OBJECT MAPPING WITH JAVA ANNOTATIONS

by

Clint Michael Frederickson

A thesis submitted in partial fulfillment
of the requirements for the degree

of

Master of Science

in

Computer Science

MONTANA STATE UNIVERSITY
Bozeman, Montana

April 2005

©COPYRIGHT

by

Clint Michael Frederickson

2005

All Rights Reserved

APPROVAL

of a thesis submitted by

Clint Michael Frederickson

This thesis has been read by each member of the thesis committee and has been found to be satisfactory regarding content, English usage, format, citations, bibliographic style, and consistency, and is ready for submission to the College of Graduate Studies.

Dr. Gary Harkin

Approved for the Department of Computer Science

Dr. Michael Oudshoorn

Approved for the College of Graduate Studies

Dr. Bruce R. McLeod

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a master's degree at Montana State University, I agree that the Library shall make it available to borrowers under rules of the Library.

If I have indicated my intention to copyright this thesis by including a copyright notice page, copying is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U. S. Copyright Law. Requests for permission for extended quotation from or reproduction of this thesis in whole or in parts may be granted only by the copyright holder.

Clint M. Frederickson

April, 2005

ACKNOWLEDGEMENTS

This work was sponsored by the U.S. D.O.E., Office of Energy Research, through INL under Contract DE-AC07-94ID13223.

Thanks are are also given to Dr. Gary Harkin, my advisor, for his work in reviewing and editing this thesis and Dan Wessol, for providing direction during my research. In addition I would also like to thank my wife, Starsha Frederickson, for her patience these past months, and for everything she did to allow me to work long hours in front of the computer. Lastly, I want to thank God, with whom, all things are possible.

TABLE OF CONTENTS

LIST OF FIGURES vii

ABSTRACT vii

1. INTRODUCTION 1

   Motivation 1
      Data Complexity and the Object-Relational Gap 1
      Application-Data Source Coupling 1
      Data Creep 1
      Previous Knowledge of Data Source specifics 2
   Previous Work 3
      Object-Relational Mapping 3
      Object Databases 4
   Tools 5
      Java's Metadata Facility 5
      JDBC 7
      Hibernate 7
      JDOM 8
      Apache Xerces2 8
      Ant 8
      JUnit 9

2. REQUIREMENTS 10

   Natural Integration with Object-Oriented Programming 10
   Extendibility 10
   Object Dependency 10
   Shallow Loading 11
   Concurrency 12
   Data Integrity 12
   Maintenance 12

3. IMPLEMENTATION 14

   Application Context 14
   Data Sources 14
      Transaction-Safe Data Sources 16
      Transaction-Unsafe Data Source 16
      Updatable Data Source 16
   Describing the Data 17

ClassInfo Annotation 17
PersistentFields Annotation 18
PersistentCollections Annotation 18
Modeling Inheritance 19
Getting and Setting Persistent Object Properties 19
Assigning Unique Identifiers 20
Architecture 22
The Data Layer 22
The Persistence Layer 24
AnnotationUtilities 24
The Persisters 24
Caching 25
Data Operations 26
MultiDatastore 26
Concurrency Pitfalls 26
DataAccessPath 26
Persist Operation 26
Load Operation 27
Remove Operation 28
Adding a New Data Source Persister 28

4. RESULTS 30

Integration with MINERVA 30
Performance 30
Shallow Loading 31
Using Hibernate with Relational Databases 32
XML JAR 32

5. FUTURE RESEARCH 34

Future Data Sources 34
More Advanced Caching 34
Object Remapping and Versioning 34
More Efficient and Extendible Shallow Loading 35
Graphical Frontend for Manipulating Datagraphs 35
Updatable XML Data Source 36
Using Annotations in JavaDoc 36
Handling Non-annotated Classes 36

REFERENCES CITED 38

APPENDIX A – DETAILED ANNOTATION PROPERTIES 40

LIST OF FIGURES

Figure Page

1. Data of the MINERVA Application 15

2. Persistence System Architecture 23

ABSTRACT

Java applications often need to store data in external data sources. Large amounts of time can be spent developing solutions to integrate specific data sources into the application. The process of mapping object-oriented data to data sources can lead to a fragile system that can not handle incremental changes to the data or the integration of new data sources cleanly.

Java 1.5 introduced a metadata facility called annotations into the Java language. Annotations can be used to describe data in a general way such that it can be mapped onto various types of data sources easily. The annotations are inspected at runtime by each data source persister, and a mapping is created.

Implementations of two persisters are given: one for relational databases, and one for XML. Other new persisters can easily be added to the system.

INTRODUCTION

Motivation

Data Complexity and the Object-Relational Gap

Applications have increasingly complicated data and often require a more sophisticated data storage system than simple flat files. A database is a solution, but the developer may face the obstacle of the object oriented data not mapping directly onto a relational database. We will investigate a solution to this problem using a general object-oriented data-mapping and persistence system, modeled using Java annotations and a set of interfaces. The persistence(storage) architecture discussed will map an application's data to various data sources for future storage and retrieval

Application-Data Source Coupling

Applications are often developed with a particular set of data sources in mind. As time goes on, new data sources may need to be added, or even replace old ones. If the code is tightly-coupled to the data source, these two operations can become quite tedious. If a dataclass could describe itself, accommodating a new data source would be a matter of developing a mapping from the object to the data source.

Data Creep

During the lifecycle of a project, data requirements are likely to change. These changes usually happen in small increments called data creep. Some data sources are inherently more flexible than others. For example, when using a relational database, adding, removing, or updating a field in an object will require at least one table to be altered. While altering a table is not terribly difficult in itself, several other tasks also need to be done:

1. Changing any database schema generation scripts

2. Handling previously inserted records and possibly generating new default values.

3. Possibly changing the code that reads and writes the object.

Even for a more forgiving data source, like XML, previously stored objects will be missing any newly added fields and the code to read and write the data may need adjustments, depending on the system design. If a DTD or XML schema document is being used, that will also need to be updated.

In the case of a self-describing dataclass, when a new field is added the dataclass could be remapped eliminating the need to change the code to read and write the object. The database schema, since it could be constructed on demand from the metadata, would also be updated. Altering the affected tables automatically and assigning default values, may still require hand editing.

Previous Knowledge of Data Source specifics

It is often the case that the dataclass developer may not be familiar with the specifics of using a particular data source. For example, if a relational database is being used, the developer may need knowledge of SQL to persist and retrieve objects. A system that would let the developer describe which parts of an object are important for persistence, as well as characteristics of each field, such as type, default value, etc. would be preferable. Java's annotations support this type of description very nicely.

Previous Work

Many tools have been developed in order to allow applications to more easily communicate with data sources. These technologies also attempt to free developers from the mire of data source specifics. The tools covered in this section are either directly used in our implementation or served as good ideas that could be built on.

Object-Relational Mapping

The idea behind Object-Relational Mapping (ORM) is to map [Java] objects to records in a relational database[1]. To understand ORM, it is beneficial to briefly consider the motivation behind it.

Object-Oriented programming (OOP) saw its beginnings in the 1970's, mostly powered by XEROX PARC's Smalltalk language[2]. Over time, many software engineers found their problems more easily modeled and solved with OOP. Relational

technology was also developed in the 1970's due to the work of IBM's Edgar Codd. Codd based his relational model on mathematics [3]; in his original paper "A Relational Model of Data for Large Shared Data Banks," he describes his work[3]:

> It provides a means of describing data with its natural structure only–that is, without superimposing any additional structure for machine representation purposes. Accordingly, it provides a basis for a high level data language which will yield maximal independence between programs on the one hand and machine representation on the other.

On one hand, the application developer has a superior software development paradigm, OOP, and on the other, an equally superior data storage technique, the relational model. Developers have long been trying to bridge this object-relational gap, and ORM is providing easier integration between the two technologies.

The Java world has no shortage of object-relational mappers. Some of the more popular ORM tools include Enterprise JavaBeans (EJB), Java Data Objects (JDO), and Hibernate. Each of these technologies defines a mechanism to allow Java objects to be persisted to a relational database. The Hibernate object-relational mapper will be covered in more detail later.

Object Databases

Object Databases (OODBMS) have also been proposed as a solution to the object-relational gap, by simply eliminating the need to translate between two systems of representation. OODMBS technology has been around since the 1980's[4], but has never been widely used. There is no consensus as to why object databases have not gained wider acceptance[5]. Some point to poor performance, no active standards

body (like ANSI SQL for relational databases), and the tight coupling to particular languages. In addition, they are fighting an uphill battle against the entrenched relational database which has been around for decades and has significant mindshare in the software development community. For this reason, a great deal of legacy data currently exists in relational databases, and migrating this data to a new storage paradigm would be difficult.

A full discussion of object databases is beyond the scope of this paper, but they have not become widely used and the industry doesn't seem to be moving in that direction. For that reason, they will not be considered further.

Tools

Java's Metadata Facility

With the release of JDK 1.5, Java introduced a metadata facility into the language called *annotations*. Metadata is not a new concept; it is simply data that describes data. At this time, XML files are widely used for this purpose. J2EE applications often use XML to define security policies to allow certain methods to be remotely accessible. This approach results in the code and metadata being separated into different files. If one is updated, and not the other, the application will (mysteriously) break. This problem may be avoided if the metadata and code were stored closer together.

There are also situations for which separate metadata files are not suitable. For example, consider a software developer who is fixing a time critical problem in his code and notices something odd going on in a non-related method. Without a metadata facility he will, at best, create an in-code comment concerning the aberration which will likely be forgotten about until someone else browses that method. What if he could intermingle metadata with his code and leave a @todo note which would be Emailed to all developers during the next build? Java's annotations are perfect for this scenario.

Although no tool is without faults, Java's annotation system gives us several advantages over a separate metadata file[6]:

Java annotations can be used by JavaDoc to enhance documentation. In our example above, the @todo annotation could be included in the JavaDoc by placing it above a method. As we will see later on, an @child annotation can be used to show how objects depend one another. This may also be useful to include in a project's JavaDoc.

Java annotations can be picked up at runtime using class reflection. Java's powerful reflection system allows code logic to be easily built around annotations. We make extensive use of metadata reflection in the AnnotationUtilities class which will be discussed further in the Implementation chapter.

Other advantages of Java annotations include IDE support for custom annotations, class and method versioning, and testing.

JDBC

JDBC is Java's facility for directly communicating with databases through SQL. Although its goal is to "provide cross-DBMS connectivity to a wide range of SQL databases[7]", real-world developers often find that its DBMS connectivity requires tweaking for each database. This is especially evident when storing binary objects. Each database vendor has its own Binary Large Object (BLOB) implementation, none of which fit exactly with the ANSI SQL specification.

JDBC is a necessary tool for any Java application wanting to communicate with relational databases. By leveraging the power of Hibernate (see the next section), nearly all the SQL needed is automatically generated, and does not have to be written by hand. Hibernate's database dialects also address JBDC's portability issues.

Hibernate

Hibernate will be used as a tool to assist our application in building a generic data source interface. Hibernate aims "to be a complete solution to the problem of managing persistent data in Java. It mediates the application's interaction with a relational database.[8]" It abstracts database specifics away from the persistence system by wrapping up all the information about a DBMS into database dialect objects. This frees the persistence system from being tied to a particular database due to non-standard implementations. Switching databases is a matter of switching the dialect. Most popular databases are supported, including Oracle, DB2, MySQL, PostgreSQL and Sybase.

Hibernate will generate a database schema, as well as SQL for inserting, updating and retrieving objects from the database using the metadata we define for an object. This will be accomplished though the mapping system discussed in the Implementation chapter.

JDOM

JDOM[9] is an API for manipulating XML documents in Java. It is easy to use, and integrates very naturally with Java, especially the Java collection framework. This makes it much easier to use with Java than either the SAX[10] or DOM[11] APIs. JDOM can use most any XML parser available, including the popular Apache Xerces[12] and Crimson[13] parsers.

JDOM will be utilized in two major places in our implementation. See the *Architecture* section for more details.

Apache Xerces2

Xerces is a "high performace, fully compliant" XML parser. This means that Xerces can read all XML 1.1 documents. It is often used in conjunction with a XML manipulation tool, such as JDOM. The Xerces parser will be used by Hibernate and directly by our application.

Ant

Ant[14] is build tool, much like make, but is pure Java and thus portable to all architectures that have a JVM. Ant build files are written in XML and are extremely

flexible. Predefined tasks exist for compiling, running, distributing, and unit testing (see JUnit section) Java projects. Custom tasks can also be defined and used.

JUnit

JUnit is a framework to assist developers in writing repeatable unit tests[15]. Unit testing allows small pieces of a system to be tested independently. Base tests typically test a single class, while test suites use the base tests to test larger portions of related code. This process can be repeated until whole subsystems or the entire system is being tested together. If a problem arises after code changes are make, locating the problem should be relatively easy, and should quickly narrow the error to a specific base test. This, of course, assumes that a sufficient number of quality tests have been written for the system.

During the implementation process, many tests were written to prove the correctness of the persistence system. The tests caught many bugs early, before they became a problem. As new features were added, the tests we expanded to test the new functionality.

REQUIREMENTS

This chapter will outline the goals of our object mapping with Java annotations implementation.

Natural Integration with Object-Oriented Programming

The system should integrate naturally with the object-oriented programming paradigm. It should not be awkward to describe objects; more complicated ideas, like inheritance and inner classes must be easily supported. Specific fields of an object should be selectable for use with the mapping and persistence services. Reasonable limitations may be imposed on these fields, but these limitations must not hinder the complexity or expressive power of the dataclass.

Extendibility

Implementation will be focused on relational databases, but should be easily extendible to other types of data sources such as XML files, XML databases, object databases, etc. The code must be designed in such a way that adding a new data source will not effect the mapping and persistence services to other data sources. Extending the system should be as simple as implementing a small set of interfaces.

Object Dependency

Sometimes objects are not physically linked together, but do in fact depend on one another. The metadata should allow object dependencies to be described and dealt with appropriately. This may mean imposing restrictions on when certain operations may be performed on specific objects. For example, if object A depends on object B, removing object B from the system will possibly leave A in an inconsistent state. These types of situations must be avoided.

Shallow Loading

Data objects can range from being very simple and small, to being very large and complex. Complex data objects, such as those containing large amounts of binary data, perhaps a set of images, can be troublesome for a persistence service due to difficultly of efficiently handling large binary objects. The time required to load these objects, and perhaps more importantly, the memory needed to store them, can be considerable. In the context of an image set, the required data is often the "header information," the image dimensions, number of bytes per pixel and so on. Therefore, to be efficient, the persistence service will offer the option to "shallow load" objects with large binary components. Shallow loading an object will load all fields of an object except the ones marked as lazy.

Hibernate takes the concept of shallow loading a bit further by allowing an object to be partially loaded, and then loading the rest of the object on demand. This is

called lazy loading. This more powerful feature will not be included in our implementation, but does present one area of future work.

Concurrency

The architecture of the persistence service must allow for concurrency. The service should be multi-threaded, and provide an application with a way to access multiple data sets at the same time. Safety of the concurrency will be largely determined by the type and setup of the data source being used and higher level locks put in place by the application itself. The persistence service itself will not be responsible for ensuring, for example, that multiple threads of execution do not attempt to save the same data concurrently to a data source.

Data Integrity

The data saved and loaded through the persistence system must maintain its integrity through the entire process. This does not mean that dataclasses can not be optimized for storage, but rather that when an object is loaded from a data source each field that was saved must remain unchanged from the original object. Care must be taken when converting some Java primitives, especially numerical values, for storage.

Maintenance

Once metadata is defined for a dataclass, making small changes and remapping onto a data source should not be difficult. Ideally, the process would look like:

1. Change the metadata for the targeted class

2. Run a 'generate mapping' ant task

3. Use dataclass and data source as usual

IMPLEMENTATION

Application Context

The application selected for the initial implementation of this data mapping strategy will be the MINERVA (Modality-Inclusive Environment for Radiotherapeutic Variable Analysis) project[16]. MINERVA is a patient-centric, multi-modal, radiation treatment planning system which may be used for planning and analyzing several radiotherapy modalities, either singly or combined, using common modality independent image and geometry construction and dose reporting and guiding.

MINERVA is a good candidate for implementation of a metadata-based object mapping system for several reasons:

(1) MINERVA has a sufficiently complex data graph (Figure 1) to warrent modeling.

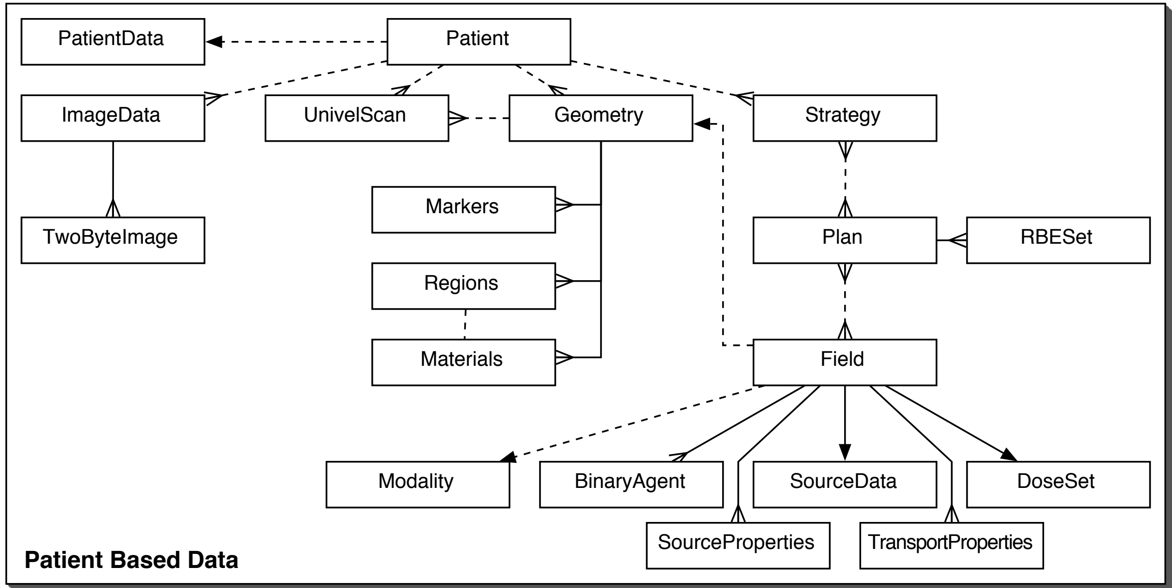
(2) MINERVA uses a combination of data sources for long term storage. A relational database is its primary data storage and future work will include adding other data sources, specifically XML files. XML databases have also been condsidered.

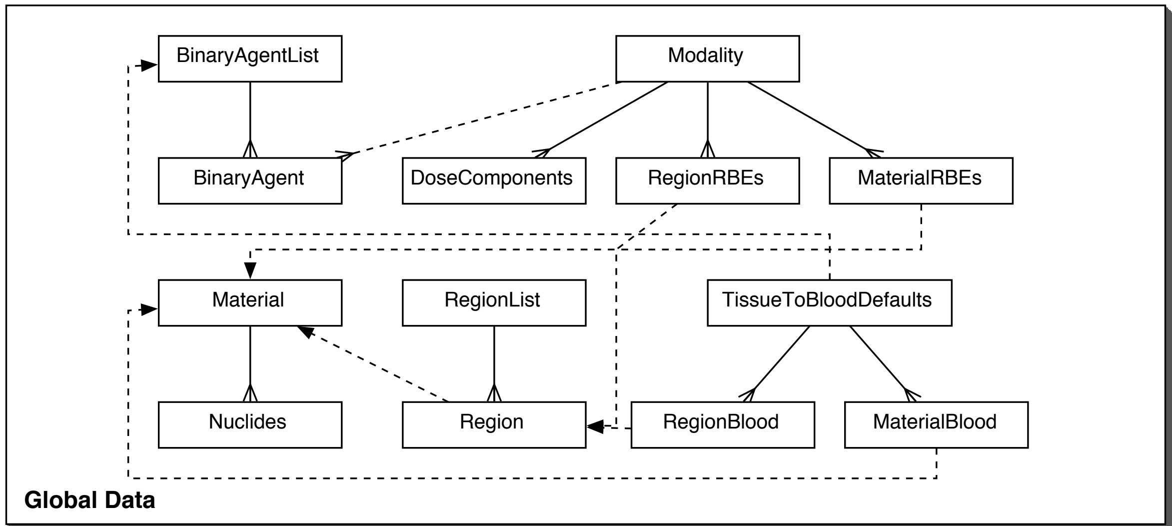
(3) The objects in need of storage are quite complex. A typical object often encapsulates other objects and large binary arrays which require efficient handling.

Data Sources

We will divide data sources into two main categories.

Tirade Data Hierarchy





Legend

- Included Directly
- References Another
- One to Many
- Many to Many

Figure 1. Data of the MINERVA Application.

Transaction-Safe Data Sources

A data source will be considered transaction-safe if it provides the capability to guarantee the integrity of the data source while performing save and remove operations. More specifically, the data source must be able to withstand interruption during a write operation, perhaps by hardware failure or user intervention, and be able to revert the data source back to the last known safe state.

This type of service is provided by most database management systems. The MySQL DBMS has been selected as our transaction-safe data source, although any JDBC compliment database that implements transactions should be sufficient.

Transaction-Unsafe Data Source

A data source will be considered transaction-unsafe if it does not meet the requirements outlined in the previous section. Most data sources, including flat files, fit into this category. XML will be used as an example of a transaction-unsafe data source in our implementation.

Updatable Data Source

Each data source is either updatable or non-updatable. Data stored in updatable data sources can be modified at any time whereas non-updatable data sources make data immutable after the data source is closed. In our implementation, data stored in relational databases is fully updatable. Data stored in XML JARS, however, is non-updatable after the JAR is closed. This means to change the data stored in an XML

JAR, it must be completely read in, and written out to a new JAR. A non-updatable data source serves as a convenient mechanism for archiving data, or transporting it between two updatable data sources. Updatable data sources are likely transaction-safe and non-updatable data sources are likely non-transaction safe, but this is not always the case.

Describing the Data

As mentioned previously, Java 1.5's annotations were chosen as the tool to model the data objects. We have designed three main types of class annotations. They are all processed at runtime, and are the basis for the entire persistence system.

ClassInfo Annotation

This annotation's purpose is to define how the object fits into the overall object graph, and how the persistence system can identify and process objects of this type. Each annotation property is described in Appendix A.

It is important to distinguish an object that *references* another object from an object that *encapsulates* another object. Consider two objects, A and B. If A references B, A does not contain B, but rather has some identifier that will uniquely identify B. In a one-to-many relationship, this is usually represented as a list of B identifier strings in A. If A encapsulates B, B will be directly included in A's class definition.

Example: A references B

```
class A
{
    List<String> bList; //eg. {"b1", "b2", "b3"};
}
```

Example: A encapsulates B

```
class A
{
    List<B> bList; //eg. {new B(), new B(), new B()}
}
```

If two classes are linked by *references* to one another, special properties should be set to show the relationship between the two classes. (See the parents() and children() properties in Appendix A for more details). If they are linked by *encapsulation*, the relationship is implicit and will be modeled using the PersistentFields and/or PersistentCollections annotations.

PersistentFields Annotation

This annotation is a list of PField annotations. A PField should be defined for each non-collection class field that needs to be stored and should include the information needed to map the object onto a data source. Each annotation property need not be used by every data source. Note that a PField definition will directly correspond to exactly one getter and setter in the class. See the PersisentFields section in Appendix A for detailed descriptions of the PField properties.

PersistentCollections Annotation

This annotation is a list of PCollection annotations. A PCollection should be defined for each collection that needs to be stored. Four types of collections are supported: java.util.List, java.util.Map, java.util.Set, and arrays. A collection must be homogenous, although polymorphism is fully supported. Note that a PCollection definition will directly correspond to exactly one getter and setter in the class. See the PersistentCollections section in Appendix A for detailed descriptions of the PCollection properties.

Modeling Inheritance

The description for a class should contain only the data defined directly in that class. Data defined further up the inheritance heirarchy should not be included, as it will be automatically included when the object is persisted. If a field or collection is declared in a superclass, the getter and setter may be overridden by a subclass, but the field or collection should nevertheless be annotated in the class where it is declared. Failure to follow this guideline will lead to undefined behavior by the persistence system.

Getting and Setting Persistent Object Properties

Each persistent collection and persistent field must also have an associated accessor and mutator. The naming convention for the accessor and mutator must follow the JavaBean convention(reference here) For example, consider the following PField:

```
@PField(
    type=SaveableType.STRING,
    variableName="name")
```

The object must also define "String getName()" and "void setName(String)" methods.

A PCollection looks similar:

```
@PCollection(
    type=CollectionType.LIST,
    variableName="imagesets",
    cascade=Cascade.ALL,
    identifiesDependentType=Univelscan.class,
    valueOrEntity=ValueOrEntity.VALUE,
    collectionValue=@CollectionValue(
        fieldName="imagesets",
        type=SaveableType.STRING)
```

The associated collection would look like "List getImagesets()" and "void setImagesets(List)"

Assigning Unique Identifiers

A unique identifier must be assigned to each data object, so that it can be distinguished from other objects of the same type. Two types of identifiers exist in our implementation. The first is a *business logic identifier.* A business logic identifier has some meaning in the context of the application. This could be, for example, a book title in the context of a bookstore inventory application. A book title may actually belong to several books, thus making it unsuitable for use as a unique identifier. Even

if another field is combined with book title, such as author, it is very hard to guarantee that the two fields together will uniquely identify the a particular book (eg, what if the book has multiple printings?). For this reason, using only business logic identifiers is not preferable for large updatable data sources.

In 1970 the ISO adopted standard 2108, introducing the International Standard Book Numbering (ISBN) system, whereby every book is issued an unique number. We use a similar concept to uniquely identify objects in data sources. Each object is assigned a *unique numerical value*. When used in conjunction with the database data source, each new identifier is generated natively by the database management system itself. Many databases simply use a counting system in which objects are assigned identifiers based on the order they are inserted into the database. Unique identifiers allow databases to more quickly assemble an object for retrieval, while still allowing the application to search for the object using business logic identifiers.

As was mentioned previously, the ClassInfo annotation has a property called keyField which defines the variable that will be used as the discriminating value between objects. This field defines the unique numerical identifier which should be a long integer and be initialized by the constructor to be null. When saving, the data source will examine the value of that field to determine what action to take. If the value is null, that indicates that this is a new object and the persister must ensure that a unique number is generated for the object. If an unique identifier is

already present, the object will overwrite the object with that identifier already in the database.

The XML implementation, because the XML JAR is being treated as a non-updatable data source, does not assign an identifier, but rather depends on each object to distinguish itself from other objects using business logic identifiers only. Because the XML JAR is a non-updatable data source (see *Data Sources* section), the entire JAR must be read into the database to be manipulated. When the objects are written to the database, unique identifiers will be determined and assigned to each object.

Architecture

The persistence system is a 2 tier architecture (Figure 2). The topmost layer pictured, the UI layer, is not part of the persistence system. It is just shown to give the reader a rough idea of how the persistence system integrates with the overall application.

The Data Layer

The top layer of the persistence system is the data layer. This layer is shared between the UI layer and the persistence system. The UI stores data in the data layer that it needs for future use. In the context of Minerva this includes a patient's name, address, image sets, and so forth. Each data class contains metadata as outlined in

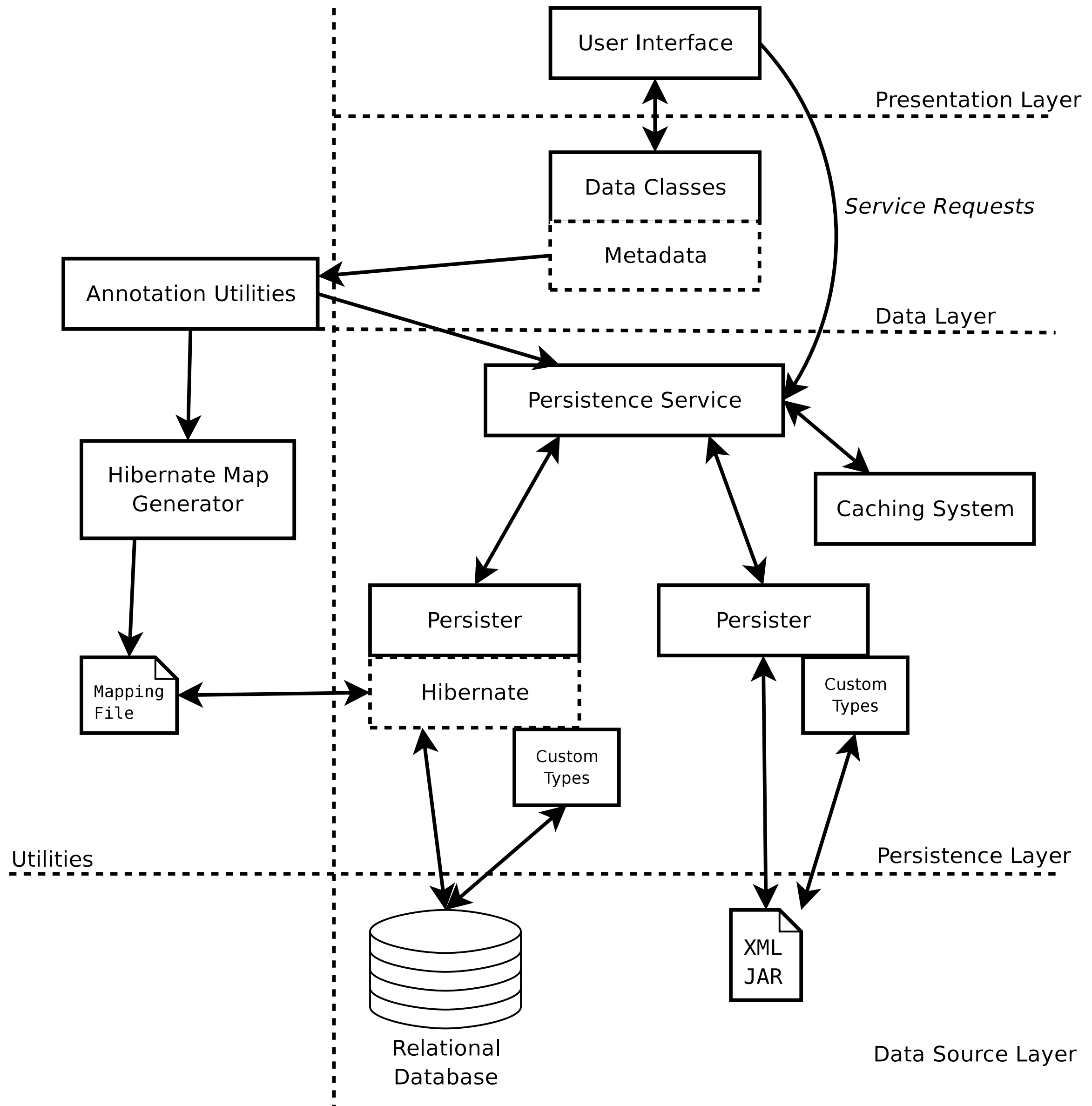


Figure 2. Persistence System Architecture.

the Describing the Data section. The metadata is parsed by the AnnotationUtilities class and delivered to the second layer of the persistence system.

The Persistence Layer

The second layer of the persistence system is the bulk of our implementation. The diagram greatly simplifies the structure of this layer for easier presentation. The "Persistence Service" box corresponds to many classes, including MultiDatastore and DataAccessPath which are the public interfaces from the persistence service to the rest of the application. These two classes will be discussed further in the *Data Operations* section.

AnnotationUtilities

The AnnotationUtilities class consists of several static methods to parse information from a class's annotations. It is used by the persisters to construct the object graph, determine dependencies, and read the information stored in each PField and PCollection. It is also the only class that interfaces directly with the metadata.

The Persisters

Below the "Persistence Service" are the various data source persisters. We have implemented two, one for relational databases and one for XML, but others can easily be added. Data source persisters implement a small set interfaces for integration with the system. See the *Adding a New Persister* section for more details. The persisters do the work of actually taking an object and serializing it into a data source.

For the XML persister, this is done directly by using the data object and its metadata along with any custom types in use to form an in-memory JDOM XML document. Additional objects may be added to the document until the XML JAR is closed. When an XML JAR is read in, the metadata and the custom types are used to read the object from the XML JAR so it can be stored in the database.

The database persister is more complex. As mentioned previously, the database persister makes use of the Hibernate ORM tool as a subsystem to generate the JDBC calls needed to interact with the database. To accomplish this, Hibernate requires external configuration files to understand how to map the objects to the database. The metadata in the data classes already contains this information, so the HibernateMapGenerator uses the AnnotationUtilities and JDOM to construct these configuration files.

Caching

For efficiency reasons a caching system was implemented by Michael L. Milvich of the Minerva team. This subsystem is available to the persistence system for use with all persisters. When an object is loaded or saved to a data source, it is also cached. If an object in the cache is requested, the cached object is returned rather than reloading it from the data source. Hibernate also offers a wide array of caching systems, but they have not been enabled. See the *More Advanced Caching* section in the *Open Questions* chapter for additional discussion.

Data Operations

MultiDatastore

As previously mentioned, the persistence system has been designed to allow for concurrency. In order to request an action from the system, a DataAccessPath (DAP) must be obtained. This can be done from the singleton class MultiDatastore (MDS). First, the singleton instance of MultiDatastore must be obtained and asked to create a DataAccessPath. The paramaters that must be given to createDataAccessPath are the Datasource to create the access path to, and the data source's identifier.

Example: Creating a DataAccessPath for database requests

```
DataAccessPath dap =
  MultiDatastore.getInstance().
  createDataAccessPath(Datasource.DATABASE, "databaseName");
```

Concurrency Pitfalls. Several DataAccessPaths may be in operation concurrently across many classes. However, if more than one DAP is writing to the same data source concurrently, safety is not guaranteed. Mutual exclusion to data is to be handled by higher layers of the application, or the data source itself.

DataAccessPath

Now that a DAP instance is available, operations may be performed. The three main operations: persist, load, and remove all work in a similar fashion. Each will be explained in more detail in turn.

Persist Operation. The persist operation should be used to insert a new object into the database, or to update an existing object. The difference between a new object and an updated one is subtle, and must be understood to correctly use the persist operation. See the *Assigning Unique Identifiers* section for more explanation.

The persist operation is invoked using the *void persistObject(Persistable obj)* method. The argument can be any class that implements the Persistable interface and has been correctly annotated. If an error occurs, a DataWriteException will be thrown.

Example: Persisting two objects

```
dap.persistObject(persistable1);
dap.persistObject(persistable2);
```

Load Operation. The load operation is used to load a currently stored object from the data source. Two types of load exist. The first, a full load, is invoked by calling *Persistable load(Class c, String dataID, String patientID)*. This method will load a single object from the data source with type c, and a combonation of identifiers, dataID and patientID. This triplet will uniquely identify a single object stored in the data source.

Example: Loading an Object

```
Persistable p = dap.load(Geometry.class, "dataID", "patientID");
```

The second type of load operation, a shallow load, is invoked by calling *Persistable loadLazy(Class c, String dataID, String patientID)*. This method will load the object

except for any collections marked with the isLazy property in the PCollection annotation. This is advantageous if the collection is very large, and not needed at the time of load. Note that accessing any lazy collections after a shallow load is undefined. If the collection is needed, the object must be reloaded from the data source using the standard load operation.

Example: Shallow Loading an Object

```
Persistable p = dap.loadLazy(Geometry.classY, "dataID", "patientID");
```

Remove Operation. The remove operation is used to remove a currently stored object from a data source. To maintain data integrity in the object graph, the object marked for removal must not be depended upon by any other object. If dependencies do exist, the parent objects must break all ties to the marked object before the remove operation will be permitted. Once the object has been isolated, the remove operation may be invoked by calling *Persistable removePersistentObject(Class c, String id).* If the object being removed also resides in memory, the object's makeTransient() method should also be called to remove all unique identifiers that may have been previously assigned. This operation has only been implemented for the database data source.

Example: Removing an Object

```
dap.removePersistentObject(Class.c, "dataID");
```

Adding a New Data Source Persister

There are two main components to adding a new data source to the system. The first is to implement the persister for the data source which is accomplished by using the Persister interface. Not all methods must be implemented, only those the persister needs to provide. For example, if the data source is not transaction-safe the beginTransaction() method should throw an UnsupportedOperationException.

The second issue to consider when adding a new data source is any custom types that may be needed. An interface should be created which, at a minimum, provides methods to write-out and read-in the custom type. Then the interface should be implemented for each custom type needed. If a custom type for a specific type of data already exists for use with other data sources, the same class can be used to implement the new interface.

Other work that must be done includes implementing the PersisterBuilder interface, which can construct and cleanup the new persister and making the persister available to DataAccessPath.

RESULTS

Integration with MINERVA

The persistence system implementation discussed provides paths to relational databases and XML. These two data sources have been integrated with parts of the MINERVA application and are performing as required. Additional work is being done by the MINERVA team to convert the entire application to the new implementation because of the advantages it offers over a hand coded and maintained persistence system (see the Motivation and Requirements sections).

Performance

The performance of the new persistence system is comparable to a similar system written with straight JDBC. The MINERVA application was used to test load and save speeds under both a custom JDBC implementation as well as the Hibernate brokered implementation discussed here. Both tests showed the custom JDBC implementation to be slightly faster in most cases. This is partly due the comparatively large number of tables being joined together by Hibernate. Other factors include the large amount of reflection being used by Hibernate itself as well as the decision to save the binary data in small chunks that must be dissassembled and assembled on save and load respectively.

In some instances, however, the Hibernate implementation out-performed stock JDBC. This occurred when Hibernate was able to optimize its update queries due to parts of a data object remaining unchanged.

Shallow Loading

The implementation of shallow loading for the database persister is functional, but does not take full advantage of Hibernate's lazy loading system. Lazy loading allows an object or collection to be loaded in stages. The first stage loads all parts of the object not marked as lazy. The second, delayed stage, can be activated by accessing the lazy component. This allows potentially large chunks of data to not be loaded unless they are needed by the application.

This could increase the efficiency of database access, but in practice it turns out to be very difficult to manage. Hibernate uses the idea of a session, a single request/response cycle (Hibernate in Action pp. 172) from the database. Lazy portions of an object must be initialized with the session that originally loaded the object. The reader may be puzzled as to how a short-lived session can be kept alive for an extended period of time needed to be used in conjunction with lazy loading, and that is a problem. There are some tricks described by the Hibernate documentation to accomplish this, but they are often riddled with holes and special cases and therefore not entirely usable.

In addition, when data is loaded within MINERVA, it is usually known if the lazy components are needed at load time. This also allows the application to have predictable response times because all the data needed is loaded up front. The Hibernate lazy loading is used in the limited fashion described earlier to provide shallow loading.

Using Hibernate with Relational Databases

While not all the features of Hibernate were used, its SQL generation capabilities and object based query language proved to be very helpful as part of the implementation. At first, the lack of documentation was frustrating, but the Hibernate community was very helpful in answering questions through the forums. Christian Bauer and Gavin King, the Hibernate architects, have now published a book titled *Hibernate in Action* which helps clarify the complex issues involved when using Hibernate.

Although it is functional when wrapped in a complex data system which abstracts it from the rest of the application, Hibernate feels much more natural when embedded directly into the data objects. Of course, this is often not possible, and Hibernate development seems to be moving away from this approach. At this time, Hibernate 3 has just released (Hibernate 2.1 was used in discussed implementation) and looks to ease integration with larger systems.

XML JAR

JDOM made building the XML document very straight-forward. Due to its generic style and heavy use of the annotations and reflection, the XML persister contains a relatively small amount of code. The custom XML types worked very well for handling the binary data, and are generic enough to handle future data that may need special handling.

FUTURE RESEARCH

Future Data Sources

As mentioned before, we have implemented persistence services to both relational databases and XML files. There are, however, many more possible data sources, such as object databases and XML databases. While the metadata fields provided have been designed to be general and to accommodate as many data sources as possible without modification, some additional fields may need to be added to the various annotations for future data sources.

More Advanced Caching

The in-house developed caching system currently in use is sufficient for the data of the MINERVA application at this time. There are, however, more advanced caching systems available such as EHCache, OSCache, and TreeCache. These caching systems are highly optimized and very robust; they are also supported natively by Hibernate and should be usable with the current persistence service with minimal effort.

Object Remapping and Versioning

Currently, when an annotation is modified, the database and XML persisters will immeaditely begin to read and write data in the new format. This however, does not allow them to read data of old formats. Versioning the data could be a solution to

solving this problem. Each annotated field could be given an "added" and "removed" versions; classes would also be assigned an overall version. This would allow data to be moved from version to version more easily. For example if class A has two fields added to it, A's version would change and when old data was read the new fields could be assigned sensible default values by looking them up in the annotation system. A similar system could be used to write data back to an old version if needed.

More Efficient and Extendible Shallow Loading

As explained in the results chapter, a shallow loading system which does not take full advantage of Hibernate's lazy loading has been implemented for the the MINERVA application. Although sufficient for MINERVA, something more advanced may be preferable for other applications. With Hibernate 3 and some advanced design patterns, a system that can take full advantage of lazy loading should be attainable.

Graphical Frontend for Manipulating Datagraphs

In order to show the relationships between various data objects a graphical user interface could be developed to allow the user to see the effects of adding, updating, or removing objects. For example, if object A depends on object B and the user requests removal of object B, a dialog could be presented to the user showing how the entire data graph will be effected by the requested action (in this case object A would also need to be removed).

Updatable XML Data Source

The XML data source was defined to be immutable after the JAR was closed for writing. This need not necessarily be the case. An XML JAR could technically be made updatable quite easily. This may offer some advantages, but using a single XML JAR for long term storage of large numbers of objects will be both inefficient and dangerous due to the lack of transaction support. One write error could render the whole JAR unreadable. If a long term XML data source is desired, an XML database is likely a better choice.

Using Annotations in JavaDoc

The annotation system specifies a lot of useful information, such as how classes depend on one another. This information could be integrated into a class's JavaDoc for easier reference. Other types of annotations may also be beneficial to add to the JavaDoc as well.

Handling Non-annotated Classes

In certain instances, a class or collection may contain objects which cannot be annotated, perhaps because they are part of a library for which the source is not available. The current methods of handling this situation are to: (a) build a wrapper class for the object and annotate that class or (b) use a custom persister to read and

write the object. A more advanced system, perhaps using a reflection tool, would be preferable.

REFERENCES CITED

[1] Mark L. Fussel. Foundations of object-relational mapping. Available at: http://www.chimu.com/publications/objectRelational/, July 1995.

[2] Alan C. Kay. The early history of Smalltalk. *ACM SIGPLAN Notices*, 28(3), 1993.

[3] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[4] Franois Banciihon. Object-oriented database systems. In *Proceedings of the seventh ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 152–162, 1988.

[5] Neal Leavitt. Whatever happened to object-oriented databases? *IEEE Computer Society*, 33(8):16–18, 2000.

[6] Lucas Jellema. Java 5.0 (tiger) - metadata and annotations - introduction and thoughts on application. Available at: http://technology.amis.nl/blog/index.php?p=188, March 2004.

[7] Sun Microsystems. Jdbc technology. Available at: http://java.sun.com/products/jdbc/, April 2005.

[8] King Bauer. *Hibernate In Action*. 2005.

[9] Jason Hunter. Jdom. Available at: http://jdom.org, April 2005.

[10] David Megginson. Sax. Available at: http://saxproject.org, April 2005.

[11] Philippe Le Hgaret. Document object model (dom) specifications. Available at: http://www.w3.org/DOM/, April 2005.

[12] Jason Hunter. Xerces2 java parser readme. Available at: http://xml.apache.org/xerces2-j/, April 2005.

[13] Crimson. Available at: http://xml.apache.org/crimson/, April 2005.

[14] Apache ant.
Available at: http://ant.apache.org/, April 2005.

[15] Junit, testing resources for extreme programming.
Available at: http://junit.org/index.htm, April 2005.

[16] J.R. Venhuizen ed. INEEL Advanced Radiotherapy Reasearch Program Annual Report 2003. 2003.

APPENDIX A

DETAILED ANNOTATION PROPERTIES

ClassInfo Annotation Properties

Child[] children()

This property defines all the classes the current object depends on via reference links. It also describes the relationship (one-one, one-many, many-many) between the two objects as well as how operations should be cascaded from the parent object to the child. The relationship and cascade properties are not currently in use by the database or XML persisters, but are included for future development of these persisters, or the addition of new persisters.

Parent parents()

The parent property is the reverse of the child property, except that it only holds the type of the parent object.

String keyField()

This property should define the name of the class field which contains a unique identifier of type Long. The identifier has no business meaning, and is present only to give persisters a faster, more efficient way to uniquely identify objects. Only the database persister currently utilizes this property. See the *Assigning Unique Identifiers* section for more details.

String identifyingProperty()

This property is the name of the class's business logic field that identifies the object.

Class extendsClass()

This property describes the superclass (if not Object.class). If a superclass does exist, its PersistentFields and PersistentCollections will be processed before the subclasses' when persisting an object.

PersistentFields Annotation Properties

SaveableType type()

This property is an enumeration that defines of which SaveableType the object is. The enumeration covers all the Java primitive types as well as String and Date. If the field does not fit into any of these categories, SavableType.CLASS and SaveableType.CUSTOM have also been included.

SaveableType.CLASS should be used when the field being annotated is itself another annotated class. The persistence system will go look at its annotation for more infomation.

SaveableType.CUSTOM should be used in two situations. The first is an instance where the object being referenced isn't annotated, perhaps because the developer doesn't have access to the source for the class. (If the class is accessible, it is usually preferable to annotate the object.) The second reason SaveableType.CUSTOM might be used is in a situation where the object needs to be handled in a very specific way, and the data source persister default handler is not sufficient.

In either of these cases, an additional interface must be implemented. Each data source will provide interfaces to read and write custom objects as part of their implementation. For the database and XML persisters, these are net.sf.hibernate.UserType and minerva.tirade.common.persistence.usertypes.XMLUserType respectively. Full details about these classes will not be given here, but implementation examples and documentation can be found for byte[] and short[] in the minerva.tirade.common.persistence.usertypes package.

Class customType() [optional]

This property defines the custom-type class to use for persisting and loading the annotated field. The class given should implement the user-type interfaces for all persisters it will be used with. This value should only be defined if type() is set to SaveableType.CUSTOM! If it is defined at any other time, the results are undefined. See type() for more information.

Class classType() [optional]

This property gives the name of the annotated field's class. This value should only be defined if type() is set to SaveableType.CLASS! If it is defined at any other time, the results are undefined. See type() for more information.

String variableName()

This property defines the variable name (field name) to associate with this PField. This must be the exact variable name used in the class definition.

String saveName() [optional]

This property defines the name to call the field in the data source. This only needs to be defined if different than the variable name.

String defaultValue() [optional]

This property defines a default value to use when persisting if the field is null.

String sqlType() [optional]

This property defines a specific SQL type to use in the event the default provided is not sufficient. This property only effects the database data source.

Class identifiesDependentType() [optional]

This property defines what dependant type this field identifies. Note that this is done for referenced associations only and not direct associations. See *ClassInfo* section of the *Implementation* chapter for additional explanation.

boolean isPrimaryIdentifier() [optional]

This property should be defined when the field is business logic primary identifier for the object. Please see Assigning Unique Identifiers section for more information.

PersistentCollections Annotation Properties

CollectionType type()

This is the type of collection. Range of values are LIST, MAP, SET, ARRAY.

String variableName()

This property defines the variable name (field name) to associate with this PCollection. This must be the exact variable name used in the class definition.

String saveName() [optional]

See corresponding PField section.

String displayName() [optional]

See corresponding PField section.

boolean isLazy() [optional]

This property should be defined if the collection has the ability to be shallow loaded upon request.

Cascade cascade() [optional]

This property defines how operations performed on the encapsulating class should handled by the collection. The possible values of cascade are:

ALL - All actions are cascaded

NONE - No actions are cascaded

SAVE - Save operations only are cascaded

DELETE - Delete operations only are cascaded

Class identifiesDependentType() [optional]

This property should be defined if the collection contains identifiers for referenced child objects. See *ClassInfo* section of the *Implementation* chapter for additional explanation.

SaveableType indexType()

This property should be defined for all MAP collections. For LIST and ARRAY SavableType.INT is used by default, and SET ignores this property. For a

MAP, it defines the type of key used in the map. It can be any SaveableType except SavableType.CLASS and SaveableType.CUSTOM.

String sqlType() [optional]

This property may be defined when a specific SQL type must be used to store the elements of a value collection. This property will have no effect on an entity collection.

ValueOrEntity valueOrEntity()

This property defines whether the collection is a value collection or entity collection. Value collections have elements of type SaveableType (except SaveableType.CLASS, SaveableType.CUSTOM). Entity collections have elements which are other annotated classes.

Example: Value Collection

List of ints, Map of Strings, Set of Dates, Array of floats

Example: Entity Collection

List of DataWrappers, Set of Constraints

There is a special situation in which an entity collection may contain objects that are not annotated types. This often happens when storing predefined objects from the Java library in a collection. In this case, a wrapper object must be constructed and annotated, and the wrapper must be used in place or the actual class in the annotation definition. The wrapper must contain annotations, getters and setters for each field of the original object needed for object reconstruction.

<u>CollectionValue collectionValue()</u>

This property will describe the name to save the value under, and the type of the value. This property should only be defined when valueOrEntity() is set to ValueOrEntity.VALUE!

<u>CollectionEntity collectionEntity()</u>

This property will describe the name to save the entity under, and the class to get the entity's annotation from. This property should only be defined when valueOrEntity() is set to ValueOrEntity.ENTITY!