## Contents

1 The Language Specifications ................................................................. 5
  1.1 Comments ....................................................................................... 5
  1.2 Basic types .................................................................................... 6
  1.3 Enum types ..................................................................................... 6
  1.4 Array types .................................................................................... 7
  1.5 Literal and object types ................................................................. 7
  1.6 Collection types ............................................................................ 8
  1.7 Inheritance ..................................................................................... 8
  1.8 Constraints .................................................................................... 9
  1.9 Referential integrity ...................................................................... 10
  1.10 Methods ....................................................................................... 11
  1.11 Triggers ....................................................................................... 12
  1.12 Indexes ....................................................................................... 12

2 The eyedbdl tool .................................................................................. 14
  2.1 Updating a schema ........................................................................ 15
  2.2 Generates C++ code ....................................................................... 15
  2.3 Generates Java code ....................................................................... 15
  2.4 Generates ODL ............................................................................... 16
  2.5 Display schema differences .......................................................... 16
  2.6 Checking the syntax of an ODL file ............................................. 16

3 Annexes ............................................................................................. 17
  3.1 A simple example .......................................................................... 17
  3.2 A more complex example .............................................................. 18
  3.3 The eyedbdl usage .......................................................................... 20
The Object Definition Language

The EyeDB Object Definition Language (ODL) is a specification language to define the specifications of object types based on the ODMG ODL (but not compliant). ODL is not intended to be a full programming language. It is a definition language for object specifications. Database management systems traditionally provide facilities that support data definition (using a Data Definition Language (DDL)). The DDL allows users to define their data types and interfaces while the Data Manipulation Language (DML) allows to create, delete, read update instances of those data types. ODL is a DDL for objects types. If defines the characteristics of types, including their properties and operations. ODL defines only the signatures of operations defined in C++ and does not address definitions of the methods that implements those operations. Operations defined in OQL can be defined in the ODL. ODL is intended to define object types that can be implemented in a variety of programming languages. Therefore, ODL is not tied to the syntax of a particular programming language.

EyeDB ODL differs from ODMG ODL from several points:

- ODMG ODL defines class attributes, relationships, method signatures and keys. It supports nested classes, typedef constructs, constant definitions and exception hints.
- EyeDB ODL defines class attributes, relationships, method signatures, attribute constraints (notnull, unique, collection cardinality), index specifications and trigger declarations. It does not support nested classes, typedef constructs, constant definitions and exception hints.
- in EyeDB ODL, any type instance can be both a literal or an object. In ODMG ODL, this property is tied to the type: all basic types and user defined struct are literal while interfaces and classes are objects. In EyeDB ODL, any type instance can be an object, even the basic types.
- at last, EyeDB ODL allows to specify whether a method is executed by the server or by the client, and whether it is a class or instance method.

1 The Language Specifications

The basic concept of the EyeDB object model is the class which, as in any traditional object model, model a set of objects of similar properties (attributes) and behaviors (methods). The attributes can be basic types, user types, references, arrays, collections. The methods can be defined in C++ or in OQL (Object Query Language).

ODL allows one to specify classes, attributes, methods, triggers, constraints, enumerate types, indexes and implementation hints.

We are going to introduced in details all the features of ODL.

1.1 Comments

The ODL comments are like in C++:

- mono-line comments: any characters following // and until the end of the line are comments
- multi-line comments: any characters (including newlines) between /* and */ are comments

For instance:
// this is a simple line comments

/ * this is
   a multi line
   comments */

1 ODL is used for shortness to denote EyeDB ODL
1.2 Basic types

The basic types are as follows:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>1-byte integer</td>
</tr>
<tr>
<td>char</td>
<td>1-byte character</td>
</tr>
<tr>
<td>short</td>
<td>2-byte integer</td>
</tr>
<tr>
<td>int</td>
<td>4-byte integer</td>
</tr>
<tr>
<td>long</td>
<td>8-byte integer</td>
</tr>
<tr>
<td>double</td>
<td>8-byte floating point</td>
</tr>
<tr>
<td>oid</td>
<td>8-byte internal object identifier</td>
</tr>
<tr>
<td>enum</td>
<td>4-byte integer</td>
</tr>
</tbody>
</table>

For instance:

```java
class C {
    attribute byte b;
    attribute char c;
    attribute short s;
    attribute int i;
    attribute long l;
    attribute double d;
    attribute oid o;
};
```

Notes:

1. The key word `attribute` is optional:

```java
class C {
    byte b;
    char c;
    // ...
};
```

is correct.

2. The grammar does not allow one to gather several attributes on the same line declaration:

```java
class C {
    attribute char c1, c2, c3; // NOT correct
    // ...
};
```

1.3 Enum types

An enumerate type is denoted by a set of integers mapped to symbols like in C++. The syntax is similar to the C++ syntax, for instance:

```java
enumer enum E1 {
    A, // A == 0
    B, // B == 1
    C // C == 2
};

enumer enum E2 {
    D = 3, // D == 3
    E, // E == 4
    F = 100, // F == 100
    G, // G == 101
    H // H == 102
};

class C {
    attribute int i;
    E1 e1;
    E2 e2;
};
```
1. THE LANGUAGE SPECIFICATIONS

1.4 Array types

The object model supports multi-dimensional fixed or variable size arrays of any type. For instance:

```java
class C {
    attribute byte b_a[4]; // fixed length mono-dimensional array
    attribute char str[]; // variable size mono-dimensional array
    attribute int i_a[3][4][8]; // multi-dimensional fixed size array
    attribute long l_a[][4][8]; // multi-dimensional variable size array
};
```

One particular interesting array type is the array of characters, which can be denoted as `string` as follows:

```java
class C {
    attribute string s; // <=> char s[] (unlimited size string)
    attribute string<32> bs; // <=> char bs[32] (bounded string)
};
```

Note that in a multi-dimensional array, only the extreme left dimension can be variable:

```java
class C {
    attribute long l_a1[][4][8]; // correct
    attribute long l_a2[4][3][8]; // NOT correct
    attribute long l_a2[4][8][3]; // NOT correct
};
```

1.5 Literal and object types

Remember that each object in a set of EyeDB databases has an unique identifier called OID. A literal attribute is an attribute fully included in the class and has no OID, while an object attribute denotes the reference to another object with an object identifier. A reference attribute is denoted by a * or a & symbol. For instance:

```java
class C1 {
    attribute int i;
};

class C {
    attribute C1 l_c1; // literal attribute included in C
    attribute C1 *o_c1; // object attribute referenced by C (or &oc1_1)
};
```

Let `c` an instance of the class `C`.

- `c` includes a literal of type `C1` through the attribute `l_c1`
- `c` can reference an object of type `C1` through the attributes `o_c1`
- if `c` is removed from the database, the attribute `l_c1` is removed at the same time, but the object denoted by `o_c1` is not removed

Do not confuse the * ODL meaning and the * C/C++ meaning: in C/C++, the * type modifier denotes an address to an area of the indicated type instances: it is a pointer to an address. This pointer can be incremented and decremented to change its location in the area.

In ODL, the * denotes a reference to one and only one object, it is why the & token is also accepted, although the meaning of this token is a little bit different in C++.

So, in ODL the construct `C1 **oc1` makes no sense, in the same manner that the construct `C1 &&oc1` makes no sense in C++.

One can have arrays of literal or object as follows:

```java
class C {
    attribute C1 l_c1_1[2];
    attribute C1 l_c1_2[];
    attribute C1 l_c1_3[][10][20];

    attribute C1 *o_c1_1[4];
    attribute C1 *o_c1_2[];
    attribute C1 *o_c1_3[][4][5];
};
```
1.6 Collection types

The EyeDB object model supports three types of collections, set, bag and array. A fourth type, list, will be implemented in a further version:

- A set is an unordered collection of elements of the same type not allowing duplicate elements.
- A bag is an unordered collection of elements of the same type allowing duplicate elements.
- An array is an ordered collection of elements of the same type allowing duplicate elements.
- A list (non yet implemented) is an ordered collection of elements of the same type allowing duplicate elements and where element insertion and removal is efficiently implemented.

An element may be of any type, literal or object and a collection attribute may be a literal or an object, and one can have arrays of collection, for instance:

```java
class C {
    attribute set<int> i_lset;  // literal set of int
    attribute set<C1> l_c1_lset; // literal set of C1 literals
    attribute set<C1> o_c1_lset; // literal set of C1 objects
    attribute set<int> *i_oset;  // object set of int
    attribute set<C1> *l_c1_oset; // object set of C1 literals
    attribute set<C1> o_c1_oset; // object set of C1 objects
    attribute bag<C1> o_c1_lbag; // literal bag of C1 objects
    attribute bag<C1> *o_c1_lbag; // array of literal bag of C1 objects
    // multi-dimensional array of literal bag of set of array of C1 objects
    attribute bag<set<array<set<C1>>>> x[2][3][4];
};
```

The differences between an array collection (i.e. `array<type>`) and an attribute array (i.e. `type []`) are:

- A collection array may exist independently from any class as an attribute array exists only within a class.
- The implementation is very different:
  - One can have a big collection array (thousand or millions of elements) without loss of performance (if the collection is well parameterised, see below). Big attribute arrays are inefficient.
  - Collection array can have “holes” without loss of performance, for instance an element at index 1 and another one at index 1000000 and nothing between. An attribute array with holes are inefficient as they are stored consecutively.
  - A collection array is heavier than an attribute array, and so is not recommended for little size.

1.7 Inheritance

The object model supports single inheritance using the keyword `extends`:

```java
class C1 {
    attribute string c1;
};
class C2 extends C1 {
    attribute string c2;
};
class C3 extends C2 {
    attribute string c3;
};
```

As in usual object conception, an object of class C2 includes the two attributes c1 and c2 and an object of class C3 includes the three attributes c1, c2 and c3.

In the following construct:
1. THE LANGUAGE SPECIFICATIONS

```java
class C4 {
    attribute C1 *oc1;
    attribute C2 *oc2;
    attribute C3 *oc3;
    attribute C1 lc1;
    attribute C2 lc2;
    attribute C3 lc3;
};
```

The attribute `oc1` may be of type `C1`, `C2` or `C3`.
The attribute `oc2` may be of type `C2` or `C3`.
The attribute `oc3` may be of type `C3` only.
The attribute `lc1` is of type `C1`.
The attribute `lc2` is of type `C2`.
The attribute `lc3` is of type `C3`.

1.8 Constraints

The object model supports currently two declarative constraints: notnull and unique. The cardinality constraint on collection is partially implemented and is not currently supported. Non declarative constraints are defined using triggers (see below).

Note that:

- unique constraint cannot be defined on several attributes and
- unique constraint on an attribute needs an index. The index is not automatically created, it must be defined in the ODL (see below) or outside using the `idxcreate` tool.

For instance:

```java
class C {
    attribute string s1;
    attribute string s2;
    attribute string s3;

    constraint<notnull> on s1;
    constraint<notnull> on s2;
    constraint<unique> on s2;
    constraint<unique> on s3;
};
```

The attribute `s1` must not be null.
The attribute `s2` must not be null and is unique in the collection of `C` objects.
The attribute `s3` is unique in the collection of `C` objects.

Constraint and inheritance propagation

By default, constraints are propagated to subclasses, let `C2` a subclass of `C`:

```java
class C2 extends C {
    attribute string c2;
};
```

When one creates an `C2` object, the attributes `s1` and `s2` must not be null and the attributes `s2` and `s3` must be unique.

Important note: the unique constraint applies separately on each class (`C` and `C2`) and not on the set of inheritance class tree. This means that one can have a `C` object with a given value for `s2` and a `C2` object with the same value for `s2`. This is not the expected default behavior and will be parameterised in a next version.

If you do not want to propagate automatically a constraint to the subclasses, you need to use the construct `propagate = off` as follows:
class C {
    attribute string s1;
    attribute string s2;
    attribute string s3;

    constraint<notnull, propagate = off> on s1;
    constraint<notnull> on s2;
    constraint<unique, propagate = off> on s2;
    constraint<unique> on s3;
}

class C2 extends C {
    attribute string c2;
};

The notnull constraint on C::s1 and the unique constraint on C1::s2 will not be propagated to C2, but the notnull constraint on C::s1 and the unique constraint on C::s3 will be propagated to C2.

Constraint on attribute of literal composite type

One can define constraints on attributes of literal composite type attribute, for instance:

class C1 {
    attribute string s1;
    attribute int i1;
};
class C {
    attribute C1 c1;

    constraint<notnull> on c1.s1;
    constraint<unique> on c1.i1;
};

1.9 Referential integrity

The EyeDB object model supports one-to-one, one-to-many and many-to-many relationships.

A relationship between a class A and a class B is materialized by attributes in the two classes of the following types according to the cardinality of the relationship:

- one-to-one: A contains an attribute of type B * and A contains an attribute of type B *
- one-to-many: A contains an attribute of type collection<B *> (collection is a set or a bag) and A contains an attribute of type B *
- many-to-many: A contains an attribute of type collection<B *> and A contains an attribute of type collection<B *>

For instance for a one-to-one relationship:

class A {
    attribute string sa;
    attribute B *b;
};
class B {
    attribute string sb;
    attribute A *a;
};

In the previous case, EyeDB maintains only partially the referential integrity: for instance, one cannot create an object A with an attribute b which refers a non-existent B object. But, if the referenced B object is removed, the attribute b will still reference the removed object.

EyeDB can maintain the referential integrity by indicating the inverse directive in the ODL as follows:
1. THE LANGUAGE SPECIFICATIONS

class A {
    attribute string sa;
    relationship B *b inverse B::b; // or inverse b
};

class B {
    attribute string sb;
    relationship A *a inverse A::b; // or inverse a
};

Note attribute has been replaced by relationship in this case: this is mandatory.
In this case, if the B object referenced by a A object through b is removed, b is set to the null value.
A one-to-many relationship:

class A {
    attribute string sa;
    relationship set<B *> b_set inverse a;
};

class B {
    attribute string sb;
    relationship A *a inverse b_set; ;
};

and a many-to-many relationship:

class A {
    attribute string sa;
    relationship set<B *> b_set inverse a_set;
};

class B {
    attribute string sb;
    relationship set<A *> a_set inverse b_set; ;
};

1.10 Methods

In ODL, one can declare the signature of C++ and OQL methods and one can defined the body of OQL methods. By default, a method is executed on the server side.
A method argument can be any basic type, reference on a composite type or mono-dimensional array of basic or composite type. An argument can be in, out or inout. Argument may be named or unnamed (only type is given), for instance:

class C1 {
    attribute string c1;
};

class C2 {
    attribute string c2;
    int perform(in int size, in string str, out double, in C1 & , inout C2 &);
};

Note that the & symbol may be replaced by the * symbol or no symbol as anyhow only a persistent object (not a litteral) may be passed to a method call.

The C::perform method must be defined in C++ but may be called from OQL or a C++ client. To define a C++ method, refer to the document C++ Binding.

Methods can be overloaded (same name but different signatures), for instance:

class C2 {
    attribute string c2;
    int perform(in int size, in string str, out double, in C1 &, inout C2 &);
    int perform(in double, out string mystr);
};

One can define OQL methods in ODL. In this case, the name of the arguments must be given:
class C2 {
    attribute string c2;
    int append(in string s)
    %oql{
        this.s2 += s;
        return strlen(this.s);
    };
};

The OQL this variable denotes the calling instance.

A method can be an instance method (the default) or a class method (equivalent to C++ or Java static methods). To defined a class method, there are two constructs, using the keyword static or classmethod:

class C {
    static int perform1(in string); // or
    classmethod int perform2(in string);

    instmethod int perform3(in string); // <=> int perform3(in string)
};

If you want to execute a method on the client side, you must use the keyword client as follows:

class C {
    instmethod<client> int perform1(in string);
    classmethod<client> int perform2(in string);

    instmethod<server> int perform3(in string); // <=> int perform3(...)
    classmethod<server> int perform4(in string); // <=> classmethod perform3(...) 
};

1.11 Triggers

Triggers are server methods which are executed when a particular event occurs on an object: before or after creation, update, load or delete.

Like methods, a trigger can be written in C++ or in OQL. On the other hand a trigger has no argument but has a name:

class C {
    attribute string s;

    // C++ triggers
    trigger<create_before> c_b();
    trigger<create_after> c_a();

    trigger<update_before> u_b();
    trigger<update_after> u_a();

    trigger<load_before> l_a();
    trigger<load_after> l_b();

    trigger<remove_before> r_b();
    trigger<remove_after> r_a();

    trigger<create_before> c_b2(); // one can have several create_before triggers

    // OQL trigger
    trigger<create_before> l_a2() %oql{
        if (strlen(this.s) > 100)
            throw "invalid length";
    };
};

1.12 Indexes

Indexes can be either defined in ODL or with the tool eyedbidxadmin. To define indexes on attributes:
1. THE LANGUAGE SPECIFICATIONS

```java
class C {
    attribute string s;
    attribute int i;

    index on s;
    index on i;
};
```

Note that we cannot define one index on several attributes.

Index and inheritance propagation

As constraints, indexes may be or not propagated to subclasses. The behavior is the same as for constraints: indexes are propagated by default to subclasses:

```java
class C {
    attribute string s;
    attribute int i;

    index on s;
    index on i;
};

class C2 extends C {
    attribute long l;
};
```

Indexes are created for `C::s`, `C::i`, `C2::s` and `C2::i`.

Note that the index on `C::s` (resp. `C::i`) is different from the index on `C2::s` (resp. `C2::i`).

To forbid propagation:

```java
class C {
    attribute string s;
    attribute int i;

    index<propagate=off> on s;
    index<propagate=off> on i;
};

class C2 extends C {
    attribute long l;
};
```

Indexes are created only for only `C::s` and `C::i`.

Index on attribute of literal composite type

One can create indexes on an attribute of a literal composite type, for instance:

```java
class C1 {
    attribute int i;
    attribute double d;
};

class C {
    attribute string s;
    C1 c1; // literal composite type

    index on s;
    index on c1.i;
    index on c1.d;
};
```
Index specifications

By default, an index on a number attribute (char, short, int, long and double) is implemented as a BTree, while an index on either a string or a bounded string is implemented as a Hash index.

The differences between BTree and Hash are as follows:

- BTree indexes allows one to retrieve in an efficient way entries with values greater or lesser than a given value. Hash indexes does not allows this in a efficient way.
- On big volume of data, BTree indexes are more efficient with the default parameters than Hash indexes with the default parameters
- On the other hand, for exact match search, a Hash index with good parameters is more efficient than a BTree index
- Usually, Hash index creation is at least four times faster than creating a BTree index

The ODL index specification allows one to change the default index type and parameters of a given target attribute. To set the type of an index on a given attribute, one uses:

```java
class C {
    attribute string<32> s;
    attribute int i;

    index<type = btree> on s; // default is hash: change to btree
    index<type = hash> on i; // default is btree: change to hash
};
```

Important note: one cannot create a BTree index on a non bounded string. One can set implementation parameters for indexes as follows:

```java
class C {
    attribute string<32> s;
    attribute int i;

    index<type = btree, hints = "degree = 64;"> on s;
    index<type = hash, hints = "key_count = 4096; initial_size = 4096;
    extend_coef = 1; size_max = 4096;"> on i;
};
```

2 The eyedbodl tool

The eyedbodl tool can be used to:

- update a database from an ODL file:
  - create a schema
  - add methods, triggers, constraints, indexes to classes
  - remove methods, triggers, constraints, indexes to classes
  - add classes
  - remove classes
  - add attributes
  - rename attributes
  - remove attributes
  - remove classes
- generate C++ stubs from an ODL file or a database containing a schema
- generate Java stubs from an ODL file or a database containing a schema
- generate ODL from a database containing a schema
- display the differences between an ODL file and a database schema
- check an ODL file syntax

For instance, let `schema.odl` an ODL file and `dbtest` a database.
2. THE EYEDBDL TOOL

2.1 Updating a schema
To update a database from an ODL file:
```
eyedbodl -u -d dbtest schema.odl
```
or:
```
eyedbodl --update --database=dbtest schema.odl
```
Important notes:
- All classes defined in the ODL file will be added to the existing schema in the database
- The classes in the database and not in the ODL file will not be removed from the database
- To remove a class from a database, one must use the `--rmcls=class` option. Because of class dependencies, the removal of a class can fail because one needs to remove other classes, for instance collection classes of the class one want to remove. In this case, one must delete classes in the good order.
- To remove an entire schema from a database, one must use the `--rmsch` option
- The methods, triggers, constraint and indexes in the ODL file and not in the database will be added to the database
- The methods, triggers, constraint and indexes in the database and not in the ODL file will not be removed from the database unless the `--rmv-undex-attrcomp=yes` option is given
- The common indexes (on same attributes) in the ODL file and in the database with a different implementation will not be updated unless the `--update-index=yes` is given
- A class C defined in the ODL and in the database with different attributes will have the following behavior:
  - an attribute a in a class C of the ODL, not in the class C of the database will be automatically added to the class C in the database
  - an attribute a in a class C in the database, not in the class C in ODL will be automatically removed from the class C in the database: This operation is not undoable
  - an attribute a in a class C in the database and in the class C in ODL with different types will lead to an update failure

2.2 Generates C++ code
To generate the C++ API from an ODL file:
```
eyedbodl --gencode=C++ schema.odl
```
To generate the C++ API from a database:
```
eyedbodl --gencode=C++ --package=schema -d dbtest
```
For a given `package.odl` ODL file, the generated files are as follows:
- `package.h`, `package.cc`: the generated C++ API to be used in a client program
- `template_package.cc`: an example of a client program using the generated API
- `Makefile.package`: an example of Makefile to compile `package.cc` and `template_package.cc`: make -f Makefile.package will compile and link the generated API and template files
- `packagestubfe.cc`, `packagestubsbe.cc`: stubs for client and server methods
- `packagemthfe-skel.cc`, `packagemthbe-skel.cc`: skeletons for client and server methods

2.3 Generates Java code
To generate the Java API from an ODL file:
```
eyedbodl --gencode=Java schema.odl
```
To generate the Java API from a database:
```
eyedbodl --gencode=Java --package=schema -d dbtest
```
For a given `package.odl` ODL file, the `package` directory contains a Java file for each class defined in the ODL file plus a Java file for each collection template class used as an attribute in classes of the ODL file.
2.4 Generates ODL

To generate the ODL from a database:

```
eyedbodl --genode=ODL -d dbtest # generates on the standard output
eyedbodl --gencode=ODL -d dbtest -o schema.odl
```

2.5 Display schema differences

To display the difference between a schema in an ODL file and a database schema:

```
eyedbodl --diff schema.odl -d dbtest
```

2.6 Checking the syntax of an ODL file

To check the syntax of an ODL file:

```
eyedbodl --checkfile schema.odl
```

Beside these major options, eyedbodl has a lot of extra options as described when running eyedbodl with the -help option.
3. Annexes

3.1 A simple example

Here is a simple example that can be found in examples/C++Binding/schema-oriented/share/schema.odl:

```plaintext
class Address {
  attribute string street;
  attribute string<32> town;
  attribute string country;

  index on street;
};

class Person {
  attribute string name;
  attribute int age;
  attribute Address addr;
  attribute Address other_addrs[];
  attribute CivilState cstate;
  attribute Person * spouse inverse Person::spouse;
  attribute set<Car *> cars inverse owner;
  attribute array<Person *> children;

  int change_address(in string street, in string town,
                   out string oldstreet, out string oldtown);

  index on name;
};

class Car {
  attribute string brand;
  attribute int num;
  Person * owner inverse cars;
};

class Employee extends Person {
  attribute long salary;
};

enum CivilState {
  Lady = 0x10,
  Sir  = 0x20,
  Miss = 0x40
};
```
3.2 A more complex example

Here is a more complex example used for the management of biological databases:

```cpp
enum StatusType {
  running = 0,
  done = 1
};

class File {
  attribute string path;
  attribute string name;
  attribute string desc;
  attribute set<Import_ctx *> imported_in inverse Import_ctx::file;
  constraint<nonnull, propagate=on> on name;
};

class Import_ctx {
  attribute File * file inverse File::imported_in;
  attribute Import * import inverse Import::contexts;
  attribute StatusType status;
  attribute string comment;
  attribute int32 count;
  attribute int32 elapsed;
  attribute float average;
  attribute string start_date;
  attribute string last_update;
  constraint<nonnull, propagate=on> on file;
  constraint<nonnull, propagate=on> on import;
};

class Import {
  attribute Db * related_db inverse Db::imports;
  attribute string database_name;
  attribute string cvs_tag;
  attribute set<Import_ctx *> contexts inverse Import_ctx::import;
  attribute string comment;
  attribute bool deletable;
  instance_method <client, called_from=OQL> time_interval getElapsed()
  constraint<unique, propagate=on> on database_name;
  constraint<nonnull, propagate=on> on database_name;
  constraint<unique, propagate=on> on cvs_tag;
  constraint<nonnull, propagate=on> on related_db;
  index<propagate=on> on database_name;
  index<propagate=on> on cvs_tag;
};

class Db {
  attribute string name;
  attribute string title;
  attribute int32 version;
  attribute set<Import *> imports inverse Import::related_db;
  attribute array<File *> files;
  attribute set<Db *> divisions;
  attribute Import * official;
  instance_method <client, called_from=OQL> string [] get_db_names();
  constraint<unique, propagate=on> on name;
}
```
constraint<nonnull, propagate=on> on name;

index< propagate=on> on name;
};
3.3 The eyedbodl usage

The usage of the eyedbodl is as follows:

```plaintext
[-db-class-prefix=<dbprefix>] [--attr-style=implicit|explicit] [--dynamic-attr]
[-gen-class-stubs] [--class-enums=yes|no] [--c-suffix=<suffix>] [--h-suffix=<suffix>]
[-export] [--down-casting=yes|no] [--gencode-error-policy=status|exception] [--attr-cache=yes|no]
[-no-cpp] <odlfile>|-|-d <dbname>|--database=<dbname> [<openflags>]
```

```plaintext
eyedbodl --gencode=Java --package=<package> [-output-dir=<dirname>] [--output-file-prefix=<prefix>]
[-schema-name=<schname>] [-class-prefix=<prefix>] [-db-class-prefix=<dbprefix>]
[-attr-style=implicit|explicit] [-down-casting=yes|no] [-gencode-error-policy=status|exception] [-export]
[-cpp=<cpp>] [-cpp-flags=<flags>]
[-no-cpp] <odlfile>|-|-d <dbname>|--database=<dbname> [openflags]
```

```plaintext
eyedbodl --gencode=ODL -d <dbname>|--database=<dbname> [--system-class]
[-o <odlfile>] [openflags]
```

```plaintext
eyedbodl --diff -d <dbname>|--database=<dbname> [-system-class] [openflags] [-cpp=<cpp>]
[<cpp-flags=<flags>] [-no-cpp] <odlfile>|-|
```

```plaintext
eyedbodl -u|--update |-update -d <dbname>|--database=<dbname> [-db-class-prefix=<dbprefix>] [openflags]
[<schema-name=<schname>] [-rmv-undef-attrcomp=yes|no] [-update-index=yes|no]
[-cpp=<cpp>] [-cpp-flags=<flags>] [-no-cpp] [-rnclass=<class>] [-rmsch] [odlfile]|-
```

```plaintext
eyedbodl --checkfile <odlfile>|-
```

```plaintext
eyedbodl --help
```

One must specify one and only one of the following major options:

```plaintext
--gencode=C++ Generates C++ code
--gencode=Java Generates Java code
--gencode=ODL Generates ODL
--update|-u Updates schema in database <dbname>
--diff Displays the differences between a database schema and an odl file
--checkfile Check input ODL file
--help Displays the current information
```

The following options must be added to the --gencode=C++ or Java option:

```plaintext
<odlfile>|-|-d <dbname>|--database=<dbname> Input ODL file (or - for standard input) or the database name
```

The following options can be added to the --gencode=C++ or Java option:

```plaintext
--package=<package> Package name
--output-dir=<dirname> Output directory for generated files
--output-file-prefix=<prefix> Output file prefix (default is the package name)
--class-prefix=<prefix> Prefix to be put at the begining of each runtime class
--db-class-prefix=<prefix> Prefix to be put at the begining of each database class
--attr-style=implicit Attribute methods have the attribute name
--attr-style=explicit Attribute methods have the attribute name prefixed by get/set (default)
--schema-name=<schname> Schema name (default is <package>)
--export Export class instances in the .h file
--dynamic-attr Uses a dynamic fetch for attributes in the get and set methods
--down-casting=yes Generates the down casting methods (the default)
--down-casting=no Does not generate the down casting methods
--attr-cache=yes Use a second level cache for attribute value
--attr-cache=no Does not use a second level cache for attribute value (the default)
```

For the --gencode=C++ option only

```plaintext
--namespace=<namespace> Define classes with the namespace <namespace>
--c-suffix=<suffix> Use <suffix> as the C file suffix
--h-suffix=<suffix> Use <suffix> as the H file suffix
--gen-class-stubs Generates a file class_stubs.h for each class
```
3. ANNEXES

--class-nums=yes Generates enums within a class
--class-nums=no Do not generate enums within a class (default)
--gencode-error-policy=status Status oriented error policy (the default)
--gencode-error-policy=exception Exception oriented error policy
--rootclass=<rootclass> Use <rootclass> name for the root class instead of the package name
--no-rootclass Does not use any root class

The following options can be added to the --gencode=ODL option:
--system-class Generates system class ODL

The following option must be added to the --update|--u option:
-d <dbname>|--database=<dbname> Database for which operation is performed

The following options can be added to the --update|--u option:
<odlfile>|- Input ODL file or '-' (standard input)
--db-name=<schname> Schema name (default is package)
--db-class-prefix=<prefix> Prefix to be put at the beginning of each database class
--rmv-undef-attrcomp=yes|no Removes (yes) or not (no) the undefined attribute components
(constraint, index and implementation). Default is no
--update-index=yes|no Updates (yes) or not (no) the index with a different
implementation in the DB. Default is no
--rmscls={<class>} Removes the given class list
--rmsch Removes the entire schema

The following options must be added to the --diff option:
-d <dbname>|--database=<dbname> Database for which the schema difference is performed
<odlfile> The input ODL file for which the schema difference is performed

The following options can be added to the --diff option:
--system-class Performs difference on system classes also

The following option must be added to the --checkfile option:
<odlfile>|- Input ODL file or '-' (standard input)

The following options can be added when an <odlfile> is set:
--cpp=<cpp> Uses <cpp> preprocessor instead of the default one
--cpp-flags=<cpp-flags> Adds <cpp-flags> to the preprocessing command
--no-cpp Does not use any preprocessor

Common Options:
-U <user>|@, --user=<user>|@ User name
-P [<passwd>], --passwd=[<passwd>] Password
--host=<host> eyedbd host
--port=<port> eyedbd port
--inet Use the tcp_port variable if port is not set
--dbm=<dbmfile> EYEDBDBM database file
--conf=<conffile> Configuration file
--logdev=<logfile> Output log file
--logmask=<mask> Output log mask
--logdate=on|off Control date display in output log
--logtimer=on|off Control timer display in output log
--logpid=on|off Control pid display in output log
--logprog=on|off Control programe display in output log
--error-policy=<value> Control error policy: status|exception|abort|stop|echo
--trans-def-mag=<magorder> Default transaction magnitude order
--arch Display the client architecture
-v, --version Display the version
--help=eyedb-options Display this message