

# Making Applications Persistent at Run-time\*

Angela Nicoara    Gustavo Alonso  
Department of Computer Science  
ETH Zurich, 8092 Zurich, Switzerland  
{anicoara, alonso}@inf.ethz.ch

## Abstract

*Persistence is a common requirement in many applications. In existing systems, persistence is added to an application at either compile or deployment time by using a variety of mechanisms. In this paper we extend the notion of orthogonal persistence to make it dynamic: persistence becomes not only an orthogonal concern but one that can be added to an application at run-time without interrupting its operations.*

## 1 Introduction

Persistence is a common application requirement whereby parts of the state of an application are stored in non-volatile memory to maintain state and facilitate recovery. The key to application persistence is determining who is responsible for making the state of a program persistent. Initially, orthogonal persistence was implemented as language extensions. An example is PJama (Persistent Java) [1], a persistent programming environment based on a modified JVM. Another example is ObjectStore [6], a persistent storage engine offering an API to endow Java and C++ programs with persistence functionality.

Today, persistence is mostly implemented as a property of the *container* where the application runs (e.g., J2EE [15] or CORBA [7]). Persistence is added to an application at deployment time by using configuration information that tells the container what needs to be made persistent.

Several research efforts have considered persistence and related concerns. To our knowledge, all of these approaches consider persistence only as a development [8, 3, 1], deployment [15] or compile time property [11, 13, 12, 14].

Object-relational mappers (ORMs) handle the impedance mismatch between the object model of the application and the relational schema of the database. ORMs

are typically implemented using Plain Old Java Objects (POJO) persistence solutions; e.g., Oracle TopLink [8], JBoss Hibernate [3].

Persistent Java (PJama) [1] is a distributed object platform that provides orthogonal persistence [1, 2] for Java by using transaction classes. It is based on the Sun JDK platform, where the JVM is modified to enable persistence using an integrated persistent object store. The application data can be stored persistently in two ways: either PJama fully manages the persistence aspects, or this task is left to the application programmer itself. PJama offers persistence by reachability where any object reachable from a persistent root object is made persistent.

In this paper we introduce the notion of *dynamic* persistence. That is, instead of persistence being a deployment time property, we turn it into a run-time property. With dynamic persistence we aim at making the application independent of the container, so that it can move from container to container and be made persistent at run-time without having to stop or redeploy the code.

In the paper, we describe dynamic persistence in detail. In the experiments we demonstrate that our solution is independent of the mapper used (we show results for Oracle's TopLink [8] and Hibernate [3]). We also show that our approach has little overhead.

## 2 Dynamic Persistence

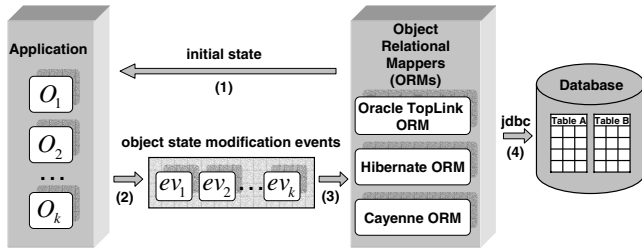
### 2.1 Basic Architecture of Orthogonal Persistence

Implementing orthogonal persistence involves two main components:

- an *object-relational mapper (ORM)*, which is interposed between the application and a relational DBMS; it converts between the object model of the application and the relational schema of the database,
- a *connector functionality* connecting the application to the ORM.

A simplified architecture of a system for object persistence is presented in Figure 1. A possible scenario may involve an application working with several objects,  $O_1 \dots O_k$ ,

\*The work presented in this paper was supported (in part) by the National Competence Center in Research on Mobile Information and Communication Systems (NCCR-MICS), a center supported by the Swiss National Science Foundation under grant number 5005-67322.



**Figure 1. Architecture for object persistence**

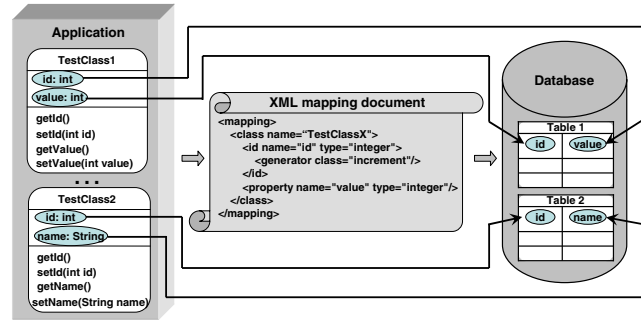
that can be instances of the same class or different classes. These objects were either created by the application itself or have been previously retrieved from the persistent repository (*step 1*). During run-time, the application will either read or modify the states of a subset of  $O_1 \dots O_k$  objects. Whenever the state of an object is changed, it will trigger a state-modification event. As a result, a stream of state-modification events  $ev_1 \dots ev_k$  is generated during the lifetime of the application (*step 2*). State modification events can be coarse-grained (e.g., object  $O_i$  was changed) or fine-grained (e.g., field  $f$  belonging to object  $O_i$  was modified).

The state modification events are consumed by object-relational mappers (e.g., Oracle TopLink, Hibernate) (*step 3*) which transform these events into database statements (inserts/updates) that are propagated to a DBMS system (*step 4*). In addition to mapping objects to relations, ORMs usually provide session and transaction management as well as a wide range of mapping algorithms the user can choose from.

The objects to be persisted are defined in XML mapping documents, which are used to describe how objects and their relationships are mapped to the tables and the relationships between them in a database (Figure 2). An ORM maps objects to database tables and views, and provides transactional operations, queries and stored procedure calls as methods to these objects. It provides an API which contains methods for saving (e.g., 'save(Object)'), updating (e.g., 'update(Object)', 'saveOrUpdate(Object)'), or deleting (e.g., 'delete(Object)') objects from database. It also provides methods for querying the state of the database that maintains the persistent representation of objects. ORM allows developers to manipulate sessions. A session ensures that all modifications performed by a thread will be propagated to the database in a consistent way.

## 2.2 Dynamic Changes to an Application

Aspect-Oriented Programming (AOP) [4] is a technique that allows the separation of concerns in software development, making it possible to modularize crosscutting aspects of a system. It is used to express modular and orthogonal functionality in software components. The orthogonal functionality affects the source code in potentially many different places and, hence, it is advantageous to express them as aspects that are *weaved* into the source code but are concep-



**Figure 2. Mapping application objects to database tables**

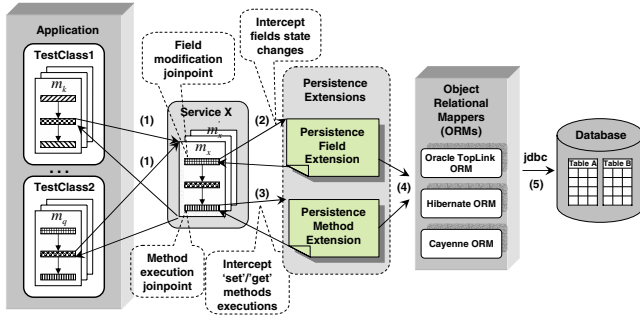
tually separated from it. The general mechanism in AOP is to describe the functionality to be added as *extensions (aspects)*. An aspect defines a collection of points in the execution of a program and what to do (e.g., connect instances of classes to a database) when these points are reached in the code (e.g., when a field has been modified, when invoking methods with certain signatures, etc.). The execution points are called *join-points* and the action to be executed at the join-points is called the *advice*.

Dynamic AOP (d-AOP) extends the original notion of AOP by allowing weaving at run-time. We use dynamic AOP to dynamically modify the code of a running application so that the state can be made persistent. In our approach, we use PROSE [5, 10, 9] as the platform for dynamic AOP. PROSE hooks into the JVM and intercepts operations at the points in the execution of a program where aspects are to be executed. In PROSE these aspects are known as *run-time extensions*.

## 2.3 Run-time Adaptation Based on Aspect-Oriented Programming

Consider an aspect that intercepts accesses to objects instance fields, which make up the state of an application. The object's instance fields are accessible in the aspect either by using the 'set'/'get' methods or by directly accessing instance fields. During run-time, every time the accesses to objects' fields are intercepted, the corresponding aspect code is executed. The aspect code takes the current field, extracts its value and writes it into persistent store. Dynamic persistence is thus implemented through the run-time insertion of code extensions that capture changes to state (e.g., field modifications) and store those changes into a database using an ORM.

In what follows, we describe in detail how such persistence aspects work (Figure 3). Consider a mobile application that arrives at a new location. Let's assume it first performs some operations and then invokes a method  $m_x$  on service  $X$ . Furthermore, the computation in  $m_x$  changes the state of service  $X$ . After  $m_x$  completes, the results are transferred back to the application. The application can then



**Figure 3. Adapting a running application with persistence functionality**

invoke an other call to  $m'_x$ , carried out in a similar manner. Suppose now that the mobile device has acquired extensions that implement dynamic persistence (Figure 3). When the method  $m_x$  is invoked (*step 1*), a state change occurs during the execution of  $m_x$  (e.g., field modifications (*step 2*) or 'set/'get' method executions (*step 3*)), and an interception takes place. The persistence extension intercepts the state change and notifies the ORM (*step 4*), which forwards the changes to the database (*step 5*). The relevant join-points of interest for capturing the state changes and persist the application state are field interceptions and method execution join-points. The first join-point type allows capturing object instance fields modifications. Using the second type, the state changes can be intercepted during the methods' executions. An example of an extension showing how a field modification join-point can be used is presented in the next section.

It is important to notice that at any time the extensions can be exchanged for new ones that specify a more sophisticated persistence behavior. The amount of persistent data can be restricted by giving a detailed definition of the relevant data set. The aspect then restricts its interception so that the aspect code gets applied to the relevant data only. Our approach allows fields to be matched by means of regular expressions (e.g., all fields whose name matches the regular expression '*field.\**' and belong to classes whose name starts with *Service* ('*Service.\**') are made persistent). We use this feature to provide powerful pattern-matching rules for persistence specification.

An important characteristic of this dynamic persistence mechanism is the ability to adjust the level of synchronization between run-time data and persistent copies. By defining aspects that specify update requests rather than directly writing the data into persistent store, additional aspects have the possibility to specify checkpoints, i.e., the moment when they actually update the persistent data. This way, the synchronization level varies between immediate updates and lazy database updates. For example, an aspect responsible for executing the requests may intercept the end of all methods requesting updates and provide an aspect code that updates the persistence store. This strategy

leads to per-method synchronization of persistence data. It is a common trade-off between update frequency and performance: the higher the update frequency, the worse the performance and vice versa.

## 2.4 Persistence as an Aspect

Figure 4 presents an example of an aspect which is used to connect all instances of classes whose name starts with 'Service' to a database. Aspects are first-class Java entities, and all related constructs are expressed using the Java language.

```

1 public class PersistenceAspect extends DefaultAspect {
2   public SessionFactory sf;
3   public Crosscut makePersistent = new SetCut {
4     // advice method
5     public void SET_ARGS(Object o, Integer f) {
6       Session ses = sf.openSession(); // session handling
7       ses.saveOrUpdate(o);           // object writes or updates
8       ses.flush();
9       ses.connection().commit();    // commit the changes into the database
10      ses.close();                   // session handling
11    }
12    // specialization
13    PointCutter pointCutter() {
14      return ( (Fields.declaredInClass("Service.*")) .AND
15              (Fields.named("field.*")) );
16    }
17  }
18 }

```

**Figure 4. A PROSE aspect for connecting all instances of classes whose name starts with 'Service' to a database using the Hibernate API**

The aspect extends the *DefaultAspect* base class (line 1). An aspect may contain one or more crosscut objects. A crosscut object defines an advice method (line 5) and describes the join-points where the advice should be executed. In Figure 4 there is just one crosscut corresponding to the *makePersistent* instance field (line 3). The advice action is defined in lines 5-11. The number and types of join-points defined by *makePersistent* depend on the signature of the advice method and on a specializer object attached to the crosscut (lines 13-16). The signature (line 5) restricts the execution of the advice to field modifications having the type *Integer* and being declared in classes assignable to *Object*. The specializer further restricts the set of join-points to all fields whose name matches the regular expression '*field.\**' and are being declared in classes whose name starts with 'Service'. Specialization is achieved using the *pointCutter()* method (line 13). The specializer passed to the *makePersistent* crosscut on line 13 is a logical-AND composition of two predefined PROSE specializers, namely *Fields.declaredInClass()* and *Fields.named()*. Specializers may be combined using OR and NOT operations as well. The operations allow a flexible construction of specializer objects out of predefined building blocks. The aspect shown in Figure 4 tells PROSE to capture field modifications having the type *Integer* and being declared in classes

whose name starts with *'Service'*, and persist them into the database.

## 2.5 Introducing Aspects at Run-time

In this section we describe how to extend the functionality of an application by transparently adding persistence at run-time using the aspect described in the previous section. When an aspect is woven into the running application, the join-point generator decomposes the aspect into join-point requests and activates join-points over the PROSE API. Immediately after insertion, PROSE inspects all the classes currently loaded by the JVM and gathers all fields whose name matches the regular expression *'field.\*'* and are being declared in classes whose name starts with *'Service'* in case of the previous example. When an active join-point is reached, the program execution is temporarily suspended, and the JVM passes the control of execution to PROSE. At this point, the PROSE dispatching logic calls the functionality in the crosscut of the aspect instances that registered the join-point. During this step, PROSE inspects the current thread stack and passes the gathered information to the advice method of the crosscut. In our example, when instance fields, having the type *Integer* and being declared in classes assignable to *'Service.\*'*, have been modified, the advice method (*SET\_ARGS*) takes the current field (i.e., the field at the current crosscut position), extracts its value and writes it into persistent store. After the advice execution is completed, PROSE returns the control to the application.

The installation of the ORM is performed by the PROSE persistence extension. The aspect contains database connectivity parameters and mappings specific to the current ORM used. After the instantiation, the mapper inspects the objects status. If the objects are known to the database, ORM attempts to restore their state. If not, it attempts to store the objects in the database. The ORM inspects the object status and discovers all fields that have to be synchronized with the database. For all these fields, PROSE extensions that report their modification as described in the previous section are installed.

Aspects in PROSE are not permanent. Aspects may be woven under a lease system. If the lease for an advice is not periodically renewed, the advice is immediately removed from the application. Leases play a crucial role in adaptation in mobile applications (advices that are location dependent), pervasive computing (advices acquired by a mobile device in a given network environment are dropped as the device moves to another network environment), and temporal adaptation (add an advice for a given period of time).

The aspect can also be withdrawn dynamically (unwoven) leaving the application as though no insertion ever took place. When the aspect is removed, the join-points are deactivated and the corresponding interception(s) will no longer take place. A more detailed description of the PROSE weaving mechanism can be found in [10, 5].

## 3 Performance Evaluation

To evaluate the performance of our approach we performed benchmarks and compared it with different persistence support systems: state of the art ORMs (Oracle TopLink and JBoss Hibernate). We measured the time to dynamically persist the state of an application using the persistence support systems mentioned above and PROSE. All experiments were performed on an AMD Athlon MP 1600+ 1.4 GHz, double processor machine with 1GB RAM running Linux 2.6.8. We compared results using the Oracle 9i database (Release 9.2) and the Java environment: SUN JDK 5.0 with the JVM running in debugging mode.

We performed our measurements using objects of the same type, i.e., instances of a test class that contains a single field of type *'int'*. Each measurement ran with an increasing number of objects: 1, 50, 100, 500, 1000 objects. We ran all the measurements ten times and then calculated the average of the execution times measured. Prior to each measurement, an initial run with one object was performed to assure that all involved classes were loaded in the VM and to be certain that the database connection was initialized. This single object configuration is not part of the performance measurements. The standard deviation for these experiments is less than 14%.

### 3.1 Evaluation of Oracle TopLink and Hibernate ORMs

To measure the efficiency of our approach, we performed a number of experiments. We compare our system with Oracle TopLink and Hibernate ORMs. For each experiment, we measured the time to insert new objects into the database and the time to update the objects already present in the database. The results are summarized in Figure 5 and Figure 6. The results indicate how much time is spent to statically and dynamically persist the application state. We calculated the relative overhead of the dynamic AOP support, which is a good indicator of the efficiency of the approach.

In the first experiment, we established the base line for comparison: we ran the test application using Oracle TopLink and Hibernate ORMs. Figure 5 and Figure 6 (columns 1 and 2) illustrate the results of this experiment.

In the second experiment, we used PROSE to intercept the object's fields' modifications at run-time using the *'set'* methods and store them persistently into the database using Hibernate and Oracle TopLink. We performed a number of measurements with the complete AOP system. We made the measurements with an activated method execution join-point. In this experiment, one simple aspect is applied upon a *'set'* method. The aspect has one crosscut that installs a method exit watch on the *'set'* method of the test class. To assure that the advice is called, a counter is increased each time it is executed. The results are shown in columns 3 and 4 of Figure 5 and Figure 6. To estimate the

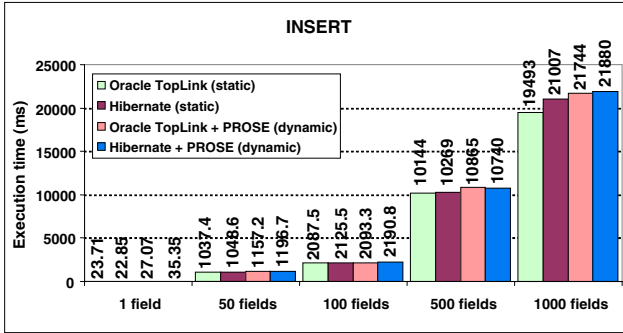


Figure 5. INSERT measurements with Oracle TopLink, Hibernate and PROSE with SUN JVM

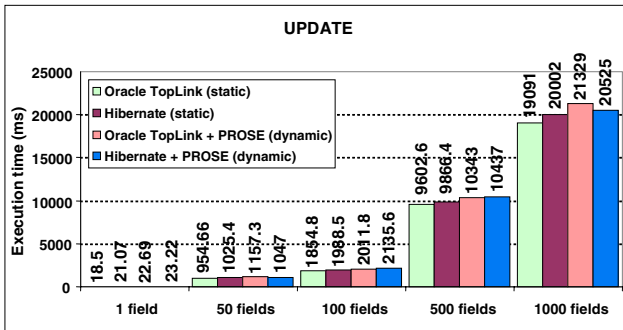


Figure 6. UPDATE measurements with Oracle TopLink, Hibernate and PROSE with SUN JVM

overhead of PROSE, we calculated the time in percentage spent by PROSE to capture object field changes and persist them into the database. The results indicate that the AOP support leads to an average overhead of 8.92% in case of insert and 14.35% in case of updates for Oracle TopLink over the static case. Given the advantage of dynamic persistence, this overhead is quite promising and proves the potential of the approach. The overhead observed is due to the time needed by the JVM to stop at the code location where the advice method should be called. Once the join-point is reached, PROSE spends some time establishing the correct advice to be called, calling the advice, and executing the body of the advice.

## 4 Conclusion

In this paper we have presented an implementation of dynamic persistence. Our approach is a Java-based dynamic AOP infrastructure where persistence functionality can be dynamically added or removed from running applications. By using our approach, persistence behavior can be added to existing applications without requiring to modify their original code. In this way the application developer is relieved from coping with the programming difficulty of making changes to the original application in order to persist its state.

We have performed performance data on the costs to per-

sist the state of an application statically and dynamically using different state of the art ORMs. Our performance evaluation indicates that the overhead to transparently persist the state of an application is modest. The results illustrate that the adaptation mechanism we presented can be used to effectively persist the state of an existing application without the application itself being aware of this. The advantage of this approach is the flexibility of the architecture that allows the exploration of new forms of persistence extensions. PROSE is an open source project and can be downloaded from <http://prose.ethz.ch>.

## References

- [1] M. Atkinson, L. Daynes, M. J. Jordan, T. Printezis, and S. Spence. An orthogonally persistent Java. In *SIGMOD Record*, volume 25, issue 4, pages 68-75, ACM Press, New York, NY, USA, 1996.
- [2] M. Atkinson and R. Morrison. Orthogonally persistent object systems. In *The VLDB Journal*, volume 4, number 3, pages 319-402, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1995.
- [3] Hibernate website. <http://www.hibernate.org/>.
- [4] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect oriented programming. In *ECOOP '97, Jyväskylä, Finland*, volume 1241 of LNCS, pages 220-242. Springer-Verlag, June 1997.
- [5] A. Nicoara and G. Alonso. Dynamic AOP with PROSE. In *International Workshop on Adaptive and Self-Managing Enterprise Applications (ASMEA'05) in conjunction with CAISE'05, Porto, Portugal*, June 2005.
- [6] ObjectStore PSE website. <http://www.objectstore.com/>.
- [7] OMG. CORBA Component Model Specification, Version 3.0, June 2002.
- [8] Oracle TopLink website. <http://www.oracle.com/technology/products/ias/toplink/>.
- [9] A. Popovici, G. Alonso, and T. Gross. Just in Time Aspects: Efficient Dynamic Weaving for Java. In *2nd International Conference on Aspect-Oriented Software Development, Boston, USA*, March 2003.
- [10] A. Popovici, T. Gross, and G. Alonso. Dynamic Weaving for Aspect Oriented Programming. In *1st International Conference on Aspect-Oriented Software Development, Enschede, The Netherlands*, April 2002.
- [11] A. Rashid. On to aspect persistence. In *GCSE Symposium, Springer-Verlag LNCS 2177*, October 2000.
- [12] A. Rashid. Aspect-oriented programming for database systems. In *Chapter in book on Aspect-Oriented Software Development. Editor(s): M. Aksit, S. Clarke, T. Elrad, R. Filman*, 2004.
- [13] A. Rashid and R. Chitchyan. Persistence as an aspect. In *International Conference on Aspect-Oriented Software Development (AOSD'03)*, March 2003.
- [14] E. L. Sergio Soares and P. Borba. Implementing Distribution and Persistence Aspects with AspectJ. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'02)*, ACM Press, pp. 174-190, 2002.
- [15] Sun Microsystems. Enterprise Java Beans Specification, Version 2.1, November 2003.