Orthogonal Object Versioning in an ODMG compliant Persistent Java

A. Marquez
Department of Computer Science, Australian National University,
Canberra ACT 0200, Australia

Email: alonso@cs.anu.edu.au

Abstract

This paper presents a general framework for Object Versioning in an ODMG-2 compliant Persistent Java (The DEJA system). We define the concept of Orthogonal Versioning as a way of achieving separation of concerns of the program with respect to the Object Versioning capabilities. We show how to add persistence to the Object Versioning classes using any implementation of the ODMG-2 Java binding. In particular we present a persistent implementation of Orthogonal Object Versioning using our ODMG-2 Java binding implementation.

1 Introduction

Object Versioning is a natural concept in many object-oriented applications. During our implementation of demonstrators using the DEJA technology we have discovered the convenience of object versioning for implement long transactions and the creation and manipulation of snapshots of the data (Historic Snapshots). This paper deals with the design and implementation of version facilities over the DEJA system. DEJA (Database Extensions for Java Applications) simplify the development of applications requiring access to persistent data. Through its appropriate separation of concerns, the application programmer, for the most part, need not be concerned with the manner in which data is made persistent. This separation of concerns greatly eases the programmer's task and allows them to concentrate on just the one task at hand – programming the business logic of the application. Moreover, it offers advantages in terms of platform/product portability so that a project is not bound to one particular vendor. DEJA implements the ODMG 2.0 Java binding standard and it is a 100% Pure Java solution. It may be used as a gateway over more traditional solutions to Java persistency such as JDBC and native calls to specialized Object Stores.

2 Design principles

Compliance with the ODMG-2 Java binding standard.

Separation of Object Versioning concerns.

2.1 Compliance with the ODMG-2 Java binding standard

We have tried to achieve maximal ODMG-compliance in designing the Object Versioning extension in order to minimize the learning curve for the application programmer. Any versioned object is also a Persistent Capable object. The implementation of read and write barriers [FootNoteReadWriteBarrier] in DEJA is extended in the Object Versioning classes to take account of additional object versioning functionality. For example, the ReadOnlyView class (one of the basic versioning classes) modifies the default semantic of the write barrier to avoid put any modified instance in the dirty list of objects.

2.2 Separation of Concerns and Orthogonal Object Versioning

The concept of orthogonal persistence [AM95], which has been under development since the early 1980's has a natural analogue in 'orthogonal versioning'. While orthogonal persistence allows the programmer to abstract over the concern of persistence, orthogonal versioning abstracts over versioning. This allows the programmer to write code without regard to the issue of versioning, except insofar as that the programmer finds it desirable to explicitly do so.

The first two principles of orthogonal persistence have obvious analogies in orthogonal versioning:

Version Independence The form of a program is independent of the version of the data which it manipulates.

Data Type Orthogonality All mutable data types should be allowed to be versioned, irrespective of their type.[1]

A meaningful implementation of the concept of 'version independence' must provide transparent support for *configuration management*[2] in order to facilitate transparent access to and manipulation of a consistent collection of component versions of a complex object. The importance of configuration management has been widely recognized in areas such as CAD, CASE and web document management systems. One of the consequences of configuration management support is the need to be able to transparently generate new versions of an object. For example, a new version of an object must be created the first time the value of the object is updated within the scope of a new configuration. That is to say that versions are not generated for each update to an object, but rather new versions are generated at the granularity of the instantiation of new configurations.

To the authors' best knowledge the concept of 'orthogonal versioning' is new and previously unimplemented.

3 Implementation

3.1 The UPSIDE project

The UPSIDE research group at the Australian National University is concerned with bringing the ideal of OPJ towards industrial relevance through performance and functionality. For this reason, performance issues being addressed by the project include high efficiency storage, byte code optimizations and Java Virtual Machine (JVM) optimizations. Key functionality issues include the efficient integration of powerful transaction models into the OPJ VM (long and short live transactions), and support for object instance and class versioning.

The DEJA (Database Extensions for Java Applications) system is a transparent, portable and scalable implementation of the ODMG-2 Java binding [**?**]. This implementation was built based on the concepts of separation of concerns. Object properties such as persistence, versioning or transactions are stored in special meta-classes that may be defined and modified independently of the application. A process of semi-dynamic program transformation at class loading time extends the semantic of user classes based on the current properties of associated meta-classes. For example, special Java View classes that map the user objects to a specific data source are generated on the fly. Object properties can be modified between different executions without any change of the program source code. As a result, the same application may make use of data sources that going from normal files to relational or object oriented databases.

3.2 Orthogonal Versioning in Java

Object versioning is a natural concept in many object-oriented applications. Versioning becomes important in applications where an exploratory or iterative approach over the object store state is needed.

Figure 1 illustrates the abstraction provided by OVJ, and depicts the relationship between versions and 'configurations'. In this example, the global environment includes many different configurations, while the depicted object only has versions in four of these. The power of the abstraction over versions provided by OVJ is seen in the left side of the digram, which presents a simple view of the object in a particular configuration *context*, while the underlying situation (which OVJ abstracts over) is the complex version tree depicted at the right. The importance of a configuration context is that it allows the denotation of versions to be implicit rather than explicit, which is essential to our goal of transparency. Having established a configuration context (perhaps implicitly), user computation proceeds, oblivious to the issue of versions, over a normal object graph which in fact corresponds to some projection of a versioned object graph into the Java heap space. In the remainder of this section we discuss four major aspects of our OVJ implementation.

Runtime overhead An important goal in designing OVJ was minimizing the performance impact on objects that are naturally transient and so not relevant to versioning. This is achieved by only applying versioning to objects at the time that they first become persistent. Any object that because of its transience never becomes persistent, avoids

[1] The principle of Data Type Orthogonality applies only to *mutable* types as the concept of versioning makes no sense except where the object may be mutated. Another way of looking at this is that immutable types are trivially versioned, but because the objects are immutable and therefore never updated, all 'versions' of the object present the same value.

[2] The term 'configuration' is used to denote a consistent view of a set of versioned objects. This is analogous to the concept of a 'release' in the context of a software management system—a release corresponds to a set of files each at a particular (but typically different) point in their own version history.

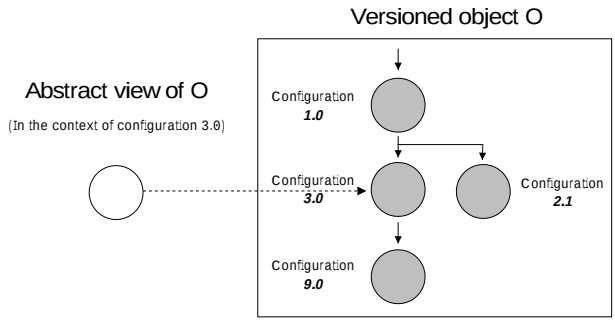

Figure 1: A versioned object in OVJ. The user's perspective (left) abstracts over the complex version hierarchy (right). The particular version of the object seen by the user is an implicit function of context (in this case configuration 3.0).

most of the overhead associated with versioning. However, some overhead is unavoidable, even for transient versioned objects. Under OVJ all instances include an additional field that is used for storing a reference to version information for that object, regardless of whether the particular instance is sufficiently long lived to be subject to versioning.

Orthogonality With Respect to Versioning and Persistence In order to ensure the persistence of all versioned objects, in the context of OVJ, OPJ's approach to defining persistent roots is extended by making all *versions* of non-transient static class members roots (in OPJ persistence is defined in terms of reachability from the static class members). In order to meaningfully fulfill the principle of version independence, OVJ must provide transparent support for configuration management. In other words, there must be means for transparently accessing and manipulating a consistent collection of component versions of a complex object. This approach provides a transparent and simple abstraction over versioning by allowing the basic versioning classes to be extended and customized. The versioning framework on which OVJ is constructed also provides a very general and powerful approach to versioning that has application to many different contexts, and that need not be as transparent.

The Configuration Hierarchy A configuration represents a set of versioned object versions, each configuration containing no more than one version of a given object. Each configuration except the first has a parent, and each may have any number of children. Thus, a linked set of configurations corresponds to a n-ary tree, and a linear sequence of configurations corresponds to a branch. In figure 1 a subset of the nodes comprising a configuration tree can be seen in the version tree of object O. While the very purpose of OVJ is to provide a means of abstracting over versions, it is necessary to provide the user with some way of explicitly controlling the configuration hierarchy. Minimally, such control might just involve a means for instantiating new configurations (akin to ABS-BR historical snapshots). OVJ also provides a much more powerful facility which allows the user to generate and manipulate a configuration hierarchy in quite general ways, including branch generation and configuration tree navigation. The current implementation does not support automatic configuration merging (which derives a new configuration by merging two or more existing configurations). However, a special class of configuration exists, whose instances may have more than one parent configuration (configuration merges turn the set of configurations from an n-ary tree to a directed acyclic graph). OVJ also supports the concept of configuration 'freezing'. A configuration may be labeled as frozen, with any attempt to write to a member of that configuration triggering an exception.

OVJ's Underlying Object Model In this section we describe something of the model underlying OVJ's implementation. While this detail is not exposed to the user of OVJ, it may serve to give the reader a better understanding of some of the issues faced in implementing OVJ. The basis for OVJ is a versioning framework which is implemented by semantically extending Java classes at class loading time through use of the semantic extension framework. For a class to be versioned, it must be (transparently, by the SEF) either made to inherit (directly or indirectly) from the class `VersionView` (or made to implement the interface `ObjectVersioning`). Each instance of `VersionView` includes a reference to a configuration instance and to the specific version instance associated with the given configuration[3]. (Such an instance corresponds to the object at the left of figure 1, which in practice would also refer to the configuration instance corresponding to 3.0.) Each of the specific version instances are instances of an automatically generated

[3] An instance of `VersionView` may also reference a *virtual configuration*. A virtual configuration represents the set of versioned object versions that satisfy a given condition. Virtual configurations typically span a configuration sub-branch.

class corresponding to the base class of the object, but containing only those fields that must be made persistent and with the addition of fields that link the various versions of the instance (these instances correspond to the gray objects at the right of figure 1). This approach of dynamic creating new classes in a manner that is transparent to the user is common to OPJ, and although beyond the scope of this paper, is described in some detail in [MZB00].

Versioning API OVJ provides an versioning API for all objects, which includes the following methods:

```
public ObjectVersioning getClosestVersion(String name);
public ObjectVersioning getCreateVersion(String name);
public void deleteCurrentVersion();
public ConfigurationInterface getConfiguration();
```

The first two, `getClosestVersion()` and `getCreateVersion()`, both return an instance of the object corresponding to the given configuration (if the configuration does not exist an exception is thrown). If such an instance version does not exist the first method will return the closest ancestor and the second will create a new version. The third method separates the associated specific version from the other versions of the object. The last returns the configuration associated with the instance it is called over.

3.3 A first prototype

A first prototype has been ready by the end of last year. Memory use is minimized sharing common object instances between configurations. The configuration and object versioning support is quite general and all the new methods may be specialized in order to customize the object versioning framework. Thus, a general-purpose object versioning framework is obtained.

Our initial result shows that the Object Versioning add an additional penalty of a 30% in the execution time.

4 conclusions

This paper has described the design and implementation of the Object Versioning support in the DEJA system, which extends the ODMG-2 Java binding to provide version functionality. We gave several design principles such as separation of programming and object storage concerns, compliance with the ODMG standard and minimization of performance overhead. In particular, we have presented an adaptation of the concepts of Orthogonal Persistence to Object Versioning: Orthogonal Object Versioning. We have built a first prototype of an object versioning framework based on Orthogonal Object Versioning on top of our DEJA prototype. The use of the concept of Orthogonal Versioning allows us to provide a more general and flexible solution compared to any of the Object Versioning solutions available in OODB. To the best of our knowledge, the Object Versioning support in the DEJA system is the only one to provide to the application programmer a complete separation of concerns with respect to the object storage and the object versioning. The separation of persistence concerns provided by the use of Object Views and the integration of Transparent Object Versioning with the ODMG-2 Java binding standard allows the developer to take advantage of a widely available, comprehensive information-processing infrastructure.

References

[AM95] Malcolm P. Atkinson and Ronald Morrison. Orthogonally persistent systems. *The VLDB Journal*, 4(3):319–402, July 1995.

[MZB00] Alonso Marquez, John N Zigman, and Stephen M Blackburn. Fast portable Orthogonally Persistent Java. *Software—Practice and Experience*, 2000. (to appear).