

# Mapping Java objects to relational databases with MPF/J

Ho Ngoc Duc<sup>1</sup> and Hendrik Höfer<sup>2</sup>

<sup>1</sup> Institut für Informationssysteme, Universität zu Lübeck,  
Osterweide 8, D-23562 Lübeck, Germany  
duc@ifis.uni-luebeck.de

<sup>2</sup> MicroDoc Computersysteme GmbH,  
Karlstraße 42, D-80333 München, Germany  
Hendrik\_Hoefler@microdoc.de

**Abstract.** We present the MicroDoc Persistence Frameworks for Java (MPF/J), an Object-Relational (O/R) mapping suite developed for overcoming the "impedance mismatch" between the object world and the world of relational databases. We discuss some issues that arise during the development of the frameworks that we consider worth investigating further. We also report on our experiences in applying the frameworks in large-scale applications.

## 1 Introduction

The present paper describes the MicroDoc Persistence Frameworks for Java (MPF/J), an Object-Relational (O/R) mapping suite developed for overcoming the "impedance mismatch" between the object world and the world of relational databases. We report on our experiences with developing MPF/J and applying it in different scenarios.

In the next section we give an overview of MPF/J. We present briefly the overall architecture of the frameworks, describe their main components and show how MPF/J are integrated within EJB containers. Then we discuss some issues that arise during the development of the frameworks that we consider worth investigating further. Then we shall compare MPF/J with other work in the area of Java persistence and with the JDO model.

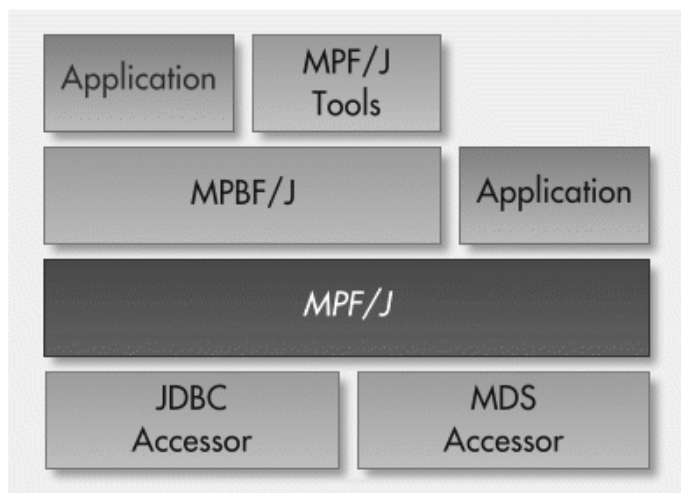
## 2 An Overview of MPF/J

MicroDoc has been providing O/R mapping tools since 1993 when it developed a persistence framework for Smalltalk. The persistence frameworks for Java MPF/J are available since 1999.

## 2.1 Architecture

The MicroDoc Persistence Framework Suite follows a layered approach. It consists of two frameworks: the low level MicroDoc Persistence Framework for Java (MPF/J) and the high level MicroDoc Persistence Behavior Framework for Java (MPBF/J). Additionally there are a set of tools, e.g. for automatic generation of mapping information from object models or to create classes and mappings from database structures. In the following we shall use MPF/J to refer to the whole suite unless specifically stated.

MPF/J does not require persistent classes to extend a certain MPF/J class or to implement a given interface. Therefore, the application can implement its business logic independently from persistence. Moreover, applications do not reference MPF/J classes directly but operate on interfaces whose implementations are created by factory objects. It is hence possible for applications to provide own implementations of MPF/J interfaces and to influence the behavior of MPF/J significantly.



© 2000 MicroDoc Computersysteme GmbH

**Fig. 1.** MPF/J Architecture

MPF/J lets the developers select one of 3 programming models: type-safe mode, generic mode, or instrumented mode [8]. They differ in the number of generated helper classes and the degree of intrusion (i.e., the amount of persistence-specific code and the need to refer to MPF/J classes and interfaces).

## 2.2 The low level framework MPF/J

The basic framework allows for mapping classes and properties to tables and columns. It offers a great flexibility of mapping schemes. The advantages and disadvantages of

the various patterns for mapping aggregations, associations, and inheritance with respect to performance, flexibility, maintenance cost etc. have been discussed extensively in the literature (e.g., [5, 3]). MPF/J takes a flexible approach and lets the designer choose the mapping appropriate for the application. The mapping options offered include: map field (property) to one or more columns, map 1:1, 1:N, M:N relationships, map whole inheritance tree to a table, map a whole inheritance path to a table, map each persistent class to a table etc.

For reading objects the basic framework MPF/J offers the API for applications to read all objects of a class, optionally up to a given limit, to read objects via streams, to read relationships of an object via a foreign/primary key, and to read objects via a query. Two principal types of queries are supported: queries defined by SQL fragments and EJB-QL queries, the latter generalized to non-bean classes.

Persistence frameworks are typically used to map static classes to database structures. However, certain complex applications have to deal with persistent objects that are not known at development time. Those objects must be implemented generically (e.g. using Maps), their structures are only known at runtime. MPF/J offers the possibility to map those generic structures by using types (IType) rather than classes. In the standard case, classes are an implementation of a type, so "normal" applications do not have a care about it. However an application can provide its type system if generic structures are necessary.

Many applications need to map a single class in different ways, e.g., to map it to different databases. A company operating in Germany and Austria may want to process all orders centrally by a single application but store order information in separate databases for the two countries. This is also supported by MPF/J.

Experience shows that in various cases it is useful to have an artificial key which has no business meaning. E.g., they enable very efficient handling of reading objects with inheritance. MPF/J offers a uniform way to generate such OIDs by means of sequence numbers. If the underlying database systems supports sequences (Oracle or DB2 7.2), native sequences are used. Otherwise such a behavior is simulated by the framework.

With the low level framework MPF/J, applications must read and write objects explicitly via MPF/J API. More sophisticated features like read on demand, cache support, change tracking, or integration into EJB container transactions are supported by MPBF/J, the High Level framework.

### **2.3 MPF/J High Level framework (MPBF)**

The high level framework MPBF extends the low level framework and offers features to further isolate the application from the underlying persistence mechanism. The main goal is to be as transparent and as fast as possible. With MPBF, application developers will not code to MPF/J APIs most of the time, but manipulate objects within so-called Object transactions. All (implicit) changes, create- and remove operation are collected in an Object transaction. Upon commit the changes are synchronized with the database. There exist various configuration options (use cache,

broadcasts, VM wide locking) for Object transactions. Also, two objects with the same primary key are identical in an Object transaction. In unmanaged environments (fat client, servlet containers), applications need to define transaction boundaries. This is not necessary in EJB servers since MPF/J Object transactions are hooked to container transactions.

Additional features supported by the high-level framework include transparent object reading on demand, change tracking (i.e., MPBF/J will track changes done to persistent objects and automatically update the affected rows upon commit or EJB container callback), object lifecycle maintenance (i.e., MPBF/J will track created and removed objects and automatically issues the corresponding database commands upon commit or EJB container callback), linearization (correct order of the modifying SQL statements). Various cache strategies are supported (Weak, Strong, LRU, etc.).

## **2.4 MPF/J Tools**

The MicroDoc Persistence Frameworks Suite offers a (graphical) development environment for mapping object structures to database tables. The mapper allows to generate Java classes and the necessary mapping information as well as the DDL script for creating the tables. It also offers the possibility to inspect the structure of existing classes and create mappings for them, or to read database structures and create Java classes that map to them. The built-in Java parser allows for smart incremental code generation, i.e., when generating code the mapper tool leaves source formatting and user code unaffected.

In addition to the graphical mode, the functionality of the mapping tool can also be accessed programmatically via Tool-API. They are used for integration into modeling tool or for building scripts. The mapping data are stored as XML, so they can be exchanged easily. It is even possible to use the XML mapping file generated by the mapper to generate the object model and the mapping information in another OO language. This has been done for Smalltalk.

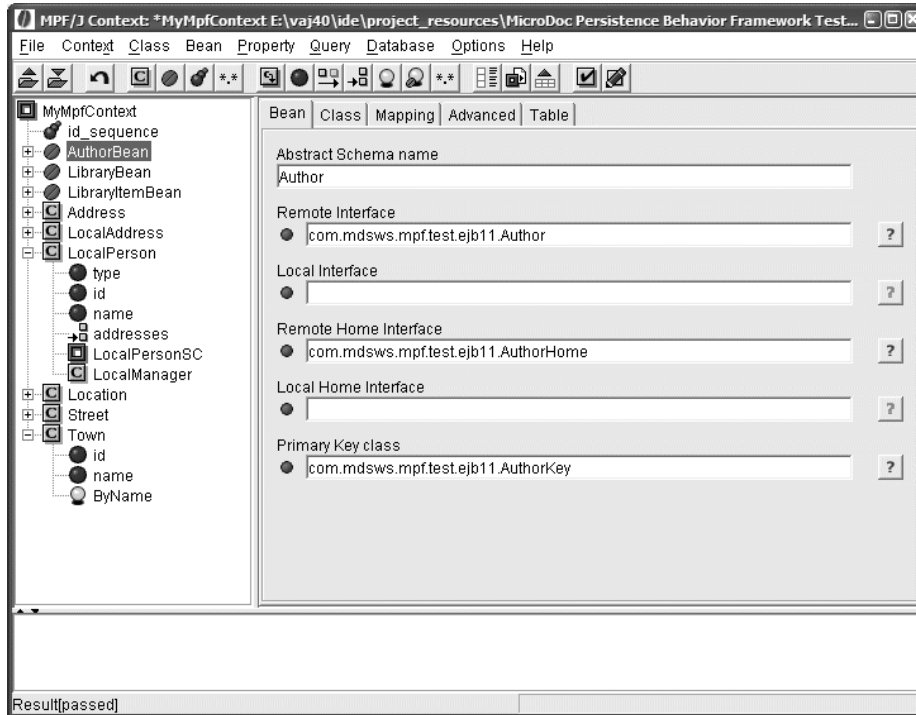


Fig. 2. MPF/J mapper tool

## 2.5 MPF/J in EJB containers

MPF/J is the O/R mapping framework used for CMP (Container-Managed Persistence) enterprise beans in Fujitsu Siemens Bean Transaction Server (BeanTA). When CMP beans are to be deployed in BeanTA, the deployer uses the MPF/J mapping tool to map the beans to the desired data sources. The tool then generates the code that is executed at runtime to manage the beans' persistence.

In addition to CMP support in BeanTA, MPF/J can be used to support BMP (Bean-Managed Persistence) in any EJB containers. For the bean developer the persistence layer is almost as transparent as CMP: he does not have to write any SQL to access the database. For example, instead of coding statements to save some data to the database the developer can implement the method `ejbCreate` of a bean by simply calling the method `mpfBeanCreate` of a helper class provided by the framework. Alternatively he can use the MPF/J tools to generate the Beans, Remote interfaces, Home interfaces etc. and does not have to write any code at all.

## **2.6 MPF/J in real-world applications**

MPF/J is being used to realize the persistence layer in a number of projects of various sizes, with different architectures. In this section we shall report on how MPF/J performs in a large-scale project.

The project aims at developing a Call Center Application to help a company to manage their job brokering process. The customer, Training and Consulting International AG (TCI), contracts with the labor department of the German government to provide training and job placement for professionals seeking new jobs. For that purpose, TCI needs to store and process data of more than 1 million employers and hundred thousands of job-seekers, together with their qualifications and their job placement notifications in various publications. When an employer is interested in a certain candidate, its representative may call the Call Center and ask for an interview or another contact form. A candidate may also call to inquire about his applications. It is projected that up to 50 agents may use the system concurrently to handle up to 30,000 calls per day.

The Call Center Application was created as an Intranet/Extranet application based on Servlets and JSPs running in IBM WebSphere. MPF/J is used for the persistence layer, with DB2 as the relational database backend. The thin clients (Web browsers) have almost no business logic at all. When a phone call comes in, the caller's number is transmitted from the ACD system (automated call dispatch) to the call center. An MPF/J query will be run against the database to identify the caller based on his phone number, and information about the caller are put together. This typically requires navigating an object net consisting of 5-15 objects. For example, if the caller is a job-seeker, the system will collect his core data, his qualifications, his pending placement attempts etc. and send the data to the browser. As a call center agent is picking up the phone, the information about the caller appears on his screen within less than a second, so the agent is able to cope with the enquiry very efficiently.

## **3 Discussion**

In this section we shall discuss some issues that arise in the context of O/R mapping that seem relevant for a larger class of applications.

### **3.1 Persistence layer in EJB containers**

Enterprise Java Beans (EJB) is a specification of a consistent component architecture framework for creating distributed n-tier middleware. It is meant for creating server-side, scalable, transactional, multi-user, secure, enterprise-level applications. The technology is still far from mature, so there are unsurprisingly many shortcomings that need to be addressed. We shall discuss one issue inherent in the EJB technology.

Inheritance and polymorphism are among the most powerful features of object-oriented technology. Unfortunately, they are not well supported by the EJB model.

Inheritance is only supported in a limited form. An entity bean implementation can be subclass of another bean, and its remote interface extends the remote interface corresponding to the superclass. However, such a relationship cannot exist between the home interfaces due to conflicting method signatures. This leads to a severe problem that can be illustrated with the following example. There are two beans `LibraryItem` and `Book`. The class `BookBean` is implemented as subclass of `LibraryItemBean`, and the remote interface `Book` extends the interface `LibraryItem`. The class `LibraryItemBean` implements the method `getName()` which returns the String `"LibraryItem"`. The class `BookBean` overrides that method to return the String `"Book"`. Now, a client (e.g., a session bean, another entity bean, or an EJB client) which looks for a `LibraryItem` by means of a finder method will always get a `LibraryItem` object, even if the object matching the finder criteria is actually a `Book`. If the method `getName()` is called on the found object, supposed a `Book`, the return value will be `"LibraryItem"` and not `"Book"` as one may expect. This contradicts OO thinking directly. Moreover, this drawback makes implementing bean relationships cumbersome.

To help overcoming this problem, the MPF/J integration in EJB containers offers a simple API for finding beans of the correct classes. That API can be used within the container by client programs, e.g., for finding `LibraryItem` beans, and the returned objects will then be of the right classes, so overridden methods are called correctly. With the help of that API, complex relationships between beans can be realized more easily, and those relationships are managed correctly.

### 3.2 Historization

Many applications need to track changes that evolve around an object over time or to reconstruct the state of an object or a whole net of objects at a given time point. For that purpose one needs to "historize" the involved objects, i.e., to keep object histories. Over time many methods have been developed for that purpose, typically based on validity intervals or on version numbers [4].

Object versioning is a high-level service that is considered desirable by the Object-Oriented Database System Manifesto [1]. It is supported by many object-oriented database management systems, e.g. O2 [9] or ORION [2, 7]. However, this service is not provided by relational DBMSs. Hence applications that use a relational database as their data store system usually have to integrate it within their business logic. The obvious disadvantage of this approach is that persistence and business logic are mixed. Let us elaborate further on that point.

When an application needs to manage object histories, a common strategy is to use one-dimensional (1D) historization. This strategy is realized by adding to the persistent classes two timestamps: `validFrom` and `validTo`, which define the objects' validity intervals. When the application sets its current time point to `T`, it should "see" all objects which are valid at `T`, i.e., objects with `VALID_FROM <= T < VALID_TO`. If the historization service is not supported by the database, the application has to take care of the following points:

- To read an object means reading the version of the object that is valid at T
- Insert a new object means inserting it to the database and marks it to be valid from T to INFINITY
- To delete an object means setting its validTo property to T
- To update an object means deleting and inserting it according to the two previous rules

Thus, the application must contain database code to manipulate correct versions of the objects. Managing object versions this way does not separate business logic cleanly from persistence.

From our point of view, historization is a high-level service that belongs to the persistence layer and not to the application logic. MPF/J offers a general abstraction for implementing different historization strategies independently of the application. The 1D strategy described above is implemented by the framework, and a programming interface is provided for applications to provide their custom strategies if necessary.

### **3.3 Related work**

Many commercial O/R tools for mapping Java objects to databases are currently available. Some of the better known ones are TopLink from Oracle, CocoBase by Thought, Inc., and the Visual Business Sight Framework (VBSF) by ObjectMatter. MPF/J offers the same core set of functionalities as these products: the possibility to map classes and relationships in different ways, graphical tools for doing the mapping, a simple API to querying and manipulating objects, support for object identity, caching, change tracking, etc. As far as we know, none of the other O/R mapping frameworks offers a historization service. Moreover, MPF/J differs from many frameworks in its high degree of support for EJB integration.

Sun JDO (Java Data Objects) is a recent specification for transparent persistence. It is meant to provide a unified, standard persistence interface supported by multiple vendors delivering competing implementations. One programming model of MPF/J (the instrumented mode) works similarly to JDO, with the following differences. First, MPF/J is less intrusive. Unlike JDO, it does not require the persistent classes to implement a PersistenceCapable interface. MPF/J can work with instrumented and "normal" classes. The converse is also true: the instrumented classes can work with or without MPF/J, i.e., the persistence code can be configured to do nothing if MPF/J is not used. Second, MPF/J seems simpler and more convenient to learn and to use. The query languages supported by MPF/J (SQL and EJB-QL) are more established than the query language that JDO introduces. MPF/J is more general in some aspects: while JDO instruments classes, MPF/J can work with generic implementations (e.g., Maps). MPF/J also offers more ways to manipulate the persistent properties: they can be accessed via field or method access.

## 4 Conclusion

We have described MPF/J, a set of frameworks and tools for mapping Java objects to relational databases and discussed some problems that arise in the context of O/R mappings in EJB containers. Furthermore we argued that historization is a high-level service that belongs to the persistence layer and showed how this issue is addressed by MPF/J.

## References

1. M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik, The Object-Oriented Database System Manifesto, in: W. Kim, J.-M. Nicolas, S. Nishio (eds.), Proceedings of The First International Conference on Deductive and Object-Oriented Databases, Kyoto (December, 1989), pp. 40-57.
2. Jay Banerjee, Hong-Tai Chou, Jorge F. Garza, Won Kim, Darrell Woelk, Nat Ballou, Hyoung-Joo Kim: Data Model Issues for Object-Oriented Applications. TOIS 5(1): 3-26 (1987)
3. Kyle Brown, Bruce G. Whitenack: Crossing Chasms, A Pattern Language for Object-RDBMS Integration, in: John M. Vlissides, James O. Coplien, and Norman L. Kerth (Eds.): Pattern Languages of Program Design 2, Addison-Wesley 1996.
4. Randy H. Katz, Towards a Unified Framework for Version Modeling, ACM Computing Surveys, V 22, N 4, (December 1990), pp. 375 - 408
5. Wolfgang Keller: Mapping Objects to Tables: A Pattern Language, in: Proceedings of the 1997 European Pattern Languages of Programming Conference, Irrsee, Germany, 1997.
6. Wolfgang Keller, Jens Coldewey: Accessing Relational Databases: A Pattern Language, in: Robert Martin, Dirk Riehle, Frank Buschmann (Eds.): Pattern Languages of Program Design 3. Addison-Wesley 1998.
7. Won Kim, Nat Ballou, Hong-Tai Chou, Jorge F. Garza, Darrell Woelk: Features of the ORION Object-Oriented Database System. In: Won Kim, Frederick H. Lochovsky (eds.): Object-Oriented Concepts, Databases, and Applications. ACM Press and Addison-Wesley 1989, pp. 251-282
8. MicroDoc Computersysteme: MPF/J User Manual, version 4.1, Munich, 2002
9. O2 Technologies: Version Management Reference Manual, version 4.6, Versailles Cedex, 1996