15 Schema migration

During his long consulting career, we met customers from a great variety of applications – coal mining, silicon circuits, telephone switches, networks, and stock exchange data – and the most typical customer was a programmer who brought a big, totally messed up program and wanted to clean it up and add persistence to it. Why didn't he use the persistence right from the beginning?, we asked. Because it looked so simple in the beginning. The customer thought they'd never need persistence, they'd never need to manage the data.

There is no real-life program that would not require changes of its classes and data structures during its life time, but especially during the development. Software design is a learning process in which the customer gradually learns what he or she wants, the programmer gradually grasps the essence of the problem and, usually after some experimentation, finds a suitable solution.

Whether you are developing a new program or supporting software already used by numerous customers, it is most annoying if you have to change the existing classes and/or data structures, and all the old storage files suddenly become unusable.

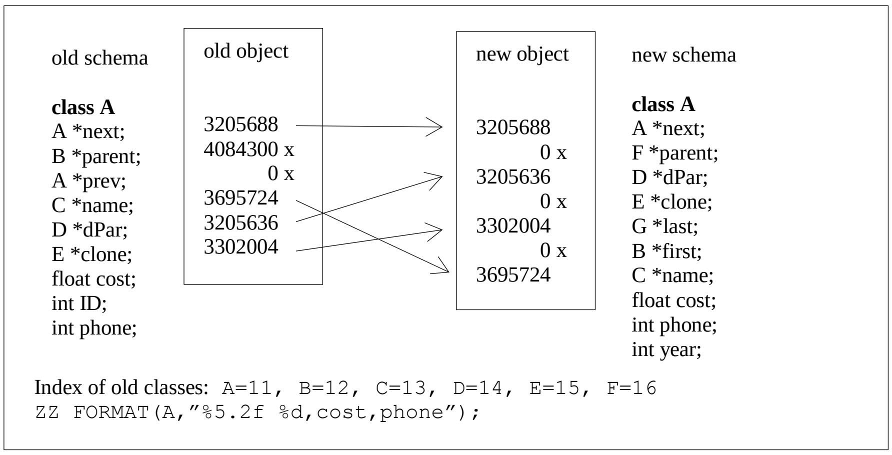
We already said that the block of Association statements is an equivalent of the database schema. It describes the overall data organization[1] and the data structures from which it is composed. When working with persistent data we face the same problem database designers do: We need to handle, as transparently and automatically as possible, the changes in the schema, and this is called *schema migration*.

Association statements not only describe the data structures; they also give us all the participating classes. However, associations alone are not the schema yet – inheritance and individual members of the classes may also change, be added or removed. Let's follow an example of how the ASCII serialization does it. The key for the entire process is that the persistence is integrated with the data structure library. The library provides the information about all the pointers in the game, their names, types, and locations – the same information other langugages provide through reflection.

ASCII serialization does not write out this information for every pointer as Java serialization does. Instead, it writes out a table for each class at the beginning of the file, and when writing objects to the disk it refers only to its type. When reading pointers from the file, only pointers with matching name and type move into the target object – see Fig.6.1. All this happens during the reading process, before

[1] Sometimes the term *framework* is used for this.

swizzling the pointers, and is probably similar what happens for example in Java. As in Java, DOL does this totally transparently and automatically.



Index of old classes: `A=11, B=12, C=13, D=14, E=15, F=16`

```
ZZ FORMAT(A,”%5.2f %d,cost,phone”);
```

Fig.6.1: Schema translation in DOL – reading pointers. Pointers marked with x did not find a match.

Storage and retrieval of members which are integers, floats, or characters is controlled by the ZZ_FORMAT statements – see Listing 6.1. This format controls both the read and write functions, so the match is guaranteed. For these values, DOL does not provide detailed matching; members which are in the ZZ_FORMAT statement are assumed to be on both sides, but their order does not have to be the same. There also may be other members on either side.

Assuming that, on the file, each old class is represented by its index, the disk record for the old object from Fig.6.1 would be

```
  11 340100 1 6        = class A, addr. 340100, 1 object, 6 pointers
  3205688 4084300 0 3695724 3205636 3302004     = all pointers
  18.99 6138362327                              = cost, phone
```

Listing 6.1 shows a problem with three simple classes participating in several data structures, using DOL syntax.

Listing 6.1: Original data organization

```
class Library {
    ZZ_EXT_Library;
};
ZZ_FORMAT(Library,””);
class Book {
    ZZ_EXT_Book;
    int pages;
    float cost;
};
ZZ_FORMAT(Book,"%d %f",pages,cost);
class Author {
     ZZ_EXT_Author;
};
ZZ_FORMAT(Author,"");

ZZ_HYPER_SINGLE_COLLECT(books,Library,Book);
ZZ_HYPER_SINGLE_COLLECT(,authors,Library,Author);
ZZ_HYPER_SINGLE_LINK(toAuthor,Book,Author);
ZZ_HYPER_NAME(bookName,Book);
ZZ_HYPER_NAME(authName,Author);
```

The full working listing is in orgc\test\schemamig – read readme.txt for information how to compile it and run. List6_1.cpp creates some test data and stores it in the ASCII mode in file1. We want to make serious changes to this design:

- remove *pages* from Book;
- replace *SINGLE_COLLECT books* by *HASH allBooks;*
- add *phone* number to the Author;
- add a new class Publication, and derive Book from it;
- add SINGLE_COLLECT<Library,Publication>.

Replacing something by something else while keeping the same names creates very difficult situations. Instead, we will remove *LinkList books,* and add a new association, *Hash allBooks*.

This is a bigger change than most of us would trust a computer to do automatically. Listing 6.2 shows the original design with marked up changes.

The file conversion requires three simple steps:

(1) Expand your source with all the features you want to add but, temporarily, keep there the features you want to remove – Listing 6.2. Open the data (file1) in the ASCII format, using the old ZZ_FORMAT, then save it in the binary format (file2).The reason is that DOL does not allow to read data in one ZZ_FORMAT and write it in a different ZZ_FORMAT.

(2) Open file2 in binary, and save it in ASCII using the new ZZ_FORMAT statements (file 3).

(3) Remove the features you do not want, and open file3 in ASCII using the new ZZ_FORMAT. Both data and your source have been converted; now you can save the data in any way you want.

Listing 6.2: Intermediate data organization includes all the additions without removing the features we eventually want to remove. Note that we are still using the old, original ZZ_FORMAT statements.

```
class Library {
    ZZ_EXT_Library;
};
ZZ_FORMAT(Library,"");
class Publication {            // introducing new class
    ZZ_EXT_Publication;
public:
    int year;                             // new member
};
ZZ_FORMAT(Publication,"%d,year");
class Book : public Publication { // adding inheritance
    ZZ_EXT_Book;
    int pages;                              // remove it
    float cost;
};
ZZ_FORMAT(Book,"%d %f",pages,cost);
class Author {
     ZZ_EXT_Author;
     int phone;                            // new member
};
ZZ_FORMAT(Author,"");


ZZ_HYPER_SINGLE_COLLECT(books,Library,Book);                // remove
ZZ_HYPER_SINGLE_COLLECT(publications,Library,Publication); // remove
ZZ_HYPER_HASH(allBooks,Library,Book);                    // add this
ZZ_HYPER_SINGLE_COLLECT(authors,Library,Author);
ZZ_HYPER_SINGLE_LINK(toAuthor,Book,Author);
ZZ_HYPER_NAME(bookName,Book);
ZZ_HYPER_NAME(authName,Author);
```

IMPORTANT:
In steps 1 and 2, you have in memory and side-by-side both the old and the new data. The new, still unused organizations are initialized as empty. This allows you to add any conversion which cannot be done automatically. In this case, you can

traverse Collection *books*, and move all the books to Hash *allBooks*. Here is the snippet from Listing6_2.cpp which does that:

```
Library *lib; Book *bk;
books_iterator bit;

// transfer LinkList to Hash
allBooks.form(lib,100); // form hash table with 100 buckets
bit.start(lib);
ITERATE(bit,bk){
   allBooks.add(lib,bk); // add bk to Hash
   books.del(lib,bk);    // remove it from LinkList
}
```

For the code that runs the entire sequence, look at `listings\list6_1_4\readme.txt` and the source files in the same directory.