YOODA : C++ user interface

E. Abécassis

NB: This is a preliminary version of the documentation. It is subject to change.

1.0 Presentation

YOODA is an object-oriented database which provides persistence to C++ objects. The programmers develop C++ applications and YOODA tools ensure persistency on their objects. Specific allocators are used to manage persistency. Extensions to C++ are provided to customize this language for database applications. The most important extensions are transaction management, index, dynamic collections and clustering.

A YOODA object is just like a classical C++ object but its identity is not acheive through pointer but through logical object identifier. Those identifiers allow the programmers to access objects across different sessions. Other particularities of YOODA objects will be detailed later.

2.0 YOODA Overview

YOODA is used to manage databases. It allows storing and retreival of objects into a database in a transactional approach.

2.1 YOODA Database

A YOODA database is composed of multiple containers of objects named ‘volume’. Each volume corresponds to an UNIX file.

2.2 YOODA Server

A single database could be managed through multiple servers. A server offers to clients access to one or more volume of the database. It manages objects accesses, concurrency control and reliability. Configuration of a database defined which servers managed which volumes.

2.3 YOODA Volume

A volume is a UNIX file. The maximum size of a volume is 2Gigabytes. It is composed of an allocation map for the volume and of clusters. The number of volumes in a database is 32.

2.4 YOODA Cluster

A cluster is a set of pages inside a single volume. It is used to group objects on the disk. The size of a cluster is limited to the size of a volume. Size of pages could be 4Kbytes or 8KBytes depending of the architecture. The number of cluster is not limited. Cluster can be defined by the administrator of the DB or dynamicly in the application code.

2.5 YOODA Object

An objects belongs to a cluster. An object is a C++ object with a size less than volume size (up to 2Gigabytes). We distinguish short object (size less than one page) and long object (more than one page). This difference is not visible for the user and the only restriction is that long objects are always contains in an integer number of pages.

2.6 YOODA Client

A client is a C++ application which uses persistent objects. To do a YOODA client, you must prepare a database, parse your database schema, and link your code with YOODA library. A client can open a database and so be connected to all the servers. A client can open a database in server manner. Each server of a database is identified by its name. The client can open the database as one of those servers. In such case, volumes managed by the client are not exported but are directly accessed. Other volumes are accessed through real servers.

3.0 YOODA installation

3.1 Environnement variables

There are five environnement variables in YOODA. They must be set before any use of YOODA.

- YOODA_HOME: absolute path for YOODA installation directory

When you load YOODA from your installation tape, you have a directory 'yooda'. The absolute path for this directory must be set in your variable YOODA_HOME.

- YOODA_BASE : name of the current database

When you want to launch client, server or YOODA tools, they always need a base name. If it is not given, they take YOODA_BASE instead. So be sure to correctly set this variable.

- YOODA_CACHE_DIR :Swap directory for YOODA client

Your YOODA clients use a special swap area. This swap file is created as a temporary file in your YOODA_CACHE_DIR directory. Be sure that this directory is local to your client host. The default is /tmp

- YOODA_CACHE_DIR :Swap directory for YOODA client

Your YOODA client must specified size for local memory area. The default is 16 Megabytes

- YOOC_MASTER : hostname of the master host for connection

To connect your servers and clients, YOODA uses a specific connection tool. This tool is described in another document. 'YOOCOM : A communication tool for client/ server architecture'. Simple description is given in the next section.

- YOOC_PORT : Port number used by Connection tool

See 'Communication in YOODA'

Moreover, you must modify one of your default environnement variables :

```
setenv PATH ${PATH}:${YOODA_HOME}/bin
```

3.2 Communication in YOODA : YOOCOM

To connect your clients and your server, YOODA needs a communication tool. This tool is YOOCOM. It is composed of two commands. The first one, called 'yooc_daemon' must be launched on every host where clients or servers could running. It registred every service available on local host. These processes communicate between them to find a server on the network. The second command is 'yooc_ls'. This command is used to get information on every yoocom_daemon launched on the network and every services avalaible. It produces a report on standard output.

There are the two steps to install commnucation tools on your network :

1. Choose one machine as the communication server. Set variable YOOC_MASTER to the name of the chosen host. Set variable YOOC_PORT to a free port. Then launch 'yooc_daemon' as background task on server host. To verify the correct installation, use the command 'yooc_ls'. This command informs you of the current services on the network. If you have an error message, verify your environnement variables and redo.
2. Launch 'yooc_daemon' on every machine on your network where you want to use a client or a server. 'YOOC_PORT' must be a free port on those machines

3.3 Create your database

The first step to build a YOODA/C++ application is to initialise a base. There is a command "yoo_init" which creates a new base. This base will store the database schema and persistent objects.

A base is defined by :

- name : use to connect a client to a server which manages this base

- default volumes directory : it is the directory where the default files for the database are created. One default volume is created at the initialisation. This file contents objects created without location specification (cf. cluster)

- system directory : it is the directory where YOODA stores system files.

- schema directory : it is the directory where YOODA will generate files for your schema.

All of this information are in a system file named "repert.sys". This file must be in the directory specified by the environnement variable "YOODA_HOME" .

The exact syntax of the repert.sys file is :

```
system<basename>.     : <system_directory>
volume.<basename>     : <default_vol_directory>
schema.<basename>     : <schema_directory>
```

example of file "repert.sys" :

```
system.testbase       : /PROJECT/YOODA_BASE/system_dir
volume.testbase       : /PROJECT/YOODA_BASE/base_dir
schema.testbase       : /PROJECT/YOODA_BASE/schema_dir
```

When your file "repert.sys" is created, you can create the base with the command :

```
yoo_init -b <basename>
```

yoo_init can also be used to reinitialise a database.

NB : You will have better performance if you declare volumes and system files on a disk local to you server host.

4.0 Administration tools

4.1 Commands

- yoo_init [+s/-s] -b <basename>

Initialises a new base corresponding to <basename> in file repert.sys. If the base already exists, it asks you if you want to delete it. You must have no server on <basename> to use yoo_init

- yoo_class -b <basename> <schemafile>

Creates classes described in file <schemafile> in the base <basename>. It generates include files for each class described in the schema and an include file <schema>.h.

- yoo_shell -b <basename> [<filename>]

Allows interactive construction of servers, volumes and clusters. It is also the only way to set a class in a cluster.

- yoo_server -b <basename> [-s <servername>]

Launchs the YOODA server <servname> on base <basename>. If the server name is not specified, the server is supposed to be unique and will manage every volumes.

4.2 yoo_init

When you have declared your base in ‘repert.sys’ file, you can use the command yoo_init to create your database. When you have already a database and you want to clean it, you can use this command. In this case, you have two choices :

1. “+s’ : Only delete the data.
2. “-s” : Delete all data, volumes, clusters and classes

The default is “-s”.

4.3 yoo_class

When your database is initialised, you must create a schema. A schema contents the description of your classes.

At the beginning, your base is empty. You must, at least, create one class. You create a a set of classes and later add new classes in your base by using the same command :

```
yoo_class -b <basename> <filename>.sch
```

The file “filename.sch” contents the description of your classes. The first part of the description is the class definition. A class definition looks like C++ except that you must follow some rules (cf class description).

example of a description class file “employee.sch” :

```
class Employee : public Person {
public :
     int               salary ; // C++ int
     Ref<Dept>         dpt; // ref to a persistent department
```

```
        ListX<Project>    proj_l ;

        void increase_salary(int percent) ;
        void migrate(Ref<Department>) ;
    } ;
```

yoo_class generates few files :

- <ref_class>.hxx : for each class defined in <filename>.sch
- <filename>.hxx : contains all the declarations of your file <filename>.sch.
- <basename>.hxx : Unique for a database.
- <basename>.cc : Unique for a database.

4.4 YOODA shell

It is a command shell to manage the organisation of your database. Only few functionnalities are avaible today.

1. Servers

```
create server <servname>
delete server <servname>
move log for server <servname> in <filename>
display servers
display server <servname>*
```

YOODA can manage a distributed database. For this purpose, you can distribute your database into server processes. Each of these servers can manage one or more volumes of data.

2. Volumes and clusters

```
create volume <volname> [in file <filename>]
create volume <volname> [in server <servname>]
move volume <volname> in server <servname>
move volume <volname> in file <filename>
delete volume <volname>
delete volumes
display volumes
display volume <volname>*
create cluster <clustername> [in volume <volname>]
delete cluster <clustername>
delete clusters
display clusters [<clustername>*]
```

Volumes are useful to distribute your base across different file systems. However, your interest is to localise volumes of a database close to the server which manages them.

When you create a volume, you must create a cluster into. When you want to store instances of a class in a specific volume, you just have to declare the class in this cluster.

Clusters are used to group objects in the same set of pages. For example, if you want to create a class Person and a class Vehicle and you know that you will work often on Person and Vehicle at the same time, YOODA offers better performance if you clusterise these two classes in the same cluster. When YOODA will load a page from the disk, it only contains Vehicle and Person and no useless objects.

Creation of a volume without file specification is created in your default volumes directory declared in your database.

Creation of a cluster without volume specification is created in your default volume.

3. Classes

```
move classes <classname> in cluster <clustername>
display classes
display classe <classname>*
```

The yooda allows the administrator to organize the localisation of the classes. When you create a class with yo_class, every instances of this class are created in default cluster in default volume. If you want that new instances will be created in a specific cluster, you can use the command ‘move classes’ to do it.

4. Names

```
display names
display name <name>*
delete name <name>
delete names
```

The external root of your database is the named object. To inspect or delete such names you can use yooda command.

5.0 Description of a class

Each class must have a description. This description contents C++ definition and declaration of collections.

5.1 cluster of a class

For every classes of your schema, you can specify a cluster at any time. This cluster must exist before the link edition of your application. Then, every time you will create an instance of this class, its storage space will be allocated in the set of pages of the specified cluster. The purpose of this mechanism, is to group in the same disk pages, the objects which are used together.

5.2 Definition of a class

In the description file, you must give the complete definition of the class. The YOODA/C++ definition is just like C++ definition except for certain particularities :

- Attributes could be one of the following types

- All C++ types
- StringX
- Reference to a persistent object
- Reference to ListX | DictX (<persitent object>)
- ListX | DictX (<persitent object>)
- Reference to ListX(<int>)
- ListX(<int>)
- Reference to ListX(<double>)
- ListX(<double>)
- Reference to ListX(<StringX>)
- ListX(<StringX>)
- C++ array of authorised type

- Overloading of new and delete are not allowed

- Use of pointer on persistent classes is prohibited

- Size of an object must be inferior to the size of a volume (2^32 bytes)

5.3 collections

If you need collections of class instances in your application, you can freely use it. For example, if you have a department object, it could have as an attribute, a set of employees

You have 2 different types of collections :

- list : An ordinate collection with insertion and suppression capabilities
- Dictionary : An ordinate collection indexed by a user defined key. Dictionary must be declared

```
DICOGEN(<dictname>,<classname>,<attr_method_key> <key_type>)
```

By default, you have ListX of int, double and StringX.

```
ex :    Ref< ListX<int> > l_int = new ListX<int> ;
```

5.4 A complete example

```
ex :
// class description for Person

        class Person {
        public :
             int              age ; // C++ int
             Ref<Person>      live_with ; //Ref to persistent person
             ListX(Person)    l_child ; // list of person
        public :
             void new_child(Ref<Person>) ;
        } ;

// class description for Employee
        class Emp : public Person {
        public :
             int               salary ; // C++ int
             Ref<Dept>         dp ; // ref to a persistent department
             ListX<Project>    proj_l ; // must have been declared

             void increase_salary(int percent) ;
             void migrate(Ref<Dept>) ;
        } ;
```

6.0 YOODA types

6.1 Basic types

To manage consistency, YOODA must detect updates during the execution. To do that, it uses memory mapping. When an object is update, a signal SEGV is received and managed as a write lock on the object.

The conclusion is that you can use almost every basic C++ types.

6.2 Nested objects

You can declare an attribute to be an instance of a class. This class have not to be declared as a persistent class. This object will be local and no persistent references are possible on it. You can access to it only through its father.

6.3 Reference to persistent object

One of the most important feature of YOODA/C++ is the possibility to use persistent pointers on your objects. Your objects will be persistent. So you cannot use standard C++ pointers which have a sense only during a session. Instead of pointer, you can use reference on persistent object.

For every class declared in your schema, YOODA generates another class which is a smart pointer on your class. You can use instances of smart pointer class just as classical C++ pointer. These pointers are intelligent and can always find the referenced object.

You can use smart pointer for an attribute of a class or like a pointer variable in your code. The name of the smart_pointer is Ref<classname>. The template class is :

```
class Ref<type> {
     // constructors
     Ref<type>() ; // empty constructor
     Ref<type>(Ref<type>&) ; // copy constructor
     Ref<type>(<type>*) ; // copy constructor

     // pointer assigment
     Ref<type>&  operator=(Ref<type>&) ;
     Ref<type>&  operator=(<type>*) ;

     // pointer operators
     <type>*     operator->() ;
     <type>&     operator*() ;

     // equal and diff
     int         operator==(Ref<type>&) ;
     int         operator==(<type>*) ;
     int         operator!=(Ref<type>&) ;
     int         operator!=(<type>*) ;
} ;
```

```
ex :
     class Project {
          Ref<dept>        dept ; // ref to a persitent department
          Ref<emp>         leader ; // ref to a persistent Employee
          int              team_size ;
     } ;

     void foo(Ref<emp> new_lead, int new_team_s) {
          Ref<Project> proj = new Project ; // Create a new project
          proj->leader = new_lead ; // copy reference
          proj->team_size = new_team_s ; // copy value
     } ;
```

6.4 Variable string 'StringX'

YOODA offers a type to manage variable strings. The interface of this type is :

```
class StringX {
public :
     // constructors
     StringX() ; // Construction of an empty stringX
     StringX(char *) ; // Copy constructor
     StringX(const StringX&) ; // Copy constructor

     // assignement (value copy)
     StringX&    operator=(char *) ;

     // multiple assignement to avoid previous concatenation
     void assign(char*) ;
     void assign(char*, char*) ;
     void assign(char*, char*, char*) ;
     void assign(char*, char*, char*, char*) ;
     void assign(char*, char*, char*, char*, char*) ;

     // append
     StringX&    operator+=(char*) ;

     // accessing to the ieme element of the string
     char&       operator[](int i) ;

     // get a substring
     StringX     operator()(int index, int count) ;
```

```
    // friend operators
    friend int          operator==(const StringX&, char*) ;
    friend int          operator!=(const StringX&, char*) ;
    friend int          operator<(const StringX&, char*) ;
    friend int          operator<=(const StringX&, char*) ;
    friend int          operator>(const StringX&, char*) ;
    friend int          operator>=(const StringX&, char*) ;
    friend StringX      operator+(const StringX&, StringX&) ;
} ;
```

6.5 YOODA collections

YOODA offers generic collections. Collections avalaible are List and Dictionary. List are defined as template so you can use them on every C++ types. Dictionary could not be defined easily as template. So you must declare dictionaries in your schema files.

```
ex :
    class Person {
    public :
        int             age ; // C++ int
        StringX         name ;
        Ref<Person>     live_with ; //Ref to persistent person
        ListX(Person)   l_friend ; //Ref to a list of person

        Person(StringX name) ;
        void new_child(Ref<Person>) ;
    }

    void foo(Ref<person>& father) {
        ...
        father->l_friend += new Person("john") ; // add element
        father->l_friend += new Person("marie") ;
        ...
    } ;
```

6.5.1 List

To define a list, you must use :

```
    ListX(<typename>) listname ;

class ListX<type> {
public :
    // Constructors
    ListX<type>() ; // create a list
    ListX<type>(const ListX<type>&) ; // copy constructor
```

```
    // Initialise a list with a predefined size. Elements are
    // undefined
    void init(int s) ;

    // value copy.
    ListX<type>&     operator=(const ListX<type>&) ;

    // comparing two lists by value
    int              operator==(const ListX<type>&) ;
    int              operator!=(const ListX<type>&) ;

    // append at the end of the list
    ListX<type>      operator+=(const ListX<type>&) ;
    ListX<type>&     operator+=(const Ref<type>&) ; // for object
    /* or */
    ListX<type>&     operator+=(type) ; // for basic type

    // access to the ieme element
    Ref<type>&       operator[](int) ; // for object
    /* or */
    type&            operator[](int) ; // for basic type

    // insert an element after position pos
    // if pos == COL_TAIL the element is inserted at the end
    void             insert(const Ref<type>&, int pos);// for object
    /* or */
    void             insert(type, int pos) ; // for basic type

    // remove an element
    void             remove(int pos) ;

    // test if an object is in collection, return the position or -1
    int              member(Ref<type>&) ; // for object
    /* or */
    int              member(type) ; // for basic type

    // cardinality
    int              count() ;

    // empty list
    void             clear() ;
} ;
```

6.5.2 Dictionary

To define a dictionary, you must declare in a schema file :

```
DICOGEN(<dictname>,<classname>,<key_type>,<attr_method_key>)
```

The key must be an attribute or a methode of the class <classname>.

The key can be of any type which allow '<' , '>' and '==' operators.

<key_type> is the classname or typename of the key.

```
class Dictname {
public :
      // Constructors
      Dictname() ; // create a reference to a dictionary
      Dictname(Dictname&) ; // copy constructor

      // Return indice. Can only be used with iterator
      int               key(KEY) ;

      // access to the first element in the dict with key
      Ref<type>         getFirst(KEY key) ;
      Ref<ListX<type>> getAll(KEY key) ;

      // Return position of the first element which matches the key
      // You can also address the dictionary as a list
      int               position(KEY key) ;

      //insert an element by using its key
      void              insert(Ref<type>&) ;
      int               insertUniq(Ref<type>&) ;

      //insert every elements of a list by using their key
      void              insert(ListX<type>&) ;
      int               insertUniq(ListX<type>&) ;

      // remove element
      void              remove(Ref<Elem>&) ;

      // remove every elements of key 'Key'
      void              removeAll(KEY& key) ;

      // test if an object is in collection. Return the position or -1
      int               member(Ref<type>&) ;

      // cardinality
      int               count() ;
```

```
        // empty dictionary
        void            clear() ;
} ;
```

6.5.3 Iterator

YOODA offers iterator for collections.

```
// iterator for list and dictionary which are subclasses of ColX
class Iterator<type> {

        // constructor for list
        Iterator<type>(ColX<int>& list, int beg = 0, int end = -1) ;

        // get the current element
        // return NULL if it is the end of the scan
        Ref<type>& get() ;

        // move cursor to the next element.
        Ref<type>& next() ;

        // move cursor to the previous element.
        void prev() ;

        // Reset iterator
        void reset(int begin = 0, int end = -1) ;

        // return 0 if the cursor is at the end position else return 1
        int more() ;

        // return the current position of the cursor
        int where() ;
} ;
```

7.0 Programming with YOODA/C++

7.1 Sessions and Transactions management <SUBJECT TO CHANGE>

- Connection :

```
int yoo_connect(char* <basename>, [char* servername]) ;
```

In your application, you must connect your process with the server. For this purpose, you must use yo_connect. <basename> is the name of the base on which you want to be connected. You could specify servername if you want to start a client in a server mode. It starts a current transaction. The result of yo_connect is 0 if everything is OK. Elsewhere, it returns -1.

- Disconnection :

```
int yoo_disconnect() ;
```

Disconnect you from current base and realize a commit on current transaction.

- Commit :

```
int yoo_commit() ;
```

Commit the current transaction. Every update is archived on the server and your locks are released. it starts a new current transaction.

- Abort :

```
int yoo_abort() ;
```

Abort the current transaction. Every update is discarded and your locks are released. It starts a new current transaction.

7.2 Construction of YOODA/C++ objects

When your base and your schema are ready, you can use the generated file to create an application. Your first need is to create persistent objects. Remember that persistent classes must be accessed only through ref pointer except in a member function.

To create an object which have a YOODA/C++ class you can :

1. Create a persitent object

```
Ref<Person> pers1 = new Person ; // create a persistent person
Ref<Person> child = new(pers1) Person ; // create a persistent person
```

In the first line, the new operator for class Person creates a persitent object by default. The result is a pointer on the persistent object which cannot be used directly. That is why we declare a Ref<Person> to receive the pointer. This object “pers” can be used as a normal pointer on a Person object.

The third line shows creation of an new object which is persistent and stored physically closed to the object passed as parameter of the new. This creation can improve performance of the system.

2. Create a temporary object

```
ex :
```

```
Person* pers = ::new Person ;
Ref<Person> pers = new(YOO_TMP_CLU) Person ;
```

In the first line, we create an object on the classical stack of the process. In the second line, we create an object in the temporary volume managed by YOODA. The only difference is that the second object can be referenced by the same manner as persistent object. Such objects will be automatically deleted at the end of the session.

3. Create a local object

```
ex :
      void foo() {
           ...
           Person pers ; // local object
           ...
      } ;
```

In that case,'pers' is not a YOODA object. The object is created by C++ in the current heap and will be deleted at the end of the scope. There is no way to make it persistent.

7.3 Named Objects

To retreive objects from your database, you must have root objects that you can access without an OID (internal object identifier). To do that, YOODA allows to name objects and get objects by its name.

To name object, you have aspecific function:

```
      int yoo_setName(const Ref<ObjectX>&, char*) ;
ex :
      Ref<Dpt> dpt = new Dpt ;
      yoo_setName(dpt, "R&D") ;
```

To retreive your objects by their name, you must use yoo_getname. It returns 1 if the object exists or 0 if not.

```
      int  yoo_getname(Ref<ObjectX>&, char*) ;
```

ex :

```
      Ref<Dpt> dpt ;
      yoo_getname(dpt, "R&D") ;
```

7.4 Concurrency mode

To manage concurrency, YOODA provides a two-phase locking transaction. Your transaction begins with you connection and ended with the call of yoo_commit, yoo_abort or yoo_disconnect. Except for yoo_disconnect, the end of a transaction is the beginning of a new one.

Locks used depends of the way the transaction is began.

<MUST BE COMPLETED>

7.5 Exception management

To manage errors during the execution of a YOODA session, you can use the YOODA exception management. When an error appends, if the system can detect it, it throw an error. To catch it, you must have declared a try block.

```
try block :
      try {
           <statements>...
      } exception {
           switch (__ERR) {
                        case <error> : <keyword>
                        case...
           }
      }
```

The try block can be declared at every level in your code and can be nested.

To handle the error, you have few keywords

UP : transfert the error to the upper try block

RETRY : redo the statements of the block try

_ERR can be :

```
YOO_ERR_NO_MEM : Problem in memory managment
YOO_ERR_DEADLOCK : Dead lock in the wait graph
YOO_ERR_NULL_OID : Reference to a null object
YOO_ERR_STRING : Out of bondaries in string access
YOO_ERR_COL : Out of bondaries in collection access
YOO_ERR_NAME_OBJ : Name already used.
```

8.0 TODO List

Complete user documentation

Technical documentation

MetaSchema to enable generic access to the objects

Query language

...

9.0 An example

```
// file essai1.sch

class Employe {
public :
 StringX name ;
 int age ;
 int salary ;
 Ref<Dpt> ref_dpt ;

 Employe() ;
 Employe(int _age, int _sal, Ref<Dpt> dpt) ;
 virtual void display() ;
 void new_year(int increase) ;
} ;

DICOGEN(DictEmpName, Employe, StringX, name) ;

// file essai2.sch
#include "essai1.h"

class Engineer : public Employe {
 public :
 Engineer() ;
 Engineer(int _age, int _sal, Ref<Dpt> dpt) :
      Employe(_age, _sal, dpt) {} ;
 virtual void display() ;
} ;

class Leader : public Employe {
 public :
 Leader (int _age, int _sal, Ref<Dpt> dpt) :
      Employe(_age, _sal, dpt) {} ;
 virtual void display() ;
} ;

class Dpt {
public :
 int                  ident ;
 ListX<Employe>       emp ;

 Dpt(int id) ;
 void displayAll() ;
```

```
} ;


// file essai.cc
#include "stream.h"
#include "essai1.h"
#include "essai2.h"

Employe::Employe(int _age, int _sal, Ref<Dpt> dpt) {
 this->age = _age ;
 this->salary = _sal ;
 this->ref_dpt = dpt ;
}

void Employe::display() {
 cout << "\tAge : " << this->age << "\n" ;
 cout << "\tSalary : " << this->salary << "\n" ;
 cout << "\tName : " << this->name << "\n" ;
} ;

void Engineer::display() {
 cout << "Engineer" ;
 this->Employe::display() ;
} ;

void Leader::display() {
 cout << "Leader" ;
 this->Employe::display() ;
} ;

void Employe::new_year(int augment) {
 this->age += 1 ;
 this->salary += this->salary / augment ;
}

void Dpt::displayAll() {
 int size = emp->count() ;
 for (int i = 0; i < size; i++) {
      emp[i]->display() ;
 }
}


void main(int argc, char** argv) {
```

```
 yoo_connect("test") ;

 Ref<Dpt> dpt ;

 if (yoo_getname(dpt, "WORLD") == 0){
      dpt = new Dpt ;
      yoo_setname(dpt, "WORLD") ;
      dpt->ident = 5 ;
      dpt->emp += new Employe(0, 1000, dpt) ;
      dpt->emp[0]->name = "John" ;
      dpt->emp[0]->age = 0 ;
      dpt->emp += new Engineer(1, 1000, dpt) ;
      dpt->emp[1]->name = "Steve" ;
      dpt->emp[1]->age = 1 ;
      dpt->emp += new Leader(2, 1000, dpt) ;
      dpt->emp[2]->name = "Alan" ;
      dpt->emp[2]->age = 2 ;
      dpt->emp += new Engineer(3, 1000, dpt) ;
      dpt->emp[3]->name = "Robert" ;
      dpt->emp[3]->age = 3 ;
 }

 dpt->displayAll() ;

 yoo_commit() ;

 dpt->displayAll() ;

 yoo_disconnect() ;

} ;
```