# Pojo Cache 1.4.1

# User Guide

# 4.3

Ben Wang, Scott Marlow

**ISBN: N/A**
**Publication date:**

This book is a User Guide for Pojo Cache

# Pojo Cache 1.4.1: User Guide

Author Ben Wang, Scott Marlow

The GPG fingerprint of the security@redhat.com key is:

CA 20 86 86 2B D6 9D FC 65 F6 EC C4 21 91 80 CD DB 42 A6 0E

1801 Varsity Drive
Raleigh, NC 27606-2072
USA
Phone: +1 919 754 3700
Phone: 888 733 4281
Fax: +1 919 754 3701
PO Box 13588
Research Triangle Park, NC 27709
USA

# Preface

PojoCache is an in-memomy, transactional, and replicated POJO (plain old Java object) cache system that allows users to operate on a POJO transparently without active user management of either replication or persistency aspects. PojoCache, a component of JBossCache (uses PojoCache class as an internal implementation, the old implementation TreeCacheAop has been deprecated.), is the first in the market to provide a POJO cache functionality. JBossCache by itself is a 100% Java based library that can be run either as a standalone program or inside an application server.

This document is meant to be a user guide to explain the architecture, api, configuration, and examples for PojoCache. We assume the readers are familiar with both JGroups and TreeCache usages. Since PojoCache uses JBossAop framework, an introduction will also be given there.

If you have questions, use the user *forum*[1] linked on the JBossCache website. We also provide tracking links for tracking bug reports and feature requests on *JBoss Jira web site*[2] . If you are interested in the development of PojoCache, post a message on the forum. If you are interested in translating this documentation into your language, contact us on the developer mailing list.

JBossCache is an open-source product based on LGPL. Commercial development support, production support and training for JBoss Cache is available through JBoss Inc. (see *JBoss web site*[3] ). JBoss Cache is a project of the JBoss Professional Open Source product suite.

In some of the example listings, what is meant to be displayed on one line does not fit inside the available page width. These lines have been broken up. A '\' at the end of a line means that a break has been introduced to fit in the page, with the following lines indented. So:

```
Let's pretend to have an extremely \
long line that \
does not fit
This one is short
```

Is really:

```
Let's pretend to have an extremely long line that does not fit
This one is short
```

---

[1] http://www.jboss.com/index.html?module=bb&op=viewforum&f=157

[2] http://jira.jboss.com

[3] http://www.jboss.com

# Terminology

## 1. Overview

The section lists some basic terminology that will be used throughout this guide.

Aop

Aspect-Oriented Programming (AOP) is a new paradigm that allows you to organize and layer your software applications in ways that are impossible with traditional object-oriented approaches. Aspects allow you to transparently glue functionality together so that you can have a more layered design. AOP allows you to intercept any event in a Java program and trigger functionality based on those events.

JBossAop

JBossAop is an open-source Aop framework library developed by JBoss. It is 100% Java based and can be run either as a standalone or inside an application server environment. More details can be found at www.jboss.com.

Dynamic Aop

Dynamic Aop is a feature of JBossAop that provides a hook so that a caller can insert event interception on the POJO at runtime. PojoCache currently uses this feature to perform field level interception.

JGroups

JGroups is a reliable Java group messaging library that is open-source and LGPL. In addition to reliable messaging transport, it also performs group membership management. It has been a de facto replication layer used by numerous open-source projects for clustering purposes. It is also used by JBossCache for replication layer.

TreeCache

TreeCache is a component of JBossCache that is a plain cache system. That is, it stores a straight Java object reference and requires an object to be serializable to perform binary-wide replication. It is Java-based, in-memory, replicated, and persistent. PojoCache currently is a sub-class of TreeCache.

POJO

Plain old Java object.

Annotation

Annotation is a new feature in JDK5.0. It introduces metadata along side the Java code that can be accessed at runtime. PojoCache currently use both JDK1.4 and JDK50 annotation to support POJO instrumentation.

Prepare

Prepare is a keyword in JBossAop pointcut language used to specify which POJO needs to be instrumented. It appears in a `jboss-aop.xml` file. However, if you can use annotation to specify the POJO instrumentation, there is no need for a `jboss-aop.xml` listing. Note that When a POJO is declared properly either through the xml or annotation, we consider it "aspectized".

Instrumentation

Instrumentation is an Aop process that basically pre-processes (e.g., performing byte-code weaving) on the POJO. There are two modes: compile- or load-time. Compile-time weaving can be done with an Aop precompiler (`aopc`) while load-time is done to specify a special classloader in the run script. This step is necessary for an Aop system to intercept events that are interesting to users.

# Introduction

## 1. Overview

The two components in JBossCache, plain cache (implemented as TreeCache) and PojoCache (implemented as PojoCache), are both in-memory, transactional, replicated, and persistent. However, TreeCache is typically used as a plain cache system. That is, it directly stores the object references and has a `HashMap`-like Api. If replication or persistency is turned on, the object will then need to implement the `Serializable` interface. In addition, it has known limitations:

- Users will have to manage the cache specifically; e.g., when an object is updated, a user will need a corresponding API call to update the cache content.
- If the object size is huge, even a single field update would trigger the whole object serialization. Thus, it can be unnecessarily expensive.
- The object structure can not have a graph relationship. That is, the object can not have sub-objects that are shared (multiple referenced) or referenced to itself (cyclic). Otherwise, the relationship will be broken upon serialization. For example, Figure 1 illustrates this problem during replication. If we have two `Person` instances that share the same `Address` , upon replication, it will be split into two separate `Address` instances (instead of just one).

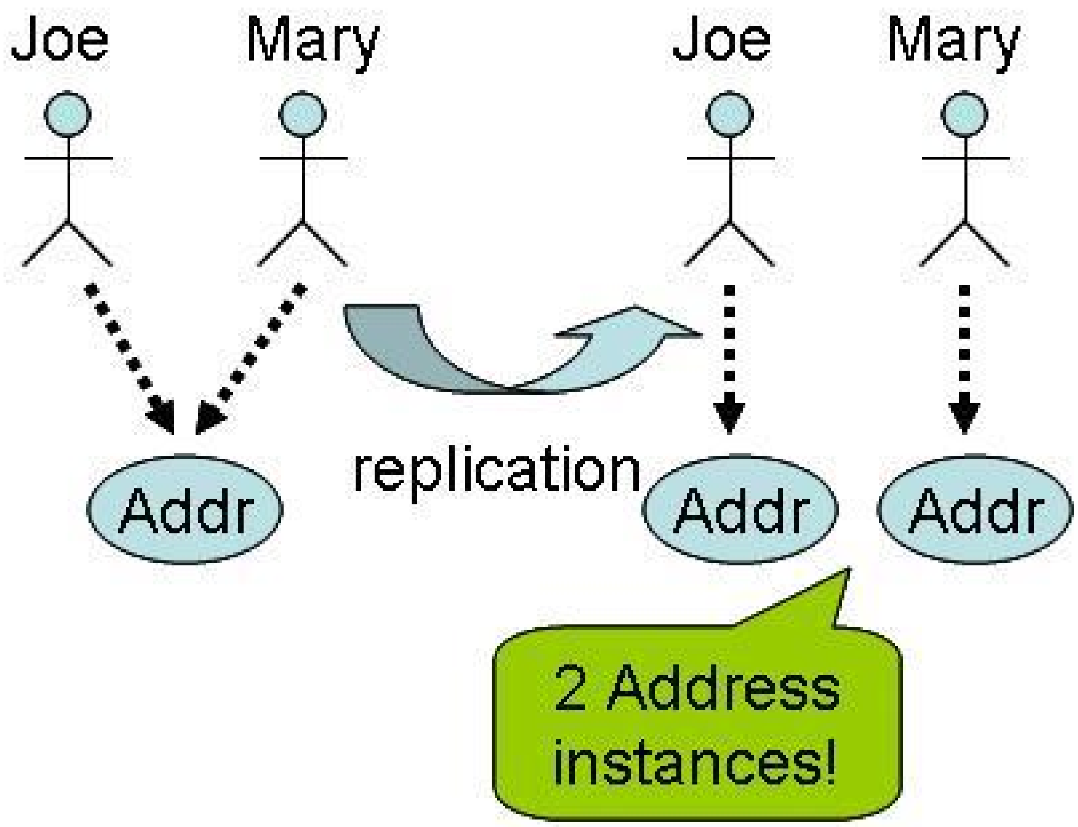


Figure 2.1. Illustration of shared objects problem during replication

PojoCache, on the other hand, is a fine-grained "object-oriented" cache. By "object-oriented", we mean that PojoCache provides tight integration with the object-oriented Java language paradigm, specifically,

- no need to implement `Serializable` interface for the POJOs.
- replication (or even persistency) is done on a per-field basis (as opposed to the whole object binary level).
- the object relationship and identity are preserved automatically in a distributed, replicated environment. It enables transparent usage behavior and increases software performance.

In PojoCache, these are the typical development and programming steps:

- Declare POJO to be "prepared" (in Aop parlance) either through an external xml file (i.e., `jboss-aop.xml`), or through annotation inside the POJO. Depending on your preference, you can either pre-instrument your POJO (compile time) or have JBossAop do it at load time.
- Use `putObject` Api to put your POJO under cache management.

- Operate on POJO directly. Cache will then manage your replication or persistency automatically.

More details on these steps will be given in later chapters.

`PojoCache` also extends the functionality of TreeCache to object based. That is, the TreeCache features such as transaction, replication, eviction policy, and cache loader, has been extended to POJO level. For example, when you operate on a POJO (say, `pojo.setName()`) under a transaction context, it will participate in that transaction automatically. When the transaction is either committed or rolled back, your POJO operations will act accordingly.

Finally, `PojoCache` can also be used as a plain `TreeCache` . For example, a user can use the `TreeCache` API [e.g., `get(String fqn)` and `set(String fqn, String key, String value)` ] to manage the cache states. Of course, users will need to consider the extra cost (albeit slight) in doing this.

## 2. Features

Here are the current features and benefits of PojoCache:

- Fine-grained replication. The replication mode supported is the same as that of the TreeCache: `LOCAL` , `REPL_SYNC` , and `REPL_ASYNC`. The replication level is fine-grained and is done automatically once the POJO is mapped into the internal cache store. When a POJO field is updated, a replication request will be sent out only to the node corresponding to that modified attribute (instead of the whole object). This can have a potential performance boost during the replication process; e.g., updating a single key in a big HashMap will only replicate the single field instead of the whole map! Please see the documentation of *JBossCache*[1] for more details on cache mode.

- Transaction. The POJO operation can be transacted once a TransactionManager is specified properly. Upon user rollback, it will rollback all POJO operations as well. Note that the transaction context only applies to the node level though. That is, in a complex object graph where you have multiple sub-nodes, only the nodes (or fields) accessed by a user are under transaction context. To give an example, if I have a POJO that has field references to another two POJOs (say, pojo1 and pojo2). When pojo1 is modified and under transaction context, pojo2 is not under the same transaction context. So you can start another transaction on pojo2 and it will succeed.

  In addition, fine-grained operation (replication or persistency) under transaction is batched. That is, the update is not performed until the `commit` phase. And if it is rolled back, we will simply discard the modifications.

- Eviction policy. PojoCache supports eviction policy that can evict the whole POJO object (and any field object references, recursively). Currently there is an eviction policy class called `org.jboss.cache.aop.eviction.AopLRUPolicy` (that is sub-class of `org.jboss.cache.eviction.LRUPolicy` ). The configuration parameters are the same as those of the TreeCache counterpart. Note that the concept of "Region" in eviction needs to be

---

[1] TreeCache.html

carefully defined at the top of the object FQN level. Otherwise, eviction policy will not operate correctly. That is, since "Region" is used to define the eviction policy, if you define a "Region" inside a POJO sub-tree, it may not be desirable.

- Object cache by reachability, i.e., recursive object mapping into the cache store. For example, if a POJO has a reference to another advised POJO, `PojoCache` will transparently manage the sub-object states as well. During the initial `putObject()` call, `PojoCache` will traverse the object tree and map it accordingly to the internal TreeCache nodes. This feature is explained in full details later.

- Object reference handling. In PojoCache, multiple and recursive object references are handled automatically. That is, a user does not need to declare any object relationship (e.g., one-to-one, or one-to-many) to use the cache. Therefore, there is no need to specify object relationship via xml file.

- Automatic support of object identity. In PojoCache, each object is uniquely identified by an internal FQN. Client can determine the object equality through the usual `equal` method. For example, an object such as `Address` may be multiple referenced by two `Person`s (e.g., `joe` and `mary`). The objects retrieved from `joe.getAddress()` and `mary.getAddress()` should be identical.

  Finally, a POJO can be stored under multiple `Fqn`s in the cache as well, and its identity is still preserved when retrieved from both places (after replication).

- Inheritance relationship. PojoCache preserves the POJO inheritance hierarchy after the object item is stored in the cache. For example, if a `Student` class inherits from a Person class, once a `Student` object is mapped to PojoCache (e.g., `putObject` call), the attributes in base class `Person` is "aspectized" as well.

- Support Collection classes (e.g., List, Set, and Map based objects) automatically without declaring them as aop-enabled. That is, you can use them either as a plain POJO or a sub-object to POJO without declaring them as "aspectized". In addition, it supports runtime swapping of the proxy reference as well.

- Support pre-compiling of POJOs. The latest JBossAop has a feature to pre-compile (called `aopc`, so-called compile-time mode in JBossAop) and generate the byte code necessary for AOP system. By pre-compiling the user-specified POJOs, there is no need for additional declaration file (e.g., `jboss-aop.xml` ) or specifying a JBossAop system classloader. A user can treat the pre-generated classes as regular ones and use PojoCache in a non-intrusive way.

  This provides easy integration to existing Java runtime programs, eliminating the need for ad-hoc specification of a system class loader, for example. Details will be provided later.

- POJO needs not implement the `Serializable` interface.

- Support annotation usage. Starting from release 1.2.3, PojoCache also supports declaration of POJO through annotation under JDK1.4. JBossAop provides an annotation precompiler that a user can use to pre-process the annotation. As a result, there will be no need for `jboss-aop.xml` file declaration for POJOs, if annotation is preferred. The JDK5.0 annotation will be supported in the next release.

- Ease of use and transparency. Once a POJO is declared to be managed by cache, the POJO object is mapped into the cache store behind the scene. Client will have no need to manage any object relationship and cache contents synchronization.

## 3. Usage

To use PojoCache, it is similar to its TreeCache counter part. Basically, you instantiate a PojoCache instance first. Then, you can either configure it programmatically or through an external xml file. Finally, you call the cache life cycle method to start the cache. Below is a code snippet that creates and starts the cache through an external xml file:

```
   cache_ = new PojoCache();
   PropertyConfigurator config = new PropertyConfigurator(); // configure
tree cache.
   config.configure(cache_, "META-INF/replSync-service.xml"); // Search
under the classpath
   cache_.start();
   ...
   cache_.stop();
```

## 4. Requirement

`PojoCache` is currently supported on both JDK1.4 and JDK50. For JDK1.4, it requires the following libraries (in addition to jboss-cache.jar and the required libraries for the plain TreeCache) to start up:

- Library:
  - jboss-aop.jar. Main JBossAop library.
  - javassist.jar. Java byte code manipulation library.
  - trove.jar. High performance collections for Java.
  - qdox.jar. Javadoc parser for annotation.

For JDK5.0, in addition to the above libaries, you will need to replace jboss-cache.jar with jboss-cache-jdk50.jar and jboss-aop.jar with jboss-aop-jdk50.jar from lib-50 directory.

# Introduction to JBossAop

In this chapter, we will give an overview to JBossAop, specifically to the usage pertinent to PojoCache. Material in this chapter can be found in the full JBossAop documentation. For more details, users are encouraged to visit its doc page.

## 1. What is it?

Aspect-Oriented Programming (AOP) is a new paradigm that allows you to organize and layer your software applications in ways that are impossible with traditional object-oriented approaches. Aspects allow you to transparently glue functionality together so that you can have a more layered design. AOP allows you to intercept any event in a Java program and trigger functionality based on those events. Combined with JDK 5.0 annotations, it allows you to extend the Java language with new syntax.

An aspect is a common feature that's typically scattered across methods, classes, object hierarchies, or even entire object models. It is behavior that looks and smells like it should have structure, but you can't find a way to express this structure in code with traditional object-oriented techniques.

For example, metrics is one common aspect that is orthogonal to the business logic. In AOP, a feature like metrics is called a crosscutting concern, as it's a behavior that "cuts" across multiple points in your object models, yet is distinctly different. As a development methodology, AOP recommends that you abstract and encapsulate crosscutting concerns.

For example, let's say you wanted to add code to an application to measure the amount of time it would take to invoke a particular method. In plain Java, the code would look something like the following.

```
public class BankAccountDAO
{
  public void withdraw(double amount)
  {
    long startTime = System.currentTimeMillis();
    try
    {
      // Actual method body...
    }
    finally
    {
      long endTime = System.currentTimeMillis() - startTime;
      System.out.println("withdraw took: " + endTime);
    }
  }
}
```

While this code works, it is difficult to turn it on and off and we have a code bloat if we want to obtain the metrics for all our methods. This approach to metrics is very difficult to maintain, expand, and extend, because it's dispersed throughout your entire code base. And this is just a tiny example! In many cases, OOP may not always be the best way to add metrics to a class.

Aspect-oriented programming gives you a way to encapsulate this type of behavior functionality. It allows you to add behavior such as metrics "around" your code. For example, AOP provides you with programmatic control to specify that you want calls to BankAccountDAO to go through a metrics aspect before executing the actual body of that code.

## 2. Creating Aspects in JBoss AOP

In short, all AOP frameworks define two things: a way to implement crosscutting concerns, and a programmatic construct -- a programming language or a set of tags -- to specify how you want to apply those snippets of code.

Let's take a look at how JBoss AOP, its cross-cutting concerns, and how you can implement a metrics aspect in JBoss.

The first step in creating a metrics aspect in JBoss AOP is to encapsulate the metrics feature in its own Java class. Listing Two extracts the try/finally block in Listing One's BankAccountDAO.withdraw() method into Metrics, an implementation of a JBoss AOP Interceptor class.

Listing Two: Implementing metrics in a JBoss AOP Interceptor

```
01. public class Metrics implements org.jboss.aop.advice.Interceptor
02. {
03.   public Object invoke(Invocation invocation) throws Throwable
04.   {
05.     long startTime = System.currentTimeMillis();
06.     try
07.     {
08.       return invocation.invokeNext();
09.     }
10.     finally
11.     {
12.       long endTime = System.currentTimeMillis() - startTime;
13.       java.lang.reflect.Method m =
((MethodInvocation)invocation).method;
14.       System.out.println("method " + m.toString() + " time: " + endTime
+ "ms");
15.     }
16.   }
17. }
```

Under JBoss AOP, the Metrics class wraps withdraw(): when calling code invokes withdraw(), the AOP framework breaks the method call into its parts and encapsulates those parts into an Invocation object. The framework then calls any aspects that sit between the calling code and the actual method body.

When the AOP framework is done dissecting the method call, it calls Metric's invoke method at line 3. Line 8 wraps and delegates to the actual method and uses an enclosing try/finally block to perform the timings. Line 13 obtains contextual information about the method call from the Invocation object, while line 14 displays the method name and the calculated metrics.

Having the metrics code within its own object allows us to easily expand and capture additional measurements later on. Now that metrics are encapsulated into an aspect, let's see how to apply it.

## 3. Applying Aspects in JBoss AOP

To apply an aspect, you define when to execute the aspect code. Those points in execution are called pointcuts. An analogy to a pointcut is a regular expression. Where a regular expression matches strings, a pointcut expression matches events/points within your application. For example, a valid pointcut definition would be "for all calls to the JDBC method executeQuery(), call the aspect that verifies SQL syntax."

An entry point could be a field access, or a method or constructor call. An event could be an exception being thrown. Some AOP implementations use languages akin to queries to specify pointcuts. Others use tags. JBoss AOP uses both. Listing Three shows how to define a pointcut for the metrics example.

Listing Three: Defining a pointcut in JBoss AOP

```
1. <bind pointcut="public void com.mc.BankAccountDAO->withdraw(double amount)">
2.       <interceptor class="com.mc.Metrics"/>
3. </bind >

4. <bind pointcut="* com.mc.billing.*->*(..)">
5.       <interceptor class="com.mc.Metrics"/>
6. </bind >
```

Lines 1-3 define a pointcut that applies the metrics aspect to the specific method BankAccountDAO.withdraw(). Lines 4-6 define a general pointcut that applies the metrics aspect to all methods in all classes in the com.mc.billing package.

There is also an optional annotation mapping if you do not like XML. See JBossAop Reference Guide for more information.

JBoss AOP has a rich set of pointcut expressions that you can use to define various points/events in your Java application so that you can apply your aspects. You can attach your aspects to a specific Java class in your application or you can use more complex compositional pointcuts to specify a wide range of classes within one expression.

## 4. Dynamic Aop

With JBoss AOP you can change advice and interceptor bindings at runtime. You can unregister existing bindings, and hot deploy new bindings if the given points have been instrumented. There is also a runtime API for adding advice bindings at runtime. Getting an instance of `org.jboss.aop.AspectManager.instance()`, you can add your binding.

```
AdviceBinding binding = new AdviceBinding("execution(POJO->new(..))", null);
binding.addInterceptor(SimpleInterceptor.class);
AspectManager.instance().addBinding(binding);
```

First, you allocated an `AdviceBinding` passing in a pointcut expression. Then you add the interceptor via its class and then add the binding through the AspectManager. When the binding is added the AspectManager will iterate through ever loaded class to see if the pointcut expression matches any of the joinpoints within those classes.

## 4.1. Per Instance AOP

Any class that is instrumented by JBoss AOP, is forced to implement the `org.jboss.aop.Advised` interface.

```
public interface InstanceAdvised
{
   public InstanceAdvisor _getInstanceAdvisor();
   public void _setInstanceAdvisor(InstanceAdvisor newAdvisor);
}

public interface Advised extends InstanceAdvised
{
   public Advisor _getAdvisor();
}
```

The InstanceAdvisor is the interesting interface here. InstanceAdvisor allows you to insert Interceptors at the beginning or the end of the class's advice chain.

```
public interface InstanceAdvisor
{
   public void insertInterceptor(Interceptor interceptor);
   public void removeInterceptor(String name);
   public void appendInterceptor(Interceptor interceptor);

   public void insertInterceptorStack(String stackName);
   public void removeInterceptorStack(String name);
   public void appendInterceptorStack(String stackName);

   public SimpleMetaData getMetaData();

}
```

So, there are three advice chains that get executed consecutively in the same java call stack. Those interceptors that are added with the `insertInterceptor()` method for the given object instance are executed first. Next, those advices/interceptors that were bound using regular `binds` . Finally, those interceptors added with the `appendInterceptor()` method to the object instance are executed. You can also reference `stacks` and insert/append full stacks into the pre/post chains.

## 4.2. Preparation

Dynamic AOP cannot be used unless the particular joinpoint has been instrumented. You can force instrumentation with the `prepare` functionality that declares in an xml file.

# 5. Annotations

Annotations are only available in JDK 5.0, but using our annotation compiler you can acheive similar functionality with JDK 1.4.2 as well.

Annotations must map to an annotation type, in JDK 5.0 they are defined as:

```
package com.mypackage;

public @interface MyAnnotation
{
   String myString();
   int myInteger();
}
```

Annotation types for use with the annotation compiler are defined in exactly the same way for JDK 1.4.2, with the important difference that '@interface' is replaced by 'interface'. i.e. the similar annotation type is a normal Java interface:

```
package com.mypackage;

public interface MyAnnotation
{
   String myString();
   int myInteger();
}
```

The syntax for using annotations in JDK 1.4.2 is almost exactly the same as JDK 5.0 annotations except for these subtle differences:

- they are embedded as doclet tags
- You use a double at sign, i.e. '@@'
- You MUST have a space after the tag name otherwise you will get a compilation error. (This is the quirkiness of the QDox doclet compiler used to compile the annotations.')
- You cannot import the annotation type, you must use the fully qualified name of the interface.

- You cannot specify default values for an annotation's value

This example shows an annotated class in JDK 1.4.2:

```
package com.mypackage;

/**
 * @@com.mypackage.MyAnnotation (myString="class", myInteger=5)
 */
public class MyClass
{
   /**
    * @@com.mypackage.MyAnnotation (myString="field", myInteger=4)
    */
   private String myField;

   /**
    * @@com.mypackage.MyAnnotation (myString="constructor", myInteger=3)
    */
   public MyClass()
   {
   }

   /**
    * @@com.mypackage.MyAnnotation (myString="method", myInteger=3)
    */
   public int myMethod()
   {
   }
}
```

# Architecture

Following explains the concepts and top-level design of PojoCache.

## 1. Dynamic AOP interception

JBossAop provides an API (`appendInterceptor`) to add an interceptor at runtime. PojoCache uses this feature extensively to provide user transparency. Every "aspectized" POJO class will have an associated `org.jboss.aop.InstanceAdvisor` instance. During a `putObject(FQN fqn, Object pojo)` operation (API explained below), PojoCache will examine to see if there is already a `org.jboss.cache.aop.CacheInterceptor` attached. (Note that a `CacheInterceptor` is the entrance of PojoCache to dynamically manage cache contents.) If it has not, one will be added to `InstanceAdvisor` object. Afterward, any POJO field modification will invoke the corresponding `CacheInterceptor` instance. Below is a schematic illustration of this process.

JBossAop has the capability to intercept both method level call and field level read write. From the perspective of PojoCache, field level interception is the appropriate mechanism to synchronize with the backend cache store. Please note that,

- the filed level interception applies to all access qualifiers. That is, regardless whether it is `public`, `protected`, or `private`
- we skip interception for field with `final`, `static`, and `transient` qualifiers. As a result, any field with these 3 qualifiers will not be replicated or persisted.

The figures shown below illustrate operations to perform field read and write. Once a POJO is managed by cache (i.e., after a `putObject` method has been called), Aop will invoke `CacheInterceptor` automatically every time there is a field read or write. However, you should see the difference between these figures. While field write operation will go to cache first and, then, invoke the in-memory update, the field read invocation does not involve in-memory reference at all. This is because the value in cache and memory should have been synchronized during write operation. As a result, the field value from the cache is returned.

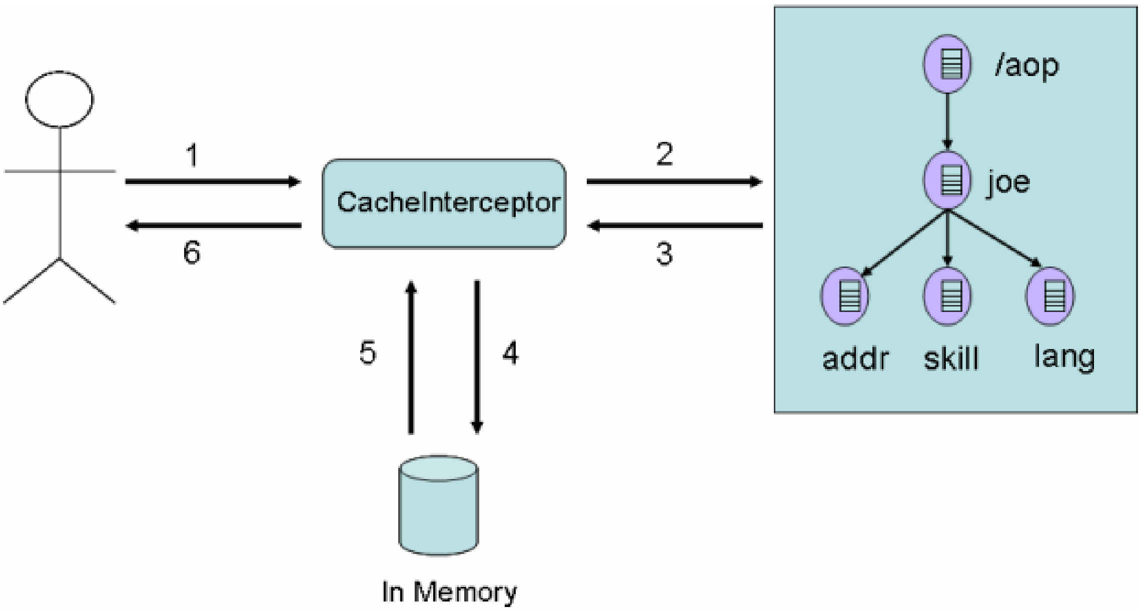


Figure 4.1. Dynamic AOP interception for field write

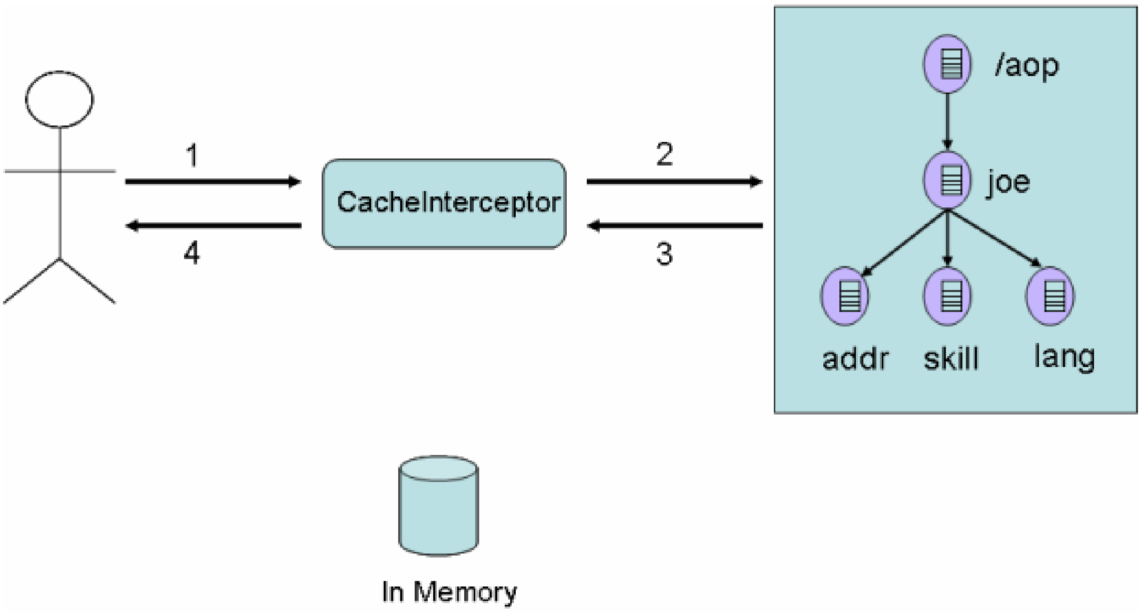


Figure 4.2. Dynamic AOP Interception for field read

## 2. Object mapping by reachability

A complex object by definition is an object that may consist of composite object references. Once a complex object is declared "prepared" (e.g., a `Person` object), during the `putObject(Fqn fqn, Object pojo)` operation, PojoCache will add a `CacheInterceptor` instance to the `InstanceAdvisor` associated with that object, as we have discussed above. In addition, the cache will map recursively the primitive object fields into the corresponding cache nodes.

The mapping rule is as follows:

- Create a tree node using `fqn`, if not yet existed.
- Go through all the fields (say, with an association `java.lang.reflect.Field` type field) in POJO,
  - If it is a primitive type, the field value will be stored under `fqn` with `(key, value)` pair of `(field.getName(), field.getValue())`. The following are primitive types supported now: `String, Boolean, Double, Float, Integer, Long, Short, Character.`
  - If it is a non-primitive type, creates a child `FQN` and then recursively executes another `pubObject` until it reaches all primitive types.

Following is a code snippet that illustrates this mapping process

```
for (Iterator i = type.getFields().iterator(); i.hasNext();) {
   Field field = (Field) i.next();
   Object value = field.get(obj);
   CachedType fieldType = getCachedType(field.getType());
   if (fieldType.isImmediate()) {
    immediates.put(field.getName(), value);
} else {
   putObject(new Fqn(fqn, field.getName()), value);
}
```

Let's take an example POJO class definition from the Appendix section where we have a `Person` object that has composite non-primitive types (e.g., List and Address). After we execute the `putObject` call, the resulting tree node will schematically look like the cache node in the following figures:

```
Person joe = new Person();
joe.setAddress(new Address());

cache.putObject("/aop/joe", joe);
```

The PojoCache APIs will be explained in fuller details later. But notice the illustration of object mapping by reachability in the following figure. The fqn `/aop/joe` is associated with the POJO `joe`. Then under that fqn, there are three children nodes: `addr,skills,` and `languages.` If you look at the `Person` class declaration, you will find that `addr` is an `Address` class, `skills` is a `Set` , and `languages` is a `List` type. Since they are non-primitive, they are recursively inserted under the parent object (`joe`) until all primitive types are reached. In this way, we have broken down the object graph into a tree view which fit into our internal structure nicely. Also note that all the primitive types will be stored inside the respective node's HashMap (e.g., `addr` will have `Zip` , `Street` , etc. stored there).

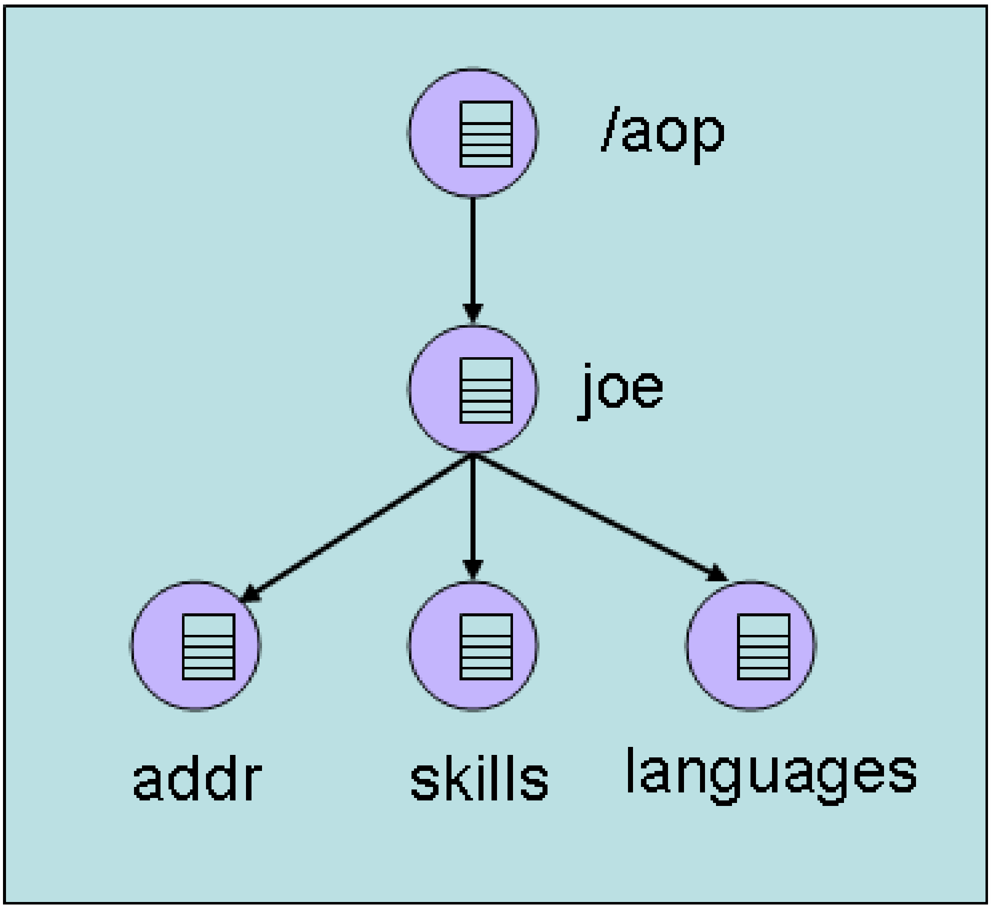


Figure 4.3. Object mapping by reachability

Here is a code snippet to demonstrate the object mapping by reachability feature that we just explained. Notice how a `Person` object (e.g., `joe`) that has complex object references will be mapped into the underlying cache store as explained above.

```
import org.jboss.cache.PropertyConfigurator;
import org.jboss.cache.aop.PojoCache;
import org.jboss.test.cache.test.standAloneAop.Person;
import org.jboss.test.cache.test.standAloneAop.Address;

PojoCache tree = new PojoCache();
PropertyConfigurator config = new PropertyConfigurator(); // configure tree
cache.
config.configure(tree, "META-INF/replSync-service.xml");

Person joe = new Person(); // instantiate a Person object named joe
joe.setName("Joe Black");
joe.setAge(31);
```

```
Address addr = new Address(); // instantiate a Address object named addr
addr.setCity("Sunnyvale");
addr.setStreet("123 Albert Ave");
addr.setZip(94086);
joe.setAddress(addr); // set the address reference

tree.startService(); // kick start tree cache
tree.putObject("/aop/joe", joe); // add aop sanctioned object (and
sub-objects) into cache.
// since it is aspectized, use of plain get/set methods will take care
// of cache contents automatically.
joe.setAge(41);
```

Note that a typical `PojoCache` usage involves instantiating the `PojoCache` , configuring, and starting the cache instance. Then, a user creates the aspectized POJO that will be put into the cache using `putObject()` API.

In addition, PojoCache also supports get/set with parameter type of some `Collection` classes (i.e., `List` , `Map` , and `Set` ) automatically. For example, the following code snippet in addition to the above example will trigger PojoCache to manage the states for the `Languages` list as well. The following figure illustrates the node structure from the replicated GUI point of view. Details of Collection class support will be given later.

```
ArrayList lang = new ArrayList();
lang.add("Ensligh");
lang.add("Mandarin");
joe.setLanguages(lang);
```

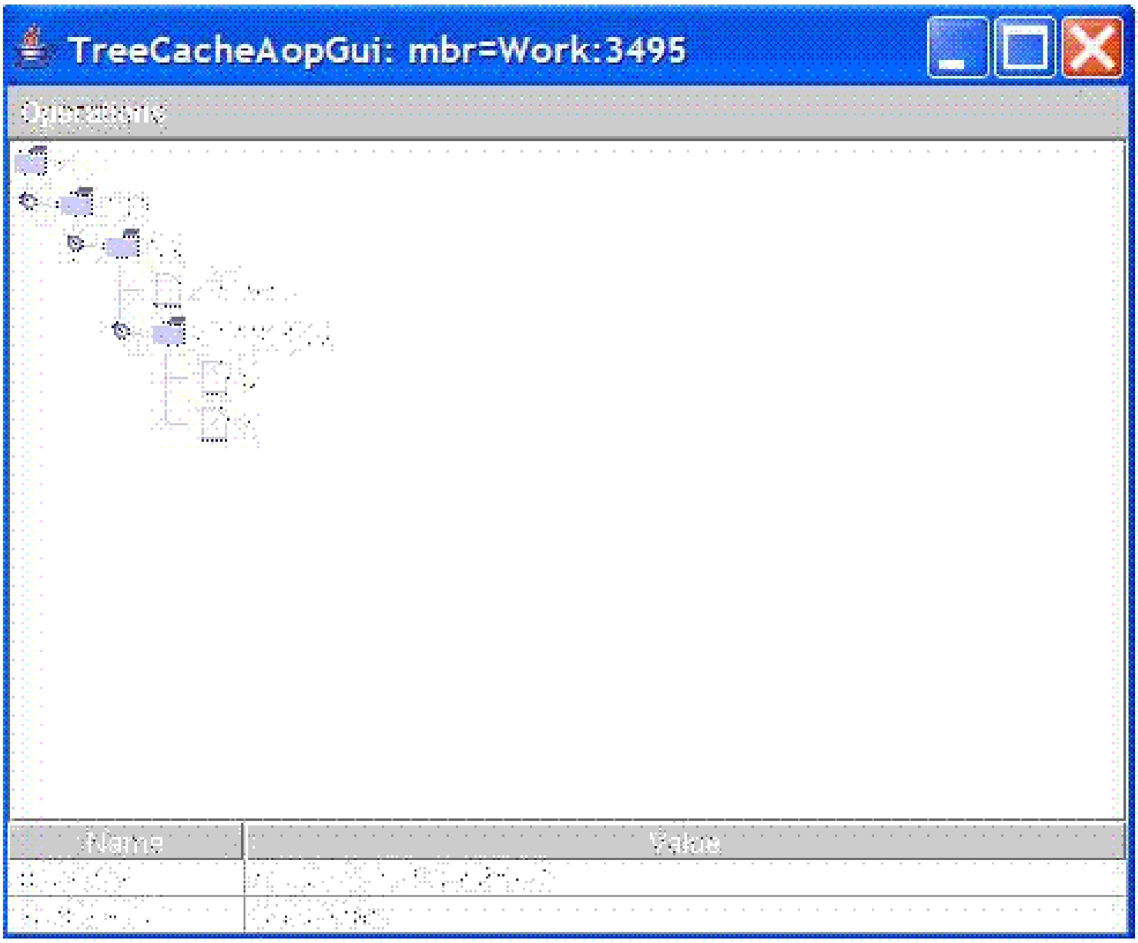


Figure 4.4. Schematic illustration of List class mapping

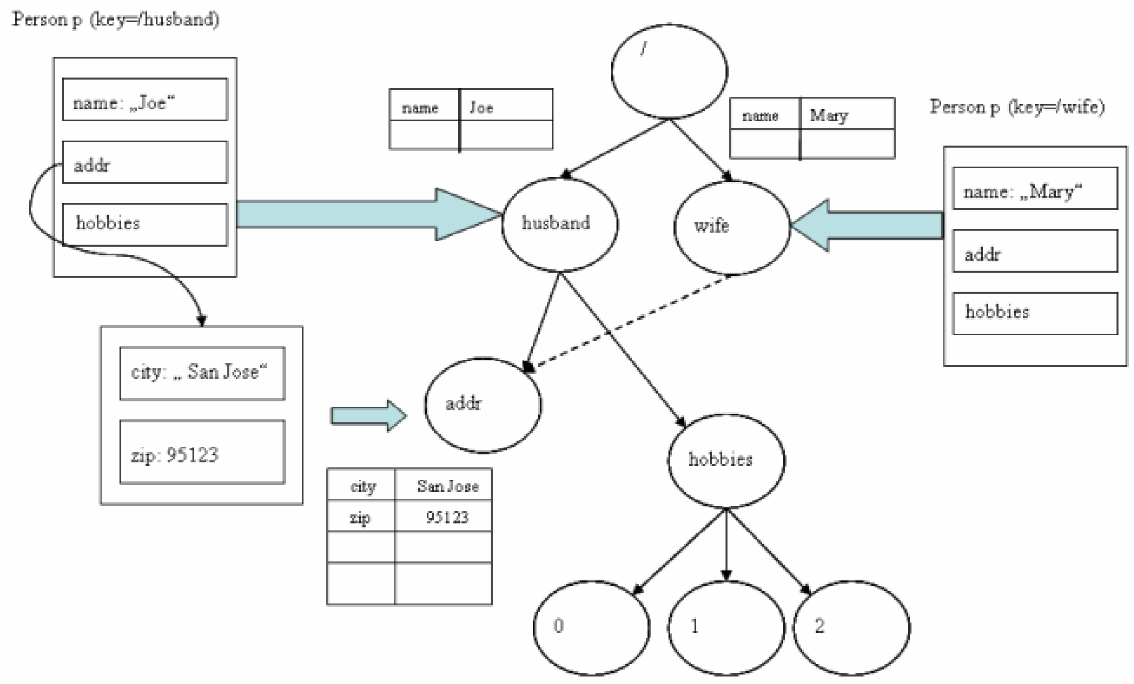
## 3. Object relationship management

Like we have mentioned, traditional cache system does not support object relationship management during serialization (be it to the persistent data store or replicated to the other in-memory nodes.) Examples of object relationship are like an address object is shared by members of the household, and a child-parent relationship. All these relationship will be lost once the objects are replicated or persisted. As a result, explicit mapping will be needed outside of the cache system to express the object relationship. PojoCache, in contrast, can manage object relationship transparently for users.

During the mapping process, we will check whether any of its associated object is multiple or circular referenced. A reference counting mechanism has been implemented associating with the `CacheInterceptor`. If a new object created in the cache referenced to another POJO, a referenced `fqn` will be stored there to redirect any query and update to the original node.

To look at one example, let's say that multiple `Persons` ("joe" and "mary") objects can own the same `Address` (e.g., a household). Graphically, here is what it will look like in the tree nodes.

Like we have covered in the previous section on the mapping by reachability, the POJO will map recursively into the cache. However, when we detect a multiple reference (in this case, the `Address`), we will keep track of the reference counting for the sub-object `addr`.



## Figure 4.5. Schematic illustration of object relationship mapping

In the following code snippet, we show programmatically the object sharing example.

```
import org.jboss.cache.PropertyConfigurator;
import org.jboss.cache.aop.PojoCache;
import org.jboss.test.cache.test.standAloneAop.Person;
import org.jboss.test.cache.test.standAloneAop.Address;

PojoCache tree = new PojoCache();
PropertyConfigurator config = new PropertyConfigurator(); // configure tree
cache.
config.configure(tree, "META-INF/replSync-service.xml");

Person joe = new Person(); // instantiate a Person object named joe
joe.setName("Joe Black");
joe.setAge(31);

Person mary = new Person(); // instantiate a Person object named mary
mary.setName("Mary White");
mary.setAge(30);

Address addr = new Address(); // instantiate a Address object named addr
addr.setCity("Sunnyvale");
addr.setStreet("123 Albert Ave");
addr.setZip(94086);
```

```
joe.setAddress(addr); // set the address reference
mary.setAddress(addr); // set the address reference

tree.startService(); // kick start tree
tree.putObject("/aop/joe", joe); // add aop sanctioned object (and
sub-objects) into cache.
tree.putObject("/aop/mary", mary); // add aop sanctioned object (and
sub-objects) into cache.

Address joeAddr = joe.getAddress();
Address maryAddr = mary.getAddress(); // joeAddr and maryAddr should be the
same instance

tree.removeObject("/aop/joe");
maryAddr = mary.getAddress(); // Should still have the address.
```

Notice that after we remove `joe` instance from the cache, `mary` should still have reference the same `Address` object in the cache store.

To further illustrate this relationship management, let's examine the Java code under a replicated environment. Imagine that we have two separate cache instances in the cluster now (`cache1` and `cache2`). Let's say, on the first cache instance, we put both `joe` and `mary` under cache management as above. Then, we failover to `cache2.` Here is the code snippet:

```

/**
 * Code snippet on cache2 during fail-over
 */
import org.jboss.cache.PropertyConfigurator;
import org.jboss.cache.aop.PojoCache;
import org.jboss.test.cache.test.standAloneAop.Person;
import org.jboss.test.cache.test.standAloneAop.Address;

PojoCache tree = new PojoCache();
PropertyConfigurator config = new PropertyConfigurator(); // configure tree
cache.
config.configure(tree, "META-INF/replSync-service.xml");

tree.startService(); // kick start tree
Person joe = tree.getObject("/aop/joe"); // retrieve the POJO reference.
Person mary = tree.getObject("/aop/mary"); // retrieve the POJO reference.

Address joeAddr = joe.getAddress();
Address maryAddr = mary.getAddress(); // joeAddr and maryAddr should be the
same instance!!!

maryAddr = mary.getAddress().setZip(95123);
int zip = joeAddr.getAddress().getZip(); // Should be 95123 as well instead
of 94086!
```

## 4. Object inheritance hierarchy

PojoCache preserves the POJO object inheritance hierarchy automatically. For example, if a `Student` extends `Person` with an additional field `year` (see POJO definition in the Appendix section), then once `Student` is put into the cache, all the class attributes of `Person` can be managed as well.

Following is a code snippet that illustrates how the inheritance behavior of a POJO is maintained. Again, no special configuration is needed.

```
import org.jboss.test.cache.test.standAloneAop.Student;

Student joe = new Student(); // Student extends Person class
joe.setName("Joe Black"); // This is base class attributes
joe.setAge(22); // This is also base class attributes
joe.setYear("Senior"); // This is Student class attribute

tree.putObject("/aop/student/joe", joe);

//...

joe = (Student)tree.putObject("/aop/student/joe");
Person person = (Person)joe; // it will be correct here
joe.setYear("Junior"); // will be intercepted by the cache
joe.setName("Joe Black II"); // also intercepted by the cache
```

## 5. Collection class proxy

The POJO classes that inherit from `Set` , `List` , and `Map` are treated as "aspectized" automatically. That is, users need not declare them "prepared" in the xml configuration file or via annotation. Since we are not allowed to instrument the Java system library, we will use a proxy approach instead. That is, when we encounter any Collection instance, we will:

- Create a Collection proxy instance and place it in the cache (instead of the original reference). The mapping of the Collection elements will still be carried out recursively as expected.
- If the Collection instance is a sub-object, e.g., inside another POJO, we will swap out the original reference with the new proxy one to promote transparent usage.

To obtain the proxy reference, users can then use another `getObject` to retrieve this proxy reference and use this reference to perform POJO operations.

Here is a code snippet that illustrates the usage of a Collection proxy reference:

```
List list = new ArrayList();
list.add("ONE");
list.add("TWO");

tree.putObject("/aop/list", list);
list.add("THREE"); // This won't intercept by the cache!
```

```
List proxyList = tree.getObject("/aop/list"; // Note that list is a proxy
reference
proxyList.add("FOUR"); // This will be intercepted by the cache
```

Here is another snippet to illustrate the dynamic swapping of the Collection reference when it is embedded inside another object:

```
Person joe = new Person();
joe.setName("Joe Black"); // This is base class attributes
ArrayList lang = new ArrayList();
lang.add("English");
lang.add("Mandarin");
joe.setLanguages(lang);
// This will map the languages List automatically and swap it out with the
proxy reference.
tree.putObject("/aop/student/joe", joe);
ArrayList lang = joe.getLanguages(); // Note that lang is a proxy reference
lang.add("French"); // This will be intercepted by the cache
```

As you can see, `getLanguages` simply returns the field reference that has been swapped out for the proxy reference counterpart.

Finally, when you remove a Collection reference from the cache (e.g., via `removeObject`), you still can use the proxy reference since we will update the in-memory copy of that reference during detachment. Below is a code snippet illustrating this:

```
List list = new ArrayList();
list.add("ONE");
list.add("TWO");

tree.putObject("/aop/list", list);
List proxyList = tree.getObject("/aop/list"); // Note that list is a proxy
reference
proxyList.add("THREE"); // This will be intercepted by the cache

tree.removeObject("/aop/list"); // detach from the cache
proxyList.add("FOUR"); // proxyList has 4 elements still.
```

## 5.1. Limitation

Use of Collection class in PojoCache helps you to track fine-grained changes in your collection fields automatically. However, current implementation has the follow limitation that we plan to address soon.

Currently, we only support a limited implementation of Collection classes. That is, we support APIs in List, Set, and Map. However, since the APIs do not stipulate of constraints like NULL key or value, it makes mapping of user instance to our proxy tricky. For example, ArrayList would allow NULL value and some other implementation would not. The Set interface maps to java.util.HashSet implementation. The List interface maps to java.util.ArrayList implementation. The Map interface maps to java.util.HashMap implementation.

Another related issue is the expected performance. For example, the current implementation is ordered, so that makes insert/delete from the Collection slow. Performance between Set, Map and List collections also vary. Adding items to a Set is slower than a List or Map, since Set does not allow duplicate entries.

# API

In PojoCache, there are 3 core APIs for for pojo management and one additional one for querying. They will be fully discussed here. Note that we have stressed that the management aspect of these APIs. This is because we expect most of the time, you only use these APIs to attach, detach, and retrieve the POJO from the cache system. After that, a user should operate on that POJO reference directly without worrying about replication and/or persistency aspects.

## 1. Attachment

```
/*
 * @param fqn The fqn instance to associate with the object in the cache.
 * @param pojo aop-enabled object to be inserted into the cache. If null,
 *             it will nullify the fqn node.
 * @throws CacheException
*/
Object putObject(Fqn fqn, Object pojo) throws CacheException;
```

Calling this Api will put your POJO into the cache management under `fqn` where `fqn` is a user-specified fully qualified name (`FQN`) to store the node in the underlying cache, e.g., "`/aop/joe`". The `pojo` is the object instance to be managed by `PojoCache`.

The requirement for `pojo` is that it must have been instrumented or implement the `Serializable` interface. An object is instrumented by JBossAop if declared either from an xml file or from annotation. More details on this will come later.

When the POJO is only serializable, PojoCache will simply treat it as an opaque "primitive" type. That is, it will simply store it without mapping the object's field into cache. Replication is done on the object wide level and therefore no fine-grained replication can be obtained.

If `pojo` has sub-objects, e.g., it has fields that are non-primitive type, this call will issue `putObject` recursively until all object tree are traversed (mapping by reachability). In addition, if you put `pojo` in multiple times, it will simply returns the original object reference right away. Furthermore, if this POJO has been referenced multiple times, e.g., referenced from other POJO or circular reference, this Api will handle the appropriate reference counting.

The return value after the call is the existing object under fqn (if any). As a result, a successful call will replace that old value with pojo instance, if it exists. Note that a user will only need to issue this call once for each POJO (think of it as attaching POJO to cache management). Once it is executed, `PojoCache` will assign an interceptor for the pojo instance and its sub-objects.

## 2. Detachment

```
/**
 * Remove aop-enabled object from the cache. After successful call of this
API, the returning
```

```
  * POJO is no longer managed by the cache.
  *
  * @param fqn Instance that associates with this node.
  * @return Original POJO stored under this node.
  * @throws CacheException
  */
Object removeObject(Fqn fqn) throws CacheException;
```

This call will detach the POJO from the cache management by removing the contents under fqn and return the POJO instance stored there (null if it doesn't exist). After successfully call, any POJO operation on the returned POJO reference is not intercepted by the cache system anymore. So in essence, you will use this Api should you decide that the POJO under `fqn` is no longer necessary. For example, if a POJO is no longer in context, you will need explicitly invoke this Api otherwise it won't get garbage collected by the VM. Note this call will also remove everything stored under `fqn` even if you have put other plain cache data there.

## 3. Query

```

/**
   * Retrieve the Pojo from the cache. Return null if object does not exist
in the cache.
   *
   * @param fqn Instance that associates with this node.
   * @return Current POJO value. Null if does not exist.
   * @throws CacheException
   */
Object getObject(Fqn fqn) throws CacheException;
```

This call will return the current object content located under `fqn`. This method call is useful when you don't have the exact POJO reference. For example, when you fail over to the replicated node, you want to get the object reference from the replicated cache instance. In that case, PojoCache will create a new Java object if it does not exist and then add the cache interceptor such that every future access will be in sync with the underlying cache store.

```

/**
  * Query all managed pojo objects under the fqn recursively. Note that this
will not return
  *the sub-object POJOs, e.g., if Person has a sub-object of Address, it
won't return Address
  *pojo. Note also that this operation is not thread-safe now. In addition,
it assumes
  *that once a pojo is found with a fqn, no more pojo is stored under the
children of the fqn.
  *That is, we don't mixed the fqn with different POJOs.
  * @param fqn The starting place to find all POJOs.
  * @return Map of all POJOs found with (fqn, pojo) pair. Return size of 0,
if not found.
  * @throws CacheException
  */
public Map findObjects(Fqn fqn) throws CacheException;
```

This call will return all the managed POJOs under cache with a base Fqn name. It is recursive, meaning that it will traverse all the sub-trees to find the POJOs under that base. For example, if you specify the fqn to be root, e.g., `"/"` , then it will return all the managed POJOs under the cache.

Note also that this operation is currently not thread-safe. In addition, it assumes that once a pojo is found with a fqn, no more pojo is stored under the children of the fqn. That is, we don't mixed the fqn with different POJOs.

# Configuration

Since PojoCache inherits from TreeCache, the xml configuration file attributes are almost identical to that of the latter one. Attributes such as replication mode, transaction manager, eviction policy, cache loader, and JGroups stack, for example, are still the same. There are two differences, however, when using the xml file--- configuring as a MBean service and eviction policy.

## 1. PojoCache MBean service

PojoCache can also be deployed as a MBean service under JBoss Application Server. However, you will need to use the correct class to instantiate. For example, this is the code snippet for the MBean attribute in the xml file:

```
<mbean code="org.jboss.cache.aop.PojoCache"
name="jboss.cache:service=PojoCache">
```

You can modify the object service name to your liking, of course.

## 2. PojoCache eviction policy

PojoCache also provides an eviction policy, `org.jboss.cache.aop.eviction.AopLRUPolicy` , that is a subclass of `org.jboss.cache.eviction.LRUPolicy` (with the same configuration parameters). The reason we need a distinctive implementation is because eviction in PojoCache is quite different from the regular TreeCache. In the plain cache world, a unit is a FQN node, while in the aop world, the concept of a unit is an object (which can have multiple nodes and children nodes!).

In addition, once a user obtains a POJO reference, everything is supposed to be transparent, e.g., cache retrieval and update operations. But if an object is evicted, that means there is no `CacheInterceptor` for the POJO, and the contents are not intercepted by the cache. Instead, every operation access will be channeled to the in-memory reference. So all operations will succeed but then a user has no way of knowing that it is merely updating the in-memory reference!

To remedy this problem, we currently requires that eviction policy is used in combination of a cache loader to persist the data. It can either be fully persistency or passivation (e.g., only persist when it is evicted). In this way, when the node is not available it will be retrieved from the persistent store. The downside is that the POJO won't be transient, e.g., it is persistent all time unless a specific user-based `removeObject` is called.

**Chapter 7.**

# Instrumentation

In this chapter, we explain how to instrument (or "aspectize") the POJOs via JBossAop. There are two steps needed by JBossAop: 1) POJO declaration, 2) instrumentation. But depends on the JDK and instrumentation mode that you are using, you may not need to pre-process your POJO at all. That is, if you use JDK5.0 and load-time mode, then all you need to do is annotation your POJO (or declare it in a xml file). This makes your PojoCache programming nearly transparent.

For the first step, since we are using the dynamic Aop feature, a POJO is only required to be declared "prepare". Basically, there are two ways to do this: either via explicit xml or annotation (new since release 1.2.3.)

As for the second step, either we can ask JBossAop to do load-time (through a special class loader, so-called loadtime mode) or compile-time instrumentation (use of an aopc pre-compiler, so-called precompiled mode)

## 1. XML descriptor

To declare a POJO via XML configuration file, you will need a `META-INF/jboss-aop.xml` file located under the class path. JBossAOP framework will read this file during startup to make necessary byte code manipulation for advice and introduction. Or you can pre-compile it using a pre-compiler called `aopc` such that you won't need the XML file during load time. JBossAop provides a so-called `pointcut` language where it consists of a regular expression set to specify the interception points (or `jointpoint` in aop language). The jointpoint can be constructor, method call, or field. You will need to declare any of your POJO to be "prepared" so that AOP framework knows to start intercepting either method, field, or constructor invocations using the dynamic Aop.

For PojoCache, we only allow all the fields (both read and write) to be intercepted. That is, we don't care for the method level interception since it is the state that we are interested in. So you should only need to change your POJO class name. For details of the pointcut language, please refer to JBossAop.

The standalone `JBossCache` distribution package provides an example declaration for the tutorial classes, namely, `Person` and `Address` . Detailed class declaration for `Person` and `Address` are provided in the Appendix section. But here is the snippet for `META-INF/jboss-aop.xml` :

```
<aop>
  <prepare expr="field(*
org.jboss.test.cache.test.standAloneAop.Address->*)" />
  <prepare expr="field(*
$instanceof{org.jboss.test.cache.test.standAloneAop.Person}->*)" />
</aop>
```

Detailed semantics of `jboss-aop.xml` can again be found in JBossAop. But above statements basically declare all field read and write operations in classes `Address` and `Person` will be "prepared" (or "aspectized"). Note that:

- The wildcard at the end of the expression signifies all fields in the POJO
- You can potentially replace specific class name with wildcard that includes all the POJOs inside the same package space
- The `instanceof` operator declares any sub-type or sub-class of the specific POJO will also be "aspectized". For example, if a `Student` class is a subclass of `Person` , JBossAop will automatically instrument it as well!
- We intercept the field of all access levels (i.e., `private` , `protected` , `public` , etc.) The main reason being that we consider all fields as stateful data. However, we can relax this requirement in the future if there is a use case for it.
- We don't intercept field modifiers of `static` , `final` , and `transient` though. That is, field with these modifiers are not stored in cache and is not replicated either. If you don't want your field to be managed by the cache, you can declare them with these modifiers, e.g., transient.

## 2. Annotation

Annotation is a new feature in Java 5.0 that when declared can contain metadata at compile and run time. It is well suited for aop declaration since there will be no need for external metadata xml descriptor. In order to support Java 1.4 as well, JBossAop has also used a xdoclet type annotation syntax, and as a result will need an annotation pre-compiler.

Note that PojoCache starts to support the use of annotation since release 1.2.3 with Java 1.4, and release 1.3 with Java 5.0.

### 2.1. JDK1.4

In order to provide maximum transparency, we have created two aop marker interfaces: `@@org.jboss.cache.aop.AopMarker` and `@@org.jboss.cache.aop.InstanceOfAopMarker`. The first interface is used to declare a regular POJO, while the latter one is used to declare the POJO and its sub-classes or sub-types.

Finally, to support annotation (to simplify user's development effort), the JBossCache distribution ships with a `jboss-aop.xml` under the `resources` directory. When you use the annotation precompiler, this file needs to be in the class path. For example, please refer to the `annoc` Ant build target. For reference, here is the content of that `jboss-aop.xml` :

```
                  <aop>
                    <prepare expr="field(*
  @@org.jboss.cache.aop.AopMarker->*)" />
                    <prepare expr="field(*
  $instanceof{@@org.jboss.cache.aop.InstanceOfAopMarker}->*)" />
                  </aop>
```

Basically, it simply states that any annotation with both marker interfaces will be "aspectized" accordingly.

Here is two code snippets that illustrate the declaration of both types through 1.4 style annotation:

```
        /**
        * @@org.jboss.cache.aop.AopMarker
        */
        public class Address {...}
```

The above declaration will instrument the class `Address`, and

```
        /**
        * @@org.jboss.cache.aop.InstanceOfAopMarker
        */
        public class Person {...}
```

The above declaration will instrument the class `Person` and all of its sub-classes. That is, if `Student` sub-class from `Personal`, then it will get instrumented automatically without further annotation declaration.

Note that the simplicity of declaring POJOs through annotation and its marker interfaces (although you will need to use the annotation pre-compiler.) See the build.xml target `annoc` for example.

## 2.2. JDK5.0

The JDK5.0 annotation is similar to the JDK1.4 counterpart except the annotation names themselves are different, e.g., the two annotations are: `@org.jboss.cache.aop.annotation.PojoCacheable` and `@org.jboss.cache.aop.annotation.InstanceOfPojoCacheable.` For example, when using JDK5.0 annotation, instead of using `@@org.jboss.cache.aop.AopMarker` you will use `@org.jboss.cache.aop.annotation.PojoCacheable` instead. In the distribution, under `examples/PojoCache/annotated50`, there is an example of using JDK50 annotation. Note that we have decided to use different annotation naming convention between JDK1.4 and 5.0.

## 2.3. JDK5.0 field level annotations

In Release 1.4, we have added two additional field level annotations for customized behavior. The first one is `@org.jboss.cache.aop.annotation.Transient`. When applied to a field variable, it has the same effect as the Java language `transient` keyword. That is, PojoCache won't put this field into cache management (and therefore no replication).

The second one is `@org.jboss.cache.aop.annotation.Serializable`, when applied to a field varaiable, PojoCache will treat this variable as `Serializable`, even when it is `PojoCacheable`. However, the field will need to implement the `Serializable` interface such that it can be replicated.

Here is a code snippet that illustrates usage of these two annotations. Assuming that you have a Gadget class:

```
public class Gadget
{
   // resource won't be replicated
   @Transient Resource resource;
   // specialAddress is treated as a Serializable object but still has
object relationship
   @Serializable SpecialAddress specialAddress;
   // other state variables
}
```

Then when we do:

```
   Gadget gadget = new Gadget();
   Resource resource = new Resouce();
   SepcialAddress specialAddress = new SpecialAddress();

   // setters
   gadget.setResource(resource);
   gadget.setSpecialAddress(specialAddress);

   cache1.putObject("/gadget", gadget); // put into PojoCache management

   Gadget g2 = (Gadget)cache2.getObject("/gadget");
   // retrieve it from another cache instance
   g2.getResource();
   // This is should be null becuase of @Transient tag so it is not
replicated.

   SepcialAddress d2 = g2.getSpecialAddress();
   d2.setName("inet");
   // This won't get replicated automatically because of @Serializable tag
   ge.setSpecialAddress(d2);
   // Now this will.
```

## 3. Weaving

As already mentioned, a user can use the aop precompiler (`aopc`) to precompile the POJO classes such that, during runtime, there is no additional system class loader needed. The precompiler will read in `jboss-aop.xml` and weave the POJO byte code at compile time. This is a convenient feature to make the aop less intrusive.

Below is an Ant snippet that defines the library needed for the various Ant targets that we are listing here. User can refer to the `build.xml` in the distribution for full details.

```
   <path id="aop.classpath"/>
       <fileset dir="${lib}"/>
          <include name="**/*.jar" //>
          <exclude name="**/jboss-cache.jar" //>
          <exclude name="**/j*unit.jar" //>
          <exclude name="**/bsh*.jar" //>
       </fileset/>
   </path/>
```

## 3.1. Ant target for running load-time instrumentation using specialized class loader

Below is the code snippet for the Ant target that is doing loadtime instrumentation through a special classloader. Basically, the Ant target `one.test.aop` using the bootclass loader generated by the Ant target `generateClassLoader` to run an individual test. Note that since JBossAop 1.3, a new `GenerateInstrumentedClassLoader` has been used since the previous `SystemClassLoader` is error prone.

```
   <target name="generateClassLoader" description=
           "Generate a new modified class loader so we can perform load time
instrumentation">
      <property name="build.bootclasspath"
value="${output}/gen-bootclasspath"/>
      <java classname="org.jboss.aop.hook.GenerateInstrumentedClassLoader">
         <classpath>
            <path refid="aop.classpath"/>
         </classpath>
         <arg value="${build.bootclasspath}"/>
      </java>
      <path id="bootclasspath">
         <pathelement location="${build.bootclasspath}"/>
         <path refid="aop.classpath"/>
      </path>
      <property name="bootclasspath" refid="bootclasspath"/>
   </target>

   <!-- eg. ant run.examples
-Dtest=org.jboss.test.cache.test.local.NoTxUnitTestCase -->
   <target name="one.test.aop" depends="compile, generateClassLoader"
   description="run one junit test case.">
      <junit printsummary="yes" timeout="${junit.timeout}" fork="yes">
         <jvmarg
```

```
value="-Djboss.aop.path=${output}/etc/META-INF/jboss-aop.xml"/>
         <jvmarg value="-Xbootclasspath/p:${bootclasspath}"/>
         <!-- jvmarg value="-Dbind.address=${bind.address}"/ -->
         <classpath path="${output}/etc" />
         <sysproperty key="log4j.configuration"
value="file:${output}/etc/log4j.xml" />
         <classpath refid="lib.classpath"/>
         <classpath path="${build}"/>
         <formatter type="xml" usefile="true"/>
         <test name="${test}" todir="${reports}"/>
      </junit>
  </target>
```

If you are running JDK5.0, you can also use the `javaagent` option that does not require a separate Classloader. Here are the ant snippet from `one-test-aop50`, for example.

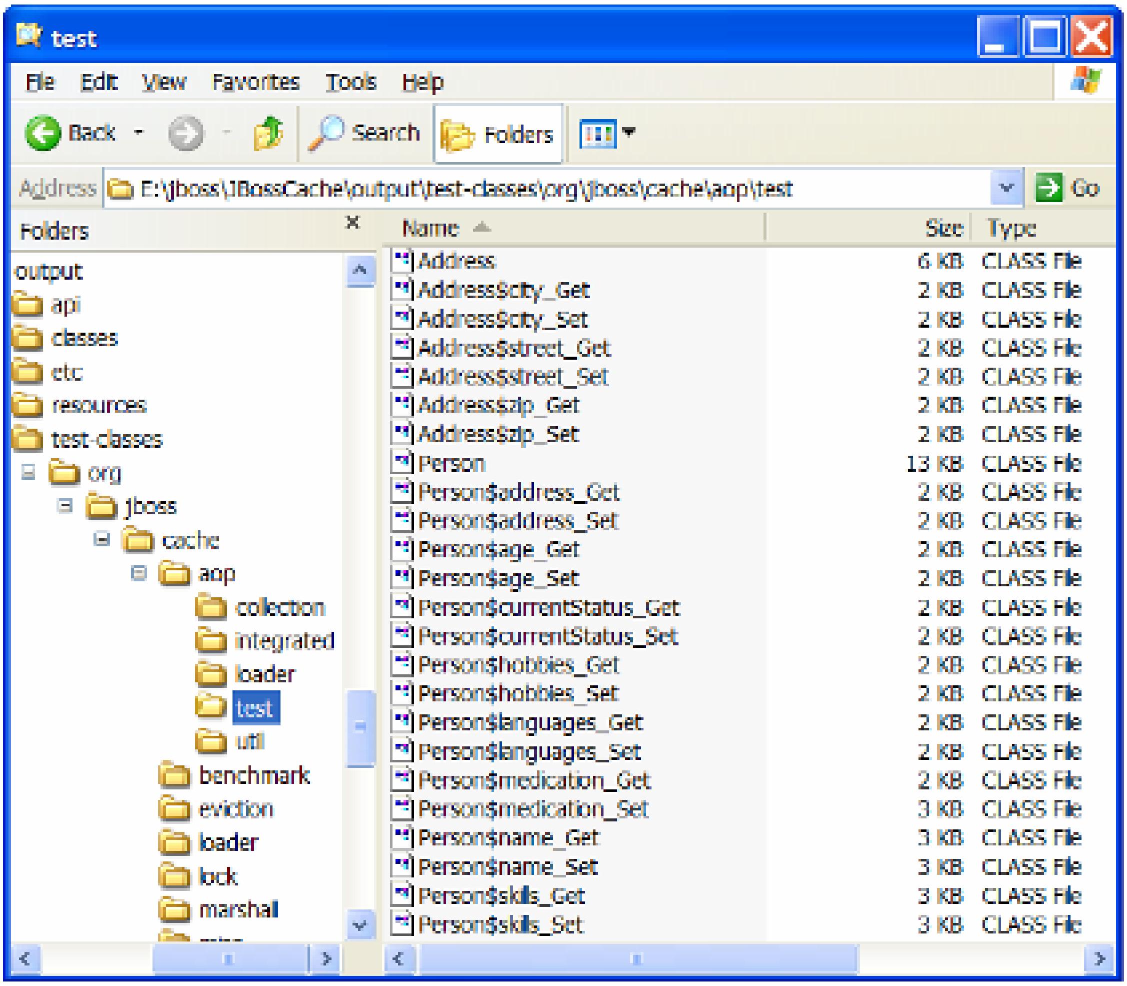
```
   <target name="one.test.aop50" depends="compile, generateClassLoader"
   description="run one junit test case.">
     <junit printsummary="yes" timeout="${junit.timeout}" fork="yes">
        <jvmarg value="-Djboss.aop.path=${output}/resources/jboss-aop.xml"/>
        <jvmarg value="-javaagent:${lib-50}/jboss-aop-jdk50.jar"/>
        <classpath path="${output}/etc" />
        <sysproperty key="log4j.configuration"
value="file:${output}/etc/log4j.xml" />
        <classpath refid="lib.classpath.50"/>
        <classpath refid="build.classpath.50"/>
        <formatter type="xml" usefile="true"/>
        <test name="${test}" todir="${reports}"/>
     </junit>
   </target>
```

## 3.2. Ant target for aopc

Below is the code snippet for the `aopc` Ant target. Running this target will do compile-time weaving of the POJO classes specified.

```
   <taskdef name="aopc" classname="org.jboss.aop.ant.AopC"
classpathref="aop.classpath"/>
   <target name="aopc" depends="compile" description="Precompile aop class">
      <aopc compilerclasspathref="aop.classpath" verbose="true">
         <src path="${build}"/>
         <include name="org/jboss/cache/aop/test/**/*.class"/>
         <aoppath path="${output}/etc/META-INF/jboss-aop.xml"/>
         <classpath path="${build}"/>
         <classpath refid="lib.classpath"/>
      </aopc>
   </target>
```

Below is a snapshot of files that are generated when aopc is applied. Notice that couple extra classes have been generated because of `aopc`.



Address E:\jboss\JBossCache\output\test-classes\org\jboss\cache\aop\test

| Name | Size | Type |
|---|---|---|
| Address | 6 KB | CLASS File |
| Address$city_Get | 2 KB | CLASS File |
| Address$city_Set | 2 KB | CLASS File |
| Address$street_Get | 2 KB | CLASS File |
| Address$street_Set | 2 KB | CLASS File |
| Address$zip_Get | 2 KB | CLASS File |
| Address$zip_Set | 2 KB | CLASS File |
| Person | 13 KB | CLASS File |
| Person$address_Get | 2 KB | CLASS File |
| Person$address_Set | 2 KB | CLASS File |
| Person$age_Get | 2 KB | CLASS File |
| Person$age_Set | 2 KB | CLASS File |
| Person$currentStatus_Get | 2 KB | CLASS File |
| Person$currentStatus_Set | 2 KB | CLASS File |
| Person$hobbies_Get | 2 KB | CLASS File |
| Person$hobbies_Set | 2 KB | CLASS File |
| Person$languages_Get | 2 KB | CLASS File |
| Person$languages_Set | 2 KB | CLASS File |
| Person$medication_Get | 2 KB | CLASS File |
| Person$medication_Set | 3 KB | CLASS File |
| Person$name_Get | 3 KB | CLASS File |
| Person$name_Set | 3 KB | CLASS File |
| Person$skills_Get | 3 KB | CLASS File |
| Person$skills_Set | 3 KB | CLASS File |

Figure 7.1. Classes generated after aopc

## 3.3. Ant target for annotation compiler

Below is the code snippet for the `annoc` Ant target. You run this step if you are using the JDK1.4 annotation. After this step is successfully run, you decide either to use compile-time or load-time mode of weaving.

```
    <!-- pre-compile directory with annotationc using jdk1.4 -->
    <target name="annoc" depends="compile" description="Annotation
  precompiler for aop class">
       <!-- Define a new ant target. This is the 1.4 annotation pre-compiler.
       After running this step, you still need to run the aopc step again,
       if you are not using system class loader.
       -->
        <taskdef name="annotationc" classname="org.jboss.aop.ant.AnnotationC"
```

```
         classpathref="aop.classpath"/>
         <annotationc compilerclasspathref="aop.classpath" bytecode="true">
            <classpath refid="lib.classpath"/>
            <classpath path="${build}"/>
            <!--System wide jboss-aop.xml is located here. -->
            <classpath path="${output.resources.dir}"/>
            <src path="${source}"/>
            <include name="org/jboss/cache/aop/test/**/*.java"/>
         </annotationc>
     </target>
```

# TroubleShooting

We have maintained a *PojoCache wiki troubleshooting page*[1]. Please refer it first. We will keep adding troubleshooting tips there.

All the current outstanding issues are documented in *JBossCache Jira page*[2] . Please check it for details. If you have discovered additional issues, please report it there as well.

---

[1] http://wiki.jboss.org/wiki/Wiki.jsp?page=PojoCacheTroubleshooting

[2] http://jira.jboss.com/jira/secure/BrowseProject.jspa?id=10051

# Appendix

## 1. Example POJO

The example POJO classes used for are: `Person`,`Student`, and `Address`. Below are their defintion (note that neither class implements `Serializable` ).

```
public class Person {
   String name=null;
   int age=0;
   Map hobbies=null;
   Address address=null;
   Set skills;
   List languages;

   public String getName() { return name; }
   public void setName(String name) { this.name=name; }

   public int getAge() { return age; }
   public void setAge(int age) { this.age = age; }

   public Map getHobbies() { return hobbies; }
   public void setHobbies(Map hobbies) { this.hobbies = hobbies; }

   public Address getAddress() { return address; }
   public void setAddress(Address address) { this.address = address; }

   public Set getSkills() { return skills; }
   public void setSkills(Set skills) { this.skills = skills; }

   public List getLanguages() { return languages; }
   public void setLanguages(List languages) { this.languages = languages; }
}
```

```
public class Student extends Person {
   String year=null;

   public String getYear() { return year; }
   public void setYear(String year) { this.year=year; }
}
```

```
public class Address {
   String street=null;
   String city=null;
   int zip=0;

   public String getStreet() { return street; }
   public void setStreet(String street) { this.street=street; }
   ...
}
```

## 2. Sample configuration xml

Below is a sample xml configuration for PojoCache. Today, it uses the same configuration option as that of TreeCache super-class except the MBean service class (if you are deploying it under JBoss).

```
<?xml version="1.0" encoding="UTF-8" ?>

<server>
<mbean code="org.jboss.cache.aop.PojoCache"
name="jboss.cache:service=PojoCache">
<depends>jboss:service=TransactionManager</depends>

<!--         Configure the TransactionManager -->
<attribute name="TransactionManagerLookupClass">
org.jboss.cache.DummyTransactionManagerLookup</attribute>

<!--             Isolation level : SERIALIZABLE
                            REPEATABLE_READ (default)
                            READ_COMMITTED
                            READ_UNCOMMITTED
                            NONE
-->
<attribute name="IsolationLevel">REPEATABLE_READ</attribute>

<!--              Valid modes are LOCAL, REPL_ASYNC and REPL_SYNC -->
<attribute name="CacheMode">REPL_SYNC</attribute>

<!--         Just used for async repl: use a replication queue -->
<attribute name="UseReplQueue">false</attribute>

<!--             Replication interval for replication queue (in ms) -->
<attribute name="ReplQueueInterval">0</attribute>

<!--             Max number of elements which trigger replication -->
<attribute name="ReplQueueMaxElements">0</attribute>

<!--  Name of cluster. Needs to be the same for all clusters, in order
           to find each other
-->
<attribute name="ClusterName">TreeCache-Cluster</attribute>

<!--  JGroups protocol stack properties. Can also be a URL,
           e.g. file:/home/bela/default.xml
         <attribute name="ClusterProperties"></attribute>
-->
<attribute name="ClusterConfig">

<config>
<!--  UDP: if you have a multihomed machine,
              set the bind_addr attribute to the appropriate NIC IP address,
              e.g bind_addr="192.168.0.2"
-->
<!--  UDP: On Windows machines, because of the media sense feature
               being broken with multicast (even after disabling media
sense)
               set the loopback attribute to true
-->
<UDP mcast_addr="228.1.2.3" mcast_port="48866" ip_ttl="64" ip_mcast="true"
mcast_send_buf_size="150000" mcast_recv_buf_size="80000"
ucast_send_buf_size="150000" ucast_recv_buf_size="80000"
loopback="false" />
<PING timeout="2000" num_initial_members="3" up_thread="false"
down_thread="false" />
<MERGE2 min_interval="10000" max_interval="20000" />
<FD_SOCK />
<VERIFY_SUSPECT timeout="1500" up_thread="false" down_thread="false" />
<pbcast.NAKACK gc_lag="50" retransmit_timeout="600,1200,2400,4800"
max_xmit_size="8192"
up_thread="false" down_thread="false" />
<UNICAST timeout="600,1200,2400" window_size="100" min_threshold="10"
down_thread="false" />
<pbcast.STABLE desired_avg_gossip="20000" up_thread="false"
down_thread="false" />
<FRAG frag_size="8192" down_thread="false" up_thread="false" />
<pbcast.GMS join_timeout="5000" join_retry_timeout="2000" shun="true"
print_local_addr="true" />
<pbcast.STATE_TRANSFER up_thread="true" down_thread="true" />
</config>
</attribute>

<!--         Whether or not to fetch state on joining a cluster -->
<attribute name="FetchStateOnStartup">true</attribute>

<!--             The max amount of time (in milliseconds) we wait until the
            initial state (ie. the contents of the cache) are retrieved from
            existing members in a clustered environment

-->
<attribute name="InitialStateRetrievalTimeout">5000</attribute>

<!--             Number of milliseconds to wait until all responses for a
            synchronous call have been received.
-->
<attribute name="SyncReplTimeout">15000</attribute>

<!--  Max number of milliseconds to wait for a lock acquisition -->
<attribute name="LockAcquisitionTimeout">10000</attribute>

<!--  Name of the eviction policy class. -->
<attribute name="EvictionPolicyClass" />
</mbean>
</server>
```

# Index