, in this case they are responsible for handling connection-related exceptions.

It is our opinion that our data store connection establishment approach is quite simple and flexible – it is encapsulated in the *ConnectionManager*, which knows how to get obtain the particular connection. The rest of application is shielded from these details. When we introduced our framework, most connections to JDBC data sources were obtained directly via JDBC *DriverManager*. Currently the more common case is to obtain these connections from application server via JNDI, which is completely transparent to clients of our framework. It takes only to introduce a new set of properties for *DBConnectionInitializer* and add the proper handling for the new type of connection.

One more important aspect of connection management is connection pooling, i.e. maintaining a number of open connections to data source, instantiating them on demand, closing after some inactivity period, etc. We have introduced a very simple connection pooling class *DBConnectionPool*, which is capable to maintain a pool of connections for data sources without other pooling mechanisms. Currently data source connection pooling has become a standard feature of most application servers or even JDBC drivers (e.g. latest Oracle JDBC drivers), therefore our pooling mechanism is not used heavily.

The particular implementations of *DBConnection* interface are *JDBCConnection LDAPDBConnection* classes. They slightly differ from each other in operation set - authentication is a separate method in *LDAPDBConnection*, for example, LDAP connection does not support transactions, etc. The general was to place only the most common operations into the generic interface and leave specific features to the particular implementations.

One last connection-related feature of our framework to mention is the *Connectable* interface, which allows setting, obtaining and verifying the type of supported connection. All classes, which implement this interface, can accept *DBConnection* from outside, so they are independent from particular connection and do not need to establish it themselves.

3.2 Persistent objects and operations with these objects

Connection management is very important aspect of any persistence framework, however it is only supporting functionality. The main task is to cope with so called "impedance mismatch" between the database and object-oriented application. While most developers still need to know and understand SQL, use it for many different tasks, the goal is to eliminate routine database operations for typical persistence needs and to provide object-relational mapping between objects and relational database.

First of all, it is necessary to deal with the lifecycle of an object – creation, reading, update, and delete (CRUD) operations. Our approach was to introduce the *PersistentObject* interface, which defines all the operations, mentioned above. This is too generic, however, for real object-relational or other storage mappings, so there are more specific interfaces *DBObjectInterface* and *JDBCObjectInterface*, suited for RDBMS specific mappings.

The main design idea behind the mapping solutions of our framework was to make these mapping as independent from changes in database structures, as reasonably possible. Client applications should be able to access object data via get/set type

methods using column names. This is reasonable, because the client still needs to know the semantics of table structure, i.e. what application-domain meaning has a column. The technical details like conversion from SQL and JDBC types to regular Java™, dealing with ResultSet objects, etc. is up to the framework. Let us discuss in greater detail how this goal is achieved.

We decided to make our persistent objects application-oriented, as explained in the previous paragraph, but database-driven. This means that persistent object is constructed providing database table name and there is mechanism to obtain structure of the table in order to establish set of attributes for the object. Mechanism of obtaining table structure is built on having the *JDBCMetaData* class, which is responsible for providing the metadata (names and types of columns) for persistent objects. Metadata is cached on per-table basis, so there is no need to extract it multiple times. Of course, this makes our framework vulnerable to data structure changes, it is necessary to restart the application if structure of tables changes. Obtaining metadata is somewhat tricky, it was mentioned earlier that some JDBC drivers did not implement this part of the JDBC Standard readily. This leads to the usage of vendor-dependent information such as system tables and data type codes and it is not very elegant solution. Nevertheless, this approach worked for us when dealing with Informix and Oracle, thus, it is feasible to get metadata even using incomplete JDBC implementations.

Another important part besides attribute-like access to data fields is generating of SQL statements for CRUD operations. This kind of functionality has been encapsulated into the default implementation of the *JDBCObjectInterface* – the *JDBCObjectInterfaceBase* class. Normally these statements are used inside the implementation of the latter class, but there is a possibility to get them for troubleshooting and debugging purposes.

The code snippet below shows how easy is to instantiate and save a new object. There is no need at all to worry about SQL statements and keeping them up-to-date when the structure of a table changes. The method *save* of the *PersistentObject* interface automatically results in SQL INSERT statement the first time object is saved and SQL UPDATE statement each time afterwards. It is also possible to delete desired object using *delete* method or revert to the database image using *retrieve* method.

```
Hashtable col_info = dbFactory.getMetaDataInfo("customeragent");

JDBCObjectInterfaceBase dbobj = new

JDBCObjectInterfaceBase("customeragent", col_info, true, false);

dbobj.setConnection(dbFactory.getConnection(), true);

dbobj.setColumnValue("cust_id", new BigDecimal(cid));

dbobj.setColumnValue("agnt_id", new BigDecimal(agid));

dbobj.setColumnValue("approved_tf", agent.isPrivate() ? "F" : "T");
```

```
dbobj.save();
```

An important thing not visible in this code example is unique ID of a persistent object. There are different possibilities to generate this "magic" number. It is possible to use SQL SELECT to get the maximum value of the primary key, increment it and use as the ID of the new object. This approach is not the best, however – it takes additional database access to obtain the maximum value, it depends on the isolation level of an application, etc. Another approach is to use vendor specific data types (e.g. SERIAL of Informix) and leave this function to a DBMS. This approach is even worse, since it leads to vendor lock-in, as these automatic counter types are non-standard. We do not claim that it is always the best practice to avoid vendor extensions, in some cases it is inevitable, but in this case there is more efficient, elegant and portable solution. We took quite common approach to generate unique ID as a function of timer and some random number. This results in 19 byte-length numbers, which contains year, ordinal number of the day, hour-to-millisecond and three-digit random number. It is kept as string in database table. While this is by no means ideal algorithm, it served us well in many projects. There are recommendations to use 16 bytes for this purpose and to have two parts for an ID: so-called factory number and ID number (similar to the approach taken by manufacturers of network interface cards). Skeptics can point out that these approaches waste space, but somebody was very considerate about space years ago, which resulted in Y2K problem. It is our opinion that our approach has a useful feature of keeping track of the time of table entry creation.

Another implementation of persistent objects is based on LDAP services. It is much more simple and straightforward, mainly because LDAP repository is object-oriented itself. Therefore, the *LDAPObjectInterface* and its default implementation, the *LDAPObjectInterfaceBase* do not require complex manipulations with metadata, etc. Identification of an entity is not a problem, since it is open and explicit – every LDAP entry must have its distinguished name – DN. Operations with LDAP-based persistent objects mainly differ from those with JDBC in the way updates are done. In LDAP case there are certain rules how to prepare attribute changes, but these things are quite straightforward. The main benefit of having LDAP implementation of the interfaces mentioned above is the possibility to work using the same abstractions with objects of different nature. This is especially true when mass creation of objects from data store takes place, because instantiation of a single LDAP-based persistent object is quite different from the JDBC:

```
LDAPObjectInterfaceBase principal =
     new LDAPObjectInterfaceBase (new netscape.ldap.LDAPEntry(dn));
principal.setConnection(ldapFactory.getConnection(), true);
netscape.ldap.LDAPModificationSet mods = new
            netscape.ldap.LDAPModificationSet();
```

```
netscape.ldap.LDAPAttribute attr = new
            netscape.ldap.LDAPAttribute( "userpassword", password );
mods.add( netscape.ldap.LDAPModification.REPLACE, attr);
principal.modifyIt (mods);
```

3.3 Object factories and processing modes of persistent objects

It is very common situation when application needs to retrieve series of objects based on some application-dependent criteria. These criteria can be SQL SELECT statements, LDAP queries, XML XPath expressions, etc. For the convenience and simplicity of this kind of operations we introduced quite intuitive object factory interfaces. Implementations of these interfaces take care of performing necessary interactions with underlying data store mechanisms and instantiating persistence-capable objects. Let us examine these interfaces more closely.

DBObjectFactory interface (it extends *Connectable*) is the main interface, which defines factory-type operations of our framework. There are several methods *createDBObjects,* which are used most frequently when there is a need to retrieve data items and produce *DBObjectInterface* type objects. It is possible to specify the maximum number of objects to be created and the number data items to be skipped. The latter feature is useful when it is necessary to deliver results of a query in portions (e.g. multi-page search).

```
Vector productList = dbFactory.createDBObjects
("Select * from product where cust_id = "+cust_ID,
            prodsPerPage,(iPage)*prodsPerPage);
```

The code snippet above illustrates most straightforward use of object factory. A vector of created objects is returned to the client application and it is up to the application to process these objects. This scenario is not always the best, however. Sometimes retrieved data items should be immediately processed, before or in parallel with the retrieval of subsequent data items, which satisfy the retrieval criteria. A good example is an application, which delivers full-text search results via WWW. The end user would be much more delighted if they could see the first results of their search as soon as they are retrieved, rather than wait for the query to end before seeing anything. Another example would be a need to direct retrieved data items to multiple destinations simultaneously. These functions would not be possible using the example above.

In order to facilitate the functionality described above, we have introduced a possibility to attach an arbitrary number of data consumers to the *DBObjectFactory* object dynamically. The *DBFactoryResultSink* interface is capable to consume retrieved data items, as its name suggests. Instances of the classes, which implement

this interface, can be added to the *DBObjectFactory* via *addResultSink* method. It is also possible to delete all or a particular sink from a factory, and specify a set of allowed sink types – classes implementing the *DBFactoryResultSink* interface.

When data item is retrieved by *DBObjectFactory* implementation from data store, it is passed to all registered sinks by invoking their *acceptDataItem* method:

```
public boolean acceptDataItem (DBObjectInterface DBObject,

                Boolean fLastItem) throws DataProcessingException;
```

Thus data sink gets control and can process the object, which is passed. It is up to data sink implementation to define further semantics of processing. Our current implementations of data sinks are synchronous, i.e. object factory waits for the completion of data object processing, but nothing prevents a sink from initiating an asynchronous action and returning control to the factory.

The users of WWW based search/retrieval system were satisfied when the results of their queries started to appear on the browser screen sooner than all results of the query were processed.

3.4 Instantiation of the persistent objects

Having explained the anatomy of *DBObjectFactory* in general, we can take a closer look at the process of creation of these objects. Here is the place where implementations become data store type dependent. RDBMS-related implementation deserves the most attention, since it is most commonly used and more complex. So we in the first place we will discuss the design and implementation solutions of JDBC-based *JDBCObjectFactoryBase* class.

There are two main possibilities to retrieve data items from RDBMS: to specify an arbitrary SQL SELECT statement or to specify a table name and, most likely, a WHERE clause. The latter scenario is the special case of the former, however, we decided to abstract it into separate overloaded method. This decision has been made in order to provide more convenient interface for single-table retrievals and to be able to distinguish between read-only and updateable retrievals. In any case, one of the main goals is independence from DBMS table structure or SQL SELECT statement during object instantiation and retrieval of column values, as it was the case designing persistence support for individual objects.

This independence is achieved again through the usage of metadata (column names and data types) in both cases. In case of arbitrary SQL select statement the metadata is extracted from JDBC ResultSet, in case of single-table retrieve, the metadata is extracted from a database. The metadata information is encapsulated into an instance of *JDBCMetaData* class. Extraction of metadata for the same table is done once and then the metadata is cached in an instance of *JDBCMetaData*. This information is shared between object factories and individual object persistence mechanism. Metadata for each table is stored as a collection of instances of the *JDBCColumn* class. This class provides a possibility to customize processing of individual columns by specifying flags read-only, not null, hidden, etc.

Sometimes it is desirable to have more business-specific object class as a basis for persistent operations. It is possible to do using aggregation, i.e. having *PersistentObject* type object inside the business object, however we provide the *NamedDBObjectFactory* class, which has *setDBObjectClass* method to specify the class of business objects to be created from data items. A class passed as a parameter of the method, mentioned above, must extend *DefaultNamedJDBCObject* class. By using this feature it is possible to have a hierarchy of application-specific persistent objects.

3.5 De-coupled database operations

Another important persistence-related issue is where to concentrate the processing activities of database interactions, since they mostly are resource hungry. Sometimes it is highly desirable to have a possibility to relocate database processing to another physical machine(s) when the workload grows.

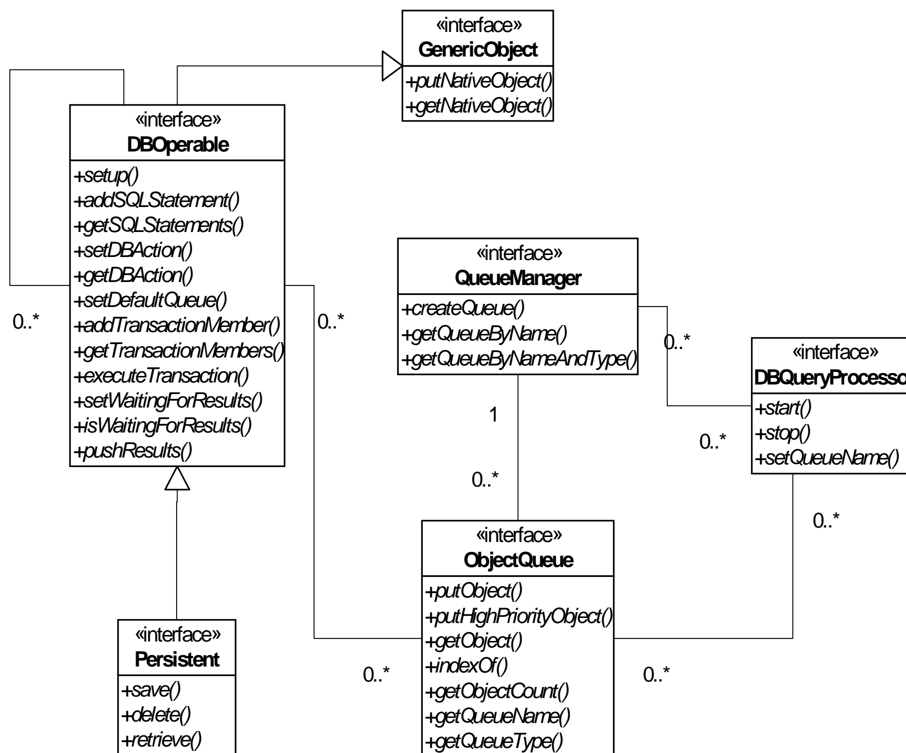

Fig. 2. UML diagram of de-coupled database operations framework components

We have designed the distributed framework based on the producer-consumer pattern, which uses queue-based mechanism to separate the producers and the consumers. The main aim is to encapsulate the processing of all forms of database operations – SELECT, UPDATE, DELETE statements, etc. Two essential parts of the implementation are:

- CORBA object implementing specific interface (*DBOperable*), which contains information needed for executing of the database operations – query or other SQL statement to be executed, what results are to be returned, etc.
- CORBA object (*DBQueryProcessor*), which has the connection to the database and can process objects implementing interface mentioned above.

The main goal is load balancing and the optimal usage of the system processing power – it is possible to plug additional machines running instances of *DBQueryProcessor*, they would connect to Queue Manager, obtain *ObjectQueue* name and would participate in the query or, to be more exact, database operations processing activity. Upon completion of database operation, DB processor returns results to the client component via callback interface.

Object queue solution provides two different communication options:

- Communication via simple data structures – client just puts other objects or itself into a queue, and processor of database operations (*DBQueryProcessor*) gets them from it. Object queue has a possibility to accept objects only of specific type, process objects with different priorities, etc. *DBQueryProcessor* also can choose queue of specific type.
- Communication via encapsulated CORBA Event Service. *DBQueryProcessor* objects register as pull consumers; some objects can register as push suppliers for the specific event channel. One queue can have multiple Event Service channels. The implementation of the queue encapsulates the creation of the channels, etc. and exposes comprehensive interface for the clients.

In both cases, the object to be processed can initiate the database operation. This can also be done by any other object, which would put an object, implementing *DBOperable* into the particular queue.

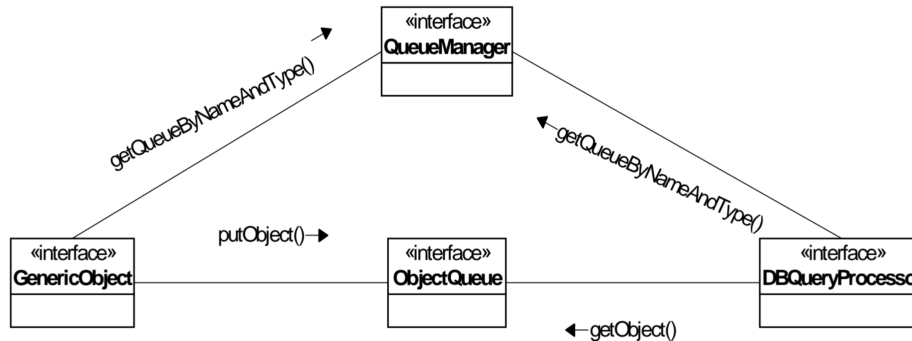

Fig. 3. UML collaboration diagram of communication using queues

4 Related work

There are many different approaches to object persistence and object-relational mapping problems ranging from implementations similar to ours to heavy and expensive products. We are keeping an eye on the developments in this area for two purposes – comparing work of others to our ideas and experiences and very pragmatic need to find something better and more standard for our projects, since we are more involved in development of business-oriented e-commerce systems, than in infrastructure-level frameworks and components. Several years ago it was the common practice to build many Java solutions in-house, however, the more mature Java Platform gets, the more clear is separation of roles in Java developments - do one thing and do it well. Having said that, let us take a look around.

As it was stated earlier, we are interested mostly in frameworks and products, which provide object-relational or object-any-data-source mappings, contrary to the type of products, which take care of object persistence only using serialization or object-oriented databases. The main reason for this is the need to integrate with existing databases, which in most cases are something more conservative than pure-object repositories.

If we are interested in a Java-related solution, it is quite natural to check out the area closest to the Sun, for obvious reasons. A couple of years ago Sun Microsystems developed Java Blend™ persistence framework compliant with ODMG standard for object relational mappings and object databases. Java Blend™ is sophisticated and feature-rich product, which has interactive DB-object mapping tools, pre-processor, supports Object Query Language (OQL), etc. Our framework does not come anywhere near. However, it is our opinion, that Java Blend™ has several drawbacks. The most important one is that Java Blend™ is a product, rather an open specification. Conformance to the ODMG standard does not replace quite successful pattern when Sun Microsystems with the help of area experts leads development of a specification, and industry-leading companies provide implementations afterwards. The second drawback is the price. With all respect to the importance of a top-notch persistence

solution, it is difficult to pay for Java-Oracle mapping middleware more than for Oracle license itself. We could be wrong in license calculation for Java Blend™, but prices starting in the range of $80K are a little bit worrying. And if we remember where ended Sun's home grown Java IDE, we would not be the first to champion Java Blend™.

It seems, however, that Sun Microsystems had had doubts about Java Blend™ too and came up with an alternative. This alternative was named Java™ Data Objects (JDO) and the specification is under development within the Java Community Process (JSR-00012). Its sole objective is to provide Java developers an object interface to data stores. In the same way JDBC provides an industry standard way of accessing data based on SQL, JDO is an industry standard way of accessing data based on Java objects.

JDO approach is similar to our, it provides data access technique, which does not require knowledge of underlying data store interface. JDO goes further by providing query constructs, which are query language neutral. JDO also provides transparent persistence via class enhancer that can process Java bytecode files and create a new one with the necessary enhancements. It depends on JDO implementation whether enhancement takes place at development time or during application run time.

JDO has been designed to work with Enterprise Java Beans, it provides transparent persistence for entity beans, the class developers do not need to provide the persistence support. With EJB session beans, the developer implements beans by explicitly using JDO APIs.

Among areas of concern we could mention the lack of standardization for data store-objects mapping. There are no significant implementations of JDO at this time; Sun Microsystems recently released the reference implementation, which is mandatory part to complete JCP specification effort.

An interesting implementation similar to JDO is Castor project by ExoLab Group. Castor also provides Java-to-LDAP and Java-to-XML mappings.

Other vendors, such as Object Design, Inc. solve object persistence problem using proprietary databases (PSE Pro/Java), which is an appealing approach when it is desirable to separate object persistence repository and business database.

Secant Technologies, Inc. has developed Object Management Group OMG™ Persistent Object Service based Secant Extreme Persistent Object Server for Java™, which bears solid architecture and well designed solutions.

Among the leaders of the Mapping Middleware (the layer between the application server and the database) products is CocoBase product from Thought Inc. The strength of it is in generating of entity beans from templates, freeing developers from tedious tasks of persistence implementation. CocoBase is optimised to work with most of leading application servers.

We believe that the strengths of our solution are simplicity, efficiency, and usage of standard RDBMS. Additional benefit is encapsulation of different data storage types under the same interface.

5 Conclusions

It is our opinion that the time and effort spent developing and maintaining Java™-based object persistence framework were a good investment. While being relatively simple, this framework is powerful and efficient enough to serve everyday needs of database-related Java™ applications. This framework was used in number of projects (eight, to be more exact) and we have plans to rework and refine some parts of it to make it up to date and ready for today applications.

The idea to develop such a framework came up after unsuccessful attempts to find adequate commercial solution a few years ago. Success of the framework in numerous projects served as an inspiration to share our ideas and examine our approach. It is our opinion that despite the fact of much broader commercial and open-source product choice today than few years ago, our framework is still applicable. It could be used for solving bean-managed persistence issues in Enterprise Java Beans (EJB™) applications.

References

1. Ambler, S.W.: Complex Data Relationships: Bet on OODBMS. Software Magazine, January 1995.
2. Ambler, S.W.: Mapping Objects to Relational Databases. Software Development, October 1995.
3. Ambler, S.W.: Object-Relational Mapping. Software Development, October 1996.
4. Ambler, S.W.: Mapping Objects to Relational Databases. URL: http://www.ambysoft.com/mappingObjects.pdf, 1997.
5. Ambler, S.W.: The Design of a Robust Persistence Layer For Relational Databases: An AmbySoft Inc. White Paper. http://www.ambysoft.com/persistenceLayer.html.
6. Atkinson, M.P., Bailey, P., Daynes, L., Printezis, T., Spence, S.: The Design of a new Persistent Object Store for Pjama. The Second International Workshop on Persistence and Java(tm) (PJW2), Half Moon Bay, California, August 1997.
7. Atkinson, M.P., Daynes, L., Jordan, M.J., Printezis, T., Spence, S.: An Orthogonally Persistent Java. ACM Sigmod Record, Volume 25, Number 4, December 1996.
8. Atkinson, M.P., Daynes, L., Jordan, M.J., Printezis, T., Spence, S.: Design Issues for Persistent Java: a type-safe, object-oriented, orthogonally persistent system. Seventh International Workshop on Persistent Object Systems (POS7) Cape May, New Jersey, May 1996.
9. Brown, K., Whitenack, B.: Crossing Chasms: A Pattern Language for Object-RDBMS Integration. Pattern Languages of Program Design 2, John M. Vlissides, J.M., Coplien, J.O., Kerth, N.L., eds., Addison-Wesley, Reading, MA., 1996.
10. Close-up on JDO: a standard for persistence of Java business objects. TechMetrix Research, February 2001, http://www.techmetrix.com/trendmakers/tmk0201/tmk0201-3.php3.
11. Castor JDO: http://castor.exolab.org/jdo.html.
12. CocoBase: http://www.thoughtinc.com/cber_index.html.

13. Cocobase™ Enterprise O/R Business Benefits Whitepaper.
14. Dolgicer, M.: CORBA and Java: Marriage or just serious dating? Application Development Trends magazine , January 1999.
15. Java Data Objects Specification. http://access1.sun.com/jdo.
16. Jordan, D.: An overview of Sun's Java Data Objects specification. JavaReport. June (2000).
17. Malani,P.: Connection Strategies in EntityBeans. JavaReport, April 2001.
18. Svirskas A., Sakalauskaite S.: Development of Distributed Systems with Java™ and CORBA™ Issues and Solutions. DB&IS 2000, Vilnius.
19. The Object Data Standard: ODMG 3.0. http://www.odmg.org.
20. The Java Blend White Paper. http://www.sun.com/software/javablend/whitepapers/index.htm.
21. Yoder, J.W., Johnson, R.E., Wilson, Q.D. : Connecting Business Objects to Relational Databases. URL: http://www.joeyoder.com/Research/objectmappings/Persista.pdf.