Generically Fixing Object Identity

Adrian Moos, EL4J Team, ELCA
http://www.elca.ch

Introduction

Object identity is an important concept in OOP, but is not always established or maintained properly. For instance, sending an object back and forth over a network (using any standard remoting protocal) will create new objects which can result in several copies of the same logical object residing within a single process. Similarly, losing an object-relational-mapper's (ORM) session between load invocations (e.g. for stateless servers and long business transactions) typically results in creating multiple proxies to the same persisted object. Accepting this loss of identity contradicts OO methodology and substantially complicates the programming model. We propose a lightweight solution for the problem. We justify why maintaining object identity is important and we explain our solution to the problem.

Importance of Object Identity

As an introductory example, consider a remotely-used processing layer, exposing operations to load entities from a database using an object-relational mapping (ORM) layer. Let us say, the entities are of types *car* and *wheel*, and that the operations on the car propagate to its wheels. We assume further that we have a simple GUI implementation that does not take into account that object identity is violated. A user wishes to create a new car and to attach some wheels. He first creates a car (instance c0) and then loads a wheel (instance w0). He opens a detail view for w0, verifies that the wheel is in working condition, locally attaches w0 to c0 and saves c0. Meanwhile, another user enters information into the database that the attached wheel’s tire is punctured. A moment later, the first user fetches the list of defunct cars from the server, which features a single instance c1 with attached wheel w1. With some surprise, the user notes that c1 is the car he previously created but there is a message saying that tire of its wheel is punctured. Surprised, he looks at the still open view for w0, which asserts that it is without defects. Only after hitting the refresh button (which the user would normally have no reason to do) is a new version (=w2) of the wheel fetched from the database and displayed. The user now fixes the punctured tire and hits save. To his astonishment, the list of defunct cars still states c1’s tire is flat.

In short, the simple GUI shows an inconsistent state while the database is consistent. This is clearly unsatisfactory.

Such flaws are usually adressed by controller code triggering cascading updates. We think fixing the client’s data model is a more direct and simpler approach: The persistence layer guarantees that any state loaded by the user in a single transaction is consistent, the problem merely is the possible coexistence of conflicting object versions. The most frequent case of such conflicts are several differing versions of the same logical object. This can be prevented by enforcing that the logical object is represented by a unique object in the data model.

Need For A Fix

Many standard technologies do not provide the notion of object identity most approapriate for the task at hand. For instance:

1. An object relational mapper such as Hibernate guarantees that every logical object is represented by a unique instance *within a session*. Sessions keep state and and are not mobile, making their use in another tier problematic.

2. Standard remoting protocols do not keep serialization context across calls, can therefore not recognize object instances transmitted in a previous request, and may thus fail to preserve object uniqueness.

Approach

The IdentifyFixer intercepts access to sources that have a wrong concept of identity and restores proper identities for and within the objects they produce. This is accomplished by traversing their object graph, passing through it and fixing each reference contained therein. For each contained reference r we do the following 3 steps:

1. Find the unique object for the logical identity.
2. Propagate the state from the object r points to to the unique object.
3. Replace r with a reference to the unique object.

Step 2 is performed by copying the object's internal representation using reflection[1]. During this step, the IdentityFixer also notifies registered observers about the change (this is useful to refresh objects whose state may become invalid by the update to unique object).

Our implementation is relatively generic. The only context-specific knowledge is located in the methods of a subclass which (i) when given an object, determine its logical identity (for remote access to an ORM, this is the object's primary key); and (ii) return whether an object is an immutable value type (this is used for performance optimization e.g. not copying Strings).

Application: Remoting Hibernate in Spring Rich Client Applications

We use a Spring interceptor to forward requests to the IdentityFixer. We make no assumptions about the remote API, thereby enabling its easy extension. Except for hibernate proxies[2], the subclass identifying objects is straightforward. We have validated the approach by a couple of unit-tests and through a demo application.

A special challenge is handling objects with a yet unknown logical identity, which arises e.g. if logical identities are not assigned on the tier creating the object (such as with database-generated primary keys). Our solution supports fixing the identities of such objects by manually providing the unique object instance. In case of AOP interception, the instance is determined from a JDK 1.5 annotation on the method in the remote API. For instance, a remote method to save an entity could be declared as:

```
@ReturnsUnchangedParameter
T saveOrUpdate(T entity) throws DataAccessException,
    DataIntegrityViolationException, OptimisticLockingFailureException;
```

The intercepting Identity Fixer concludes from the annotation that the object graph reachable from the parameter is identical with the object graph returned, and maps each identity-sensitive object in the return graph to the corresponding object in the original graph. Since we work on the internal representation, this works even if the public API does not permit access to, or recognition of objects (for instance if a java.util.Set is

[1] Our implementation assumes that all logically identical objects have the same dynamic type.

[2] Due to limitations with respect to the dynamic type of proxies (it may be incorrect in subtype relations), the identities of proxies can in general not be fixed. In addition, the handling of proxies is particularly involved due to their different internal state representation, which would require manual coding and additional hooks in the superclass. To keep matters simple, we chose not to fix identities of proxies at this time.

passed). On the other hand, the algorithm can not cope with different internal representations for the same externally visible behavior.

Example usage

```
public class IdentityFixerDemo {
    static AbstractIdentityFixer identityFixer
        = new HibernateProxyAwareIdentityFixer();

    static AbstractIdentityFixer.GenericInterceptor interceptor
        = identityFixer.new GenericInterceptor(IdentityFixedRepository.class);

    void test(PersonRepository identityManglingRepo) {
        IdentityFixedPersonRepository fixedRepo
            = (IdentityFixedPersonRepository)
                interceptor.decorate(identityManglingRepo);

        Person p = new Person();
        p.name = "Mickey Mouse";
        assert !p.married;
        fixedRepo.saveOrUpdate(p);

        // we assume there is only one person named Mickey Mouse ;-)
        Person p2 = fixedRepo.findByName("Mickey Mouse");
        assert p == p2;
        p2.married = true;
        assert p.married;
    }
}
```

Further Information

This software is part of ELCA's open source application framework EL4J (http://el4j.sourceforge.net/). The code can be found in our subversion repository at sourceforge.

The implementing classes are:
ch.elca.el4j.services.persistence.generic.repo.AbstractIdentityFixer
ch.elca.el4j.services.persistence.hibernate.HibernateProxyAwareIdentityFixer

Our unit test is in
ch.elca.el4j.tests.refdb.repo

Since Adrian is leaving ELCA to further pursue his studies, questions should be directed to: Philipp \dot\ oser \at\ elca \dot\ ch