Generic Object-Oriented Database System

Konstantin Knizhnik

March 3, 1999

Contents

List of Tables

1 Introduction

GOODS is general purpose distributed multilingual object-oriented database management system. It supports client-server model with active clients and passive servers. Database consists of a set of storages, each storage is managed by separate server. Distribution of objects between storages is mostly transparent for client, but it can attach object to concrete storage. All object methods are executed at clients. Objects are stored in storage in language and operating system independent format, so it is possible for applications written in different programming languages to work simultaneously with the same database. GOODS provides distributed garbage collector to guarantee references integrity. All features of classical

DBMSes, like ACID transactions, concurrency control, online backups, crash recovery and server monitoring are provided by GOODS. GOODS also supports online scheme modification without stopping client applications. Moreover, it is possible for several clients, having different class definitions, to work simultaneously with the same database. Conversion of object instances will be done automatically in lazy mode, when object is loaded and modified by some client.

Currently interfaces for C++ and Java language are provided. Interface with C++ is based on using of smart pointers and is discussed in readme.htm document. Interface for Java language is based on using special Java byte-code preprocessor and is described in the following sections.

2 Metaobject Protocol for Java

2.1 Transparent persistency and concurrency control

There are three main requirements to GOODS language interface: transparency, flexibility and efficiency. Depending on language, different approaches should be used to reach this goal. But the main idea is the same: the main advantage of object oriented databases is elimination of gap between application and database data models, so that programmer can use the single paradigm and approach to design application and database objects. To achieve this goal we need transparent persistency: should not be any differences between code working with transient and persistent objects. But database applications have some aspects, which are not present in other applications. They have to control concurrency, security, data consistency. So it will not be possible to completely exclude this database specific code from the application. The idea is to separate application code itself and code responsible for controlling concurrency and consistency. These two separate aspects can be combined together at compile time by using Metaobject Protocol. Such approach makes development of application code much easier as well as development and debugging of synchronization code. Also using of MOP provides high flexibility, which is not possible in the systems supporting only few predefined concurrency control disciplines.

One of desired features of OODBMS programming language interface is support of orthogonal persistency. It means that the property of object to be persistent or transient is orthogonal (independent) from it's class and constructor used for object creation. This purpose used to be achieved through persistency by reachability. Object becomes persistent when it is reachable (exist references to the object) from persistent roots or from other persistent objects. But complete support of orthogonal persistency cause significant runtime overhead. That is why GOODS language interface provides semi-orthogonal approach. Only object of some selected classes (they should be derived from special base class) can become persistent. We will call them persistent capable classes. Principle of persistency by reachability is still used here: not all objects of persistent capable class becomes persistent, but only ones reachable from persistent roots. Such approach allows not to pay performance penalty for normal (transient) objects, and so provides efficiency of OODBMS language interface implementation.

2.2 JavaMOP utility

As far as Java language supports only structural reflection and doesn't provide any facilities for behavioral reflection, we need some preprocessing tool to implement metaobject protocol for Java. Possible alternative is development of special Java Virtual Machine, but this is not so simple task and also this approach will restrict sphere of application using. I prefer to develop interface compatible with JVM implementations of different vendors.

The decision was made to use byte-code preprocessor. Development of such preprocessor is much simpler than development of source-level preprocessor (no language parser should be constructed). Also absence in Java language of such construction as C `#line` directive, makes compiling and debugging of application with source-level preprocessor very inconvenient. That is why byte-code preprocessor JavaMOP was developed. It is implemented in C++, to increase speed of processing and makes development more comfortable.

JavaMOP receives as a parameter list of Java classes, directories or archives of Java classes (.JAR or .ZIP without compression). If archive file was specified, JavaMOP is able only to read definition of classes in this archive, but not to preprocess this classes. So to preprocess some class library, you should first extract files from the archive and pass root directory of the extracted tree to JavaMOP. If directory name is passed to JavaMOP, it will handle all .class files in the directory and recurse into all subdirectories.

JavaMOP looks for classes containing "metaobject" field as instance variable. All such classes and classes derived from them are considered to be controlled by metaobject. JavaMOP wrap bodies of methods of all such classes with calls of metaobject methods: preDaemon() and postDaemon(). As it is clear from their names, first is invoked before execution of first method statement, and last one after return (normal or as a result of exception) from the method. JavaMOP also detects accesses to non-self instance variables of such classes. JavaMOP wraps such accesses in all methods (not only in instance methods of object controlled by metaobject) with invocation of preDaemon() and postDaemon() methods.

To implement transparent access to database, it is necessary to detect moment when object is modified and distinguish methods, which access object in read-only mode, from method, which can modify the object. Instance methods, which can (but not necessary) modify self object, are called in GOODS mutators. Knowledge of whether method is mutator or not is very significant to select proper object locking mode. The naive approach is to lock object in shared mode when it is first accessed and to upgrade lock to exclusive mode, when it is going to be modified. But such approach is deadlock prone: consider two applications accessing the same object. They both set shared lock and then try to modify the object. But none of them can grant exclusive lock since other application has shared lock, preventing from locking object in exclusive mode.

JavaMOP uses two-pass algorithm to correctly calculate set of all mutator methods. All methods modifying their object instance variables are considered to be mutators. If some method f() invokes method g() for the same object, and method g() was marked as mutator, then method f() should be also marked as mutator. Situation becomes more complicated because of polymorphic calls (virtual methods). We should mark method f() as mutator if any of implementations of method g() in self or derived classes was marked as mutator.

At first pass JavaMOP generates methods call graph and marks as mutators methods with instructions assigning some values to self instance components. Then JavaMOP propagates mutator attribute, iteratively scanning methods call graph and marking as mutators each method, which invokes some mutator method. After that, code is generated for all methods with known status of each method.

To detect moment of object modification, JavaMOP allocates extra local variable and assign to it non-zero value before each store instruction to self instance field. When postDaemon() metaobject method is called, value of this variable is passed to it to determine whether object was modified by the method. JavaMOP reduces size of generated code by doing basic block optimization: if there are several store instructions within one basic block, then modification marker variable can be assigned only once.

Constructors should be handled in special way since it is not possible to wrap constructor body with invocations of metaobject methods (because first statement of constructor should be invocation of constructor of superclass). So JavaMOP inserts call of postDaemon() method only after the top-level construction invocation statement, following object creation by new operator. Method preDaemon() can be called by application explicitly from constructor of some base class.

Although JavaMOP will do it best to set mutator and modified attributes, it is possible to application to explicitly mark some method as mutator or object as modified. This can be done by invocation of two special static methods: mutator() and modify(). JavaMOP removes code invoking these methods and, instead of this, changes correspondent attributes. So metaobject protocol, provided by JavaMOP, is very simple and consists only of two primary methods:

```
public abstract class Metaobject {
    //
    // Invocation of this method is inserted by MOP generator before each
    // method invocation or object component access.
    //
    abstract public void preDaemon(Object obj, int attr);
    //
    // Invocation of this method is inserted by MOP generator after each
    // method invocation or object component access.
```

```
    //
    abstract public void postDaemon(Object obj, int attr, boolean modified);


    //
    // Attributes for daemons
    //
    final static int MUTATOR    = 1; // object can be changed
    final static int VARIABLE   = 2; // access to non-self instance variable
    final static int CONSTRUCTOR= 4; // wrapped method is constructor
    final static int EXCEPTION  = 8; // method is terminated by exception

    //
    // This method is only hint to MOP preprocessor to consider method
    // invoking Metaobject.mutator() as been mutator. No actual code
    // will appear in preprocessed class file.
    //
    public static void mutator() {}

    //
    // This method provide MOP preprocessor with information that objects
    // was (or will be ) modified. No method invocation will appear in
    // preprocessed code.
    //
    public static void modify() {}
};
```

JavaMOP provides some other useful facilities. For each class, controlled by metaobject, it generates special constructor with single Metaobject parameter, which is passed to the constructor of base class. Such constructor is used in GOODS to create object instance when it is loaded from database. But because of using this generated constructor it is impossible to declare instance final components in classes controlled by metaobjects (JavaMOP is not able to insert correct initialization code for such components in the generated constructor, causing verification error). Attribute final can not be used even with instance fields declared as transient.

The second JavaMOP feature can be used to overcome limitation of reflection facilities of JDK 1.1, which violates access to non-public components of non-public objects (in JDK 1.2 it is possible to set bypass flag and access all components of all objects). As far as there is only beta release of JDK 1.2 at this moment (and even after JDK 1.2 will be available, a lot of users will continue to use JDK 1.1) JavaMOP has special option -public, which tells JavaMOP to set public attribute to all MOP classes and their fields. Using of this option is not needed with JDK 1.2.

Another JavaMOP option "-classes" allows you to specify list of classes, which should be controlled by metaobjects. In this case exactly one class with "metaobject" instance field should be provided to JavaMOP. We will call this class MOP root class. JavaMOP will transform definition of the classes, specified in the list, to make them derived from MOP root class. The option "-classes" should be followed by file name. This file should contains list of full class names, separated by space characters (new line, space, tabulation...) Class name should be written in the same format as in Java import statement. For example: `goodslib.ArrayOfChar` or `goodslib.*`. If '*' symbol is placed at the end of class name, then all classes from specified package and nested packages are implicitly derived from MOP root class. Classes specified in the "-classes" option list should not be archived, otherwise JavaMOP will be not able to preprocess them.

When JavaMOP is used for GOODS client interface, there is the single class `goodsjpi.Persistent`, containing metaobject instance field. All persistent capable classes should be derived from Persistent class (it can be done implicitly by JavaMOP, see paragraph below). So the set of persistent capable classes is equal with the set of MOP classes (classes controlled by metaobjects).

It is possible to use existed Java class library without changing sources by putting list of persistent capable classes in some file and using JavaMOP with "-classes" option. But be care with using this option. The following rules should be observed:

1. Class should not have more than one instance array component.
2. If instance array component is modified (by storing array element), method `Metaobject.modify()` should be called explicitly to mark object as modified.
3. All classes, which objects can be referenced from some persistent capable class, should be also persistent capable (so persistent capable classes should form closed set in sence of inteclass references).

It is significant to notice that although JavaMOP was designed for GOODS Java interface, it has no GOODS specific and can be used in any application requiring non-standard manipulation with object instances.

2.3 Basic metaobjects

GOODS interface for Java language provides set of predefined metaobjects, implementing most popular disciplines of concurrency control. By deriving from these classes, programmer can develop own metaobjects, implementing sophisticated synchronization disciplines and fitting requirements of concrete application.

Most of metaobject functionality is implemented in BasicMetaobject class. This class is responsible for managing object cache, controlling transaction processing and handling object invalidation messages from the servers. GOODS provides implicit model of fixing transactions: each invocation of method of object controlled by BasicMetaobject is considered as nested subtransaction. Transaction is automatically committed when all nested subtrasnactions are terminated (in another words, when there are no more active methods of classes controlled by metaobject). BasicMetaobject metaobject invokes `beginReadAccess()`, `beginWriteAccess()` and `endAccess()` methods, which implement concrete synchronization policy by setting shared or exclusive locks. Three policies are supported now and so three metaobjects, derived from BasicMetaobejct, are defined:

OptimisticMetaobject Accessed objects are not locked at all. When transaction is committed, the server checks if client has the most recent version of the object. If some other client modified object before this client, then transaction is aborted by the server and it depends on application how to handle this situation. Objects, which are only read and not modified by application during transaction, are not checked by the server to be up-to-date. This strategy is efficient, when conflicts are rare and it is possible to restart transaction. It also can be used when access to the object is always done through some other object (container), which is responsible for synchronization.

PessimisticMetaobject This metaobject sets exclusive lock on the object, when it is accessed by the mutator method, to prevent loosing of modifications as a result of transaction abort. Locks are released at the end of transaction. This metaobject doesn't lock objects accessed in read-only mode and such objects are not checked by the server to be up-to-date.

PessimisticRepeatableReadMetaobject This metaobject is derived from PessimisticMetaobject and in addition to setting exclusive locks for modified objects, it also locks read-only objects in shared mode to prevent object from modification by the other clients until the end of transaction. This metaobject is used by default for all persistent capable classes. It provides the highest level of consistency, but at the expense of extra messages passing and reducing concurrency. Also it can cause deadlocks by upgrading shared locks to exclusive.

Basic metaobject controls work of object cache. As far as loading of objects is the most frequent operation in OODBMSes, performance of database application mostly depends on effective work of cache mechanism. That is why special attention was paid to implementation of cache object replacement discipline in GOODS client library. The most popular discipline for cache management is LRU (replacing of lest recently used object). But it is not smart enough for database applications. For example, sequential search through large number of objects can completely flush the cache by replacing all objects in it and most of loaded objects will not be used by application in future.

To solve this problem, extension of standard LRU scheme was proposed in GOODS. Object cache is divided into two parts: primary cache and cache for frequently used objects. First time the object is accessed, it is placed in the primary cache. The object will be moved to the "frequently used objects" cache only when it is accessed several times. Both parts of the cache are managed by standard LRU discipline separately: when total size of objects in the one part exceeds some limit value, least recently used objects from this part are thrown away from clients memory, not affecting objects in another part of the cache. It is possible to set limit values for both parts of the cache by BasicMetaobject.setCacheLimits(int l0, int l1) method. Parameter l0 specify limit for primary cache, and l1 - for cache of frequently used objects.

3 How to develop Java application for GOODS

Development of Java application for GOODS is very simple. First of all, you should import package goodsjpi and derive all classes, objects of which you are going to store in GOODS database, from goodsjpi.Persistent class (you can use "-classes" option of JavaMOP to let JavaMOP do this work for you). We call such classes persistent capable classes. All object referenced from persistent capable classes, should also be persistent capable. The only exception is fields declared as transient or static. GOODS Java interface doesn't store values of transient fields in database. After loading persistent object from database all its transient fields are set to default value. Also values of static fields are not saved and will be lost after application termination. In addition to reference fields, it is also possible to use all Java primitive types for declaring components of persistent objects. You can't use final modifier for instance fields, because of presence of special constructor generated by JavaMOP.

Special convention regulates using of array types as persistent capable object components. As far as Java arrays are derived only from Object class, they are not persistent capable. Instead of this GOODS interprets persistent object with single array component as object with varying length (corresponding to C type with varying length of last component: `struct string { int length; char body[1]; };` So it is not possible to have more than one array component in persistent capable class and also you have to explicitly use `Metaobject.modify()` and/or `Metaobject.mutator` methods, because JavaMOP is not able to detect modification of array components. It is better to avoid using of Java arrays for components of persistent objects and instead of it use dynamic array classes from GOODS persistent class library for Java.

You should choose synchronization strategy to control concurrent access to the objects of your classes. By default in GOODS pessimistic repeatable read scheme is used. This scheme provides the highest level of isolation, but causes significant communication overhead and is deadlock prone. You can use pessimistic or even optimistic scheme or develop your own metaobject class. It is possible to assign metaobject to the object instance explicitly (but this binding will remain only until object is in client cache) or associate default metaobject with the object class (so that it will be used for all instances of the class). If the class definition has static final component of Metaobject type, then GOODS client library treats value of this field as default metaobject for this class and all derived classes (unless they specify their own default metaobjects).

Database servers are accessed by client application through Database class. Database is opened by `Database.open(String config_File)` method, which takes name of database configuration file as its parameters. This file can be accessed via NFS from central server or can be distributed to client computers through some other mechanism. First line of this file contains the number of storages in database. All successive lines specify location of each storage server. Each line consists of three parts separated by colon: storage identifier, host name and port number. Storage identifier should be in range `0 .. number-of-storages`.

After successful database opening, it is necessary to extract root object. Each GOODS storage has single root object. It can be retrieved by Database.getRoot(int sid) or `Database.getRoot()` method. Last one get root of storage with identifier 0. If storage was not yet initialized, getRoot() method returns null. In this case it is necessary to create root object and assign it to the storage by means of

`Database.setRoot()` method. To prevent concurrent initialization of storage by several client, root object is locked by `Database.getRoot()` method (only if null is returned) and lock is released only by `Database.setRoot()` method. Method `Database.setRoot()` can be invoked only once and should not be used for already initialized storage.

Having reference to root object, you can access any other object in the databases using standard facilities of Java language. Newly created object of persistent capable classes will become persistent when you store reference to it in some other persistent object. By default object will be placed in the same storage as persistent object, containing reference to it. But it is possible to explicitly assign the storage to the object by means of Persistent.attachToStorage(Database db, int sid) method. This method can not be applied to already persistent object in order to reallocate it in another storage. The object will become persistent after commit of transaction.

GOODS interface for Java uses model of implicit transaction commit: each invocation of method of persistent capable class is considered as nested subtransaction (without possibility to undo changes). The transaction is automatically committed when all nested subtransactions are terminated (in another words, when there are no more active methods of persistent capable classes). It is possible to explicitly start and finish nested transaction by means `BasicMetaobject.beginNestedTransaction()` and verb+BasicMetaobject.endNestedTransaction()+ methods.

GOODS can notify application about deterioration of object instance (if it was modified by some other client). Notification is done through object of goodsjpi.CondEvent class. Client wishing to receive notification about the object modification should invoke `Database.notifyOnModification(Persistent obj, CondEvent event)` method and create separate thread, which will wait until the event object will be signaled using `CondEvent.waitSignal` method. To cancel delivery of notifications, pass null value as the event parameter.

If optimistic synchronization strategy is used, it is possible to receive notification about aborting transaction. This can be done by `Database.notifyOnTransactionAbort(CondEvent event)` method.

After finishing work with database, you should call `Database.close()` method to finish database session. If client session was aborted by server (due to server shutdown or server/communication failure), then `Database.disconnected(int sid)` method is called, which throws `goodsjpi.SessionDisconnectedError` exception. Other errors, detected by GOODS client interface, are handled by `Database.handleError(String text)` or `Database.handleException(IOException x)` methods. By default these methods just raise exception signaling about server communication failure. Application programmer can redefine these methods by creating its own class derived from `Database` class.

So the template scheme of Java GOODS application is the following:

```
import goodsjpi.*;

class MyRoot extends Persistent {
    ...
}

class Application {
    static public void main(String args[]) {
        Database db = new Database();
if (db.open("app.cfg")) {
    MyRoot root = (MyRoot)db.getRoot();
    if (root == null) {
        root = new MyRoot();
db.setRoot(root);
    }
    ... // do something with database
    db.close();
} else {
    System.err.println("Failed to open database");
}
    }
```

```
}
```

After compiling your application, you should run JAVAMOP preprocessing utility. You should specify complete list of all Java class files, used in your application, together with `goodjpi.jar` (and `goodslib.jar` if used). You can see examples of running JAVAMOP tool in GOODS makefiles. JAVAMOP also has two optional parameters. The option "-package package-name" specifies name of the package containing definition of Metaobject. You should always use "-package goodsjpi" for GOODS applications. The second option is -public and it should be used only with JDK 1.1. With this option present, JAVAMOP will change access attributes of classes and fields to PUBLIC. Reflection package in JDK 1.1 doesn't allow to access non-public components and so makes loading of object from the database impossible. You can certainly declare all classes and their fields as public in Java sources yourself, but it is better to preserve class encapsulation at source level and use -public option to solve problem with JDK 1.1 reflection restrictions. The following command line illustrate usage of JAVAMOP:

```
javamop -public -package goodsjpi *.class \goods\goodsjpi.jar \goods\goodslib.jar
```

To run you application you should first prepare configuration file, for example "app.cfg":

```
1
0: localhost:6100
```

Then you should start the database server(s) at specified net nodes. Server is started by the command "goodsrv database-name [storage-identifier]". If storage identifier is not specified, value 0 is used by default. For example, GOODS server for this template application can be started by the command `goodsrv app`. More information about `goodsrv` program can be found in readme.htm document. After the server(s) is(are) started, you can run one or more client applications.

It is possible to change definition of classes without loosing instances of these classes already stored in database. GOODS support automatic online scheme evaluation. You can add new fields to the class, delete some fields, change primitive types of fields (any conversions between primitive types are allowed). Reformatting of the object instances according to the new format will be done in lazy mode. When client loads the object, descriptor of application class is compared with description of class in database. If descriptors are different, GOODS client library performs automatic conversion of the object to the new format. If object then is modified by application, it will be stored in the storage in the new format. It is possible for several clients to have different class definitions and work simultaneously with the same database. Unfortunately, GOODS automatic update mechanism is not able to handle renaming of fields, changing of types of references and changing of inheritance graph.

4 Persistent class library

GOODS client interface provides library of persistent capable classes implementing efficient algorithms for searching/retrieving object from database. Using of such container classes will simplify programmer development of database applications for GOODS. This classes are available from goodslib package.

4.1 Dynamic arrays

As far as Java builtin array types can't become persistent objects in GOODS Java interface model, special dynamic array classes, providing direct access to their elements by index, were developed. Size of such array can be changed at runtime and new elements can be inserted or removed from the array. Dynamic array classes for all Java primitive types are defined, as well as dynamic array of object references:

ArrayOfByte.java
ArrayOfChar.java
ArrayOfShort.java
ArrayOfInt.java
ArrayOfLong.java
ArrayOfFloat.java
ArrayOfDouble.java
ArrayOfObject.java
ArrayOfBoolean.java

Class `ArrayOfChar` is persistent analogue of Java String and provides basic methods for string manipulation. And implementation of `ArrayOfBoolean` provides space efficient way to store bitmaps. Definition of dynamic array classes in Java persistent class library is compatible with correspondent array definition in C++ class library (file `dbscls.h`). In C++ dynamic arrays are implemented as template class, and only the following instantiations of this template are defined in C++: `ArrayOfByte`, `ArrayOfInt`, `ArrayOfDouble` and `ArrayOfObject`.

All dynamic arrays can be constructed from corresponding Java array type (`ArrayOfChar` and `ArrayOfByte` can be also constructed from object of `String` type). Method `asArray()` construct Java array from the dynamic array class. Method `copy(int dstIndex, byte[] src, int srcIndex, int count)`, which is analogue of `System.arraycopy` method, is available for all dynamic arrays. Dynamic arrays also implement stack protocol by providing such methods as `Push(T value)`, `Pop()`, `Top()`. Methods `insert(int index, int count, T value)` and `remove(int index, int count)` can be used to add or remove elements from the dynamic array.

4.2 Ordered and unordered sets

Set container is implemented as L2 list. Owner of the set is represented by `SetOwner` class, it contains L2-list header and counter of elements in the set. Object-members of the set should be derived from SetMember class and implement `Ordered` protocol. `Ordered` protocol contains methods for comparison between set members and with key object.

Class SetOwner provides methods for insertion new members at the beginning of the set/at the end of the set, before/after specified set member and for removing member from the set. Also simple sequential search method is implemented in this class.

Class `OrderedSetOwner` extends `SetOwner` class and keeps members of its set in the increasing order (using comparison methods from Ordered interface). It redefines insertion methods of `SetOwner` class to preserve set members order.

4.3 B-Tree

B-tree is classical data structure for DBMS. It minimize number of disk read operations needed to locate object by key and preserve order of elements (range requests are possible). Also maintenance of B-tree can be done efficiently (insert/remove operations have $\log(N)$ complexity).

iN classical implementation of B-tree, each B-tree page contains set of pairs $< key,\ page-pointer >$. The items at the page are ordered by key, so binary search can be used to locate item with greater or equal key. In `B*-tree`, pointers to members are stored only in leaf pages of B-tree. All other pages contain pointers to child pages. But in Java language there is no structural types with value semantic, so object containing set of pairs $< key,\ page-pointer >$ can not be defined in this language (certainly it is possible to represent it by $M+1$ objects, where M is number of items at the page, but it is definitely not what we want). So decision was made to store reference to key in the B-tree page itself. So now the B-tree page contains only array of pointers and one extra pointer to the largest key at this page. Leaf pages contain pointers to the object inserted in B-tree, which should be inherited from `SetMember` class. All other pages contain pointers to the child pages. Such decision significantly simplifies algorithm, but increases number of objects, loaded by client from the server during search/update operations. `Btree` class is derived from `SetOwner` class and so allows sequential access to set members.

4.4 Blob

Most of modern database applications have to deal with large objects, used to store multimedia and text data. GOODS class library has special class Blob to provide efficient mechanism for storing/extracting large objects. As far as loading large object can consume significant time and memory, Blob object allows scattering of large objects into parts (segments), which can be accessed sequentially. Moreover, `Blob` object takes advantage of Java threads and makes it possible to load the next parts of the Blob object in parallel with handling (playing, visualization,...) of the current part of Blob. Such approach minimizes delays caused by loading object from the storage.

4.5 Closure

Class ObjectClosure provides a way to store transient objects in GOODS storage using Java serialization mechanism. Objects implementing `java.io.Serializable` interface can be packed into array of bytes using `java.io.ObjectInputStream`. This array can be stored with `ObjectClosure` object in GOODS database and lately retrieved and unpacked by `java.io.ObjectOutputStream`. So this mechanism can be used to store objects of classes, which can't be derived from `Persistent` base (not persistent capable). But be careful with this approach: semantic of object serialization is quite different with persistency by reachability: if the same object `A` is accessed from two different objects `B` and `C` and closures with root `B` and `C` are stored in database, then after restoring objects from this closures, objects B and C will refer to two different instances of the original C object!

4.6 Hash table

Hash table provides fast random access to the object by key. Implementation of hash table for GOODS is almost one-to-one copy of `java.util.hashtable` class. This class provides automatic resizing of hash table when number of elements in hash table becomes too large. Objects placed in hash table and their keys should belong to persistent capable classes. It is possible however to pass key of java.lang.String type, which is automatically converted to `ArrayOfChar` type. Methods `get(Object` key) and remove(`Object` key) accept key of any type.

Object of persistent capable class used as hash table key should either define its own `hashCode()` method or have already assigned persistent identifier (be persistent). Method `hashCode()` is redefined in `goodsjpi.Persistent` class to return the same value for all database sessions. But it is possible only if object has already assigned persistent identifier (it was made persistent in one of the previous transactions). Invoking `hashCode()` for transient object of persistent capable class, which doesn't redefine `hashCode()` method, will cause assertion failure.

4.7 H-Tree

Class `Htree` is combination of hash table and index tree. It can be used when size of hash table is too large to make possible represention of hash table as single object (as array of pointers). H-Tree first calculates normal hash key and then divide it into several groups of bits. First group of bits is used as index in the root page of H-Tree, second group of bits as index in the page referred from the root page, and so on... So if the size of the hash table is 1000003, then H -tree with pages, containing 128 pointers, requires access to three pages to locate any object. So total size of loaded objects is $128 * 6 * 3 = 2304$ bytes instead of 6Mb if `HashTable` class with such size is used (size of reference in GOODS is 6 bytes).

4.8 Class library

Class `ClassLibrary` can be used to store Java classes in GOODS database. This class is using Java class loaders mechanism to provide loading of class files not from file system, but from GOODS persistent objects.

Method `ClassLibrary.storeClass(String className)` stores class with the specified name in GOODS storage. Parameter className should be fully qualified Java class name separated by period character.

File name of the class file is produced by replacing period character with OS file separator symbol and appending suffix ".class". Then contents of the file is placed in the GOODS storage and reference to it is inserted in the hash table. This class can throw IOException if it fails to read the class file data.

Method `ClassLibrary.loadClass(String name)` searches in hash table for `ArrayOfByte` object with class file data. The name parameter should be the same as in correspondent storeClass method. Class is loaded and resolved using special persistent class loader and pointer to created `Class` object is returned. If no class definition was found in the hash table, than null is returned.

5 Building GOODS Java client library

GOODS Java client interface is implemented in pure Java and that is why it is portable and can be used with JDK 1.1 or JDK 1.2 and JVM of different vendors. Only JAVAMOP utility is implemented in C++ and is compiled and installed as other GOODS utility programs. GOODS Java client interface consists of two packages:

goodsjpi.jar Java programming interface for GOODS

goodslib.jar Java persistent class library for GOODS

To build these packages, you should specify target `java` to `make` utility. Make should be invoked from GOODS root directory. For example the following command will build all GOODS staff, including server library, utilities, client library for C++, client library for Java, C++ and Java application examples:

```
make all java
```

There is one unportable item in implementation of GOODS client interface for Java. To support cache of loaded persistent objects we need weak references functionality (weak reference provides a way to access object but doesn't prevent garbage collector from deallocating the object). Weak references are included in JDK 1.2, but they are absent in JDK 1.1. In JDK 1.1 exists undocumented class `sun.misc.Cache` which is actually hash table with possibility to GC to collect unreferenced elements. Unfortunately, this class doesn't work correctly with JVM's of all vendors. For example JDK shipped with Microsoft IE 4.0 contains this class, but it lost "weak" property: objects placed in the Cache can not be collected by GC. Instead of this Microsoft provides class `com.ms.vm.WeakReference` (which is not present in JDK for IE 3.0). So GOODS tries to do it best to guess, which implementation of weak references should be used. It contains special package goodsjpi.weak, which encapsulates system dependent details of weak reference implementation. But as far as JDK 1.1 has no standard support for weak references, not all JVMs using JDK 1.1 can be used for running GOODS Java client interface. At least the following environments are tested to be compatible with GOODS: Sun JDK 1.1.1 - 1.1.6, Sun JDK 1.2, Microsoft Jview from IE 4.0, Borland Jbuilder 2.0.

6 Compatibility with C++

As far as object are storage in GOODS storage in language independent format, application written in different languages can access the same database. GOODS object model was chosen to be flexible enough to support object models of many existed object oriented languages. Currently only C++ and Java interfaces are supported. Unfortunately object models in this languages contain a lot of differences, so there are some restrictions for object formats, if you want to make them accessible from C++ and Java applications.

First of all Java, language has no structured types with value semantic. In other words, Java object can't contain instances of other objects, it can only have references to them. GOODS object model allows structure components of objects, but if you want to access objects of this class from Java applications, you should avoid use of such structures.

Second restriction is caused by different representation of object with varying component length in Java and in C++. In C++ it is defined in the following way:

```
    class string {
      protected:
int  size;
char body[1]; // component with varying size
    };
```

So space for varying component is allocated after the end of fixed part of the object and it is not possible to derive some class with instance variables from class containing varying component. In Java language, varying component of object is represented by array, and it is possible to derive any class from it:

```
class string {
    protected int  size;
    protected byte body[];
}

class substring extends string {
    protected int  offs;
}
```

You should avoid such usage of arrays in Java if you want to access the objects from C++ applications.

One more issue is caused by different representation of strings in Java and C++. In Java strings are using Unicode characters and counter, while in C++ strings as usual are zero terminated and consists of 8-bit characters (which can be certainly represented by Java byte type). If you prefer to store in database ASCII zero terminated strings instead of Java Unicode strings, then you can use methods of `pi.Converersion`class to convert array of bytes to ASCII zero terminated string and visa versa.

Due to historical reasons only dynamic arrays and `blob` class from GOODS persistent class library have compatible representation in Java and C++.

7 Examples

It seems to me that the easiest way to become familiar with some product is to look at the examples. That is why I prepared a set of examples of GOODS Java applications.

7.1 Guess an animal

This is very small and simple program, showing all benefits of using OODBMS for storing object in database. In spit of its simplicity, program behaves like it has elements of "artificial intelligence" (really it just collecting information obtained from user and the more information user entered the cleverer the program becomes).

This application is an example of using optimistic model of synchronization. If two or more users simultaneously gives answers for the same question, then only one of them (who finish first) will succeed and other will see the message "Lets try again..." and information, he have entered, will be lost.

This test example also illustrates how C++ and Java applications can work with the same database, Root GOODS directory contains implementation of the same game for C++ guess.cxx, which is compatible with this Java application. You can run both of them simultaneously.

You can start this application by starting server `goodsrv` guess in root GOODS directory and run application itself by java Guess in java directory.

7.2 Graphic editor

This is very small and simple program, showing all benefits of using OODBMS for storing object in database. In spit of its simplicity, program behaves like it has elements of "artificial intelligence" (really it just collecting information obtained from user and the more information user entered the cleverer the program becomes).

This application is an example of using optimistic model of synchronization. If two or more users simultaneously gives answers for the same question, then only one of them (who finish first) will succeed and other will see the message "Lets try again..." and information, he have entered, will be lost.

This test example also illustrates how C++ and Java applications can work with the same database, Root GOODS directory contains implementation of the same game for C++ guess.cxx, which is compatible with this Java application. You can run both of them simultaneously.

You can start this application by starting server goodsrv guess in root GOODS directory and run application itself by java Guess in `java` directory.

7.3 Test of Blob

This example is test for Blob object. It was also used to measure effect of using parallel load algorithm. This test creates very large object (100 Mb) and then retrieves it from database. First time it is made using Blob.play() method, which starts separate thread to load next part of the object in parallel with handling of the current part, and second time object is loaded sequentially part-by-part without any parallelism. Results of running this test at my computers shows that time for parallel loading (134 seconds) is 10 seconds less than time of sequential loading (145 seconds). Handling of object in this example consists only of one `sleep(10)` method, which delays handling thread for 10 milliseconds.

7.4 Test of B-Tree

This is also mostly performance measuring test. Program inserts specified number of records in B-tree, then do several search passes (to make sure that they are still in B-tree:) and then removes all inserted records. It is possible to run several programs in parallel, to check how synchronization works, or to split data between two or more storages to test work of distributed algorithms.