Robust and portable software design and implementation based on an object-oriented database interface

P. Heinckiens, H. Tromp & G. Hoffman

Department of Information Technology, University of Ghent, St. Pietersnieuwstraat 41, B-9000 Ghent, Belgium

ABSTRACT

An object-oriented database interface is presented and it is shown how it can be used to design and implement robust and portable software systems of a high degree of quality. It hides the underlying data storage from the application developer and allows him to model his problem domain and to design the application at a much higher level of abstraction. Persistent objects can be treated in exactly the same way as temporary objects and persistent class hierarchies can be designed. The practical use of the database interface is also illustrated: a general purpose data management and entry system is presented, which also hides the graphical user interface.

INTRODUCTION

Due to the ever growing user requirements, as well as the explosive evolution of available hardware resources, software systems have become extremely complex. The need to manage that complexity, and still deliver robust software, is now a major challenge for the software engineer. To ensure software quality, reusability, portability, maintainability and extendibility, object-orientation has emerged as an important enabling technology. It allows the software designer to decompose his problem domain in a set of objects, which offer a well-defined interface to the other components of the system, while their internal structure is completely hidden. The implementation of these objects can be fully verified and validated. Moreover, by using inheritance mechanisms, specialised objects can be derived from more generic ones. This leads to optimal software reusability. The application developer can design software with a minimum of effort, and still do that reliably, since he builds his application on top of an object space, which has been validated independently and offers a high level of abstraction. Portability is also a major asset. Operating system and user interface dependencies can easily be hidden within
objects. If the software is to migrate to a different platform, only the internal structure of a limited number of objects must be modified. As long as their interface to the outside world is left unchanged, they can be incorporated in the new software system without a major disruption of its operation.

In this paper, the ideas presented above are illustrated by a concrete example. Database systems have become a more and more important subsystem of modern software systems. The growing need for data storage and manipulation has led to a tremendous proliferation of database software of a largely varying degree of complexity, and an overwhelming variety of application program interfaces. The result is that far too often the software engineer has to cope with a lot of details about the storage and manipulation of persistent data structures, instead of being able to concentrate on the application itself. This stands in sharp contrast with the requirements of current software design practice. From the point of view of the application software designer, data storage is nothing else then a technique to store objects. His view on the data is essentially object-oriented, and should be strictly separated from the actual mechanism used to store and retrieve the data. The latter may range from simple file structures, such as standard C-ISAM, it may be a simple database interface such as CTREE or Codebase, it may be a relational database system or a fully developed object-oriented database system. As long as the interface, offered to the application program, is identical for all these data storage mechanisms, it becomes perfectly feasible to port the application program between platforms and from one database system to another. The same application can be used, without any changes, on low cost platforms, such as a PC’s using a simple and cheap file system to store the data, as well as within sophisticated client-server architecture’s, based on top-of-the-bill relational or object-oriented database management systems. In this way, existing investments in software development are fully protected and reliability is strongly enhanced.

The database interface to be presented in this contribution meets the requirements stated in the previous paragraph. It frees the application designer from adhering to a single database application program interface, the choice of which is often a mission critical decision, but not really related to the application. If it must be made in an early design phase, it might become a serious burden later on. By using the database interface to be presented, the programmer can fully concentrate on the real essence of his work, without having to be burdened by routine tasks, or even having to know whether a database is involved or not.
THE DATABASE INTERFACE : THE USER PERSPECTIVE

In this section, the database interface mentioned in the introduction is presented. It serves mainly as an example to demonstrate how the principles of object orientation can be used to offer a consistent, reliable and portable interface, on top of which applications can be built that exhibit inherent quality, both to the software developer and to the user of the software system that is being developed. Some technical implementation details will be discussed, although the main focus will be on the features offered to application developers. More specific implementation details are given by Heinckiens et al [1]. In this, as well as in subsequent sections, extensive C++ code fragments will be used to illustrate the underlying ideas.

Datasets
The major element of the database interface is the abstract class Dataset, which should be regarded as a persistent container class. To the outside world, it offers a simple and consistent interface, which is both platform and database independent. This implies that it encapsulates all calls to the low level engine, as well as the internals of the physical storage mechanism. The application developer, who uses class Dataset, becomes (almost) unaware of the fact that he is working with a database. Class Dataset takes care of all database operations in a way that is completely transparent for him. The interface to class Dataset is shown below.

```cpp
class Dataset {
private :
    static DatabaseManager db_manager;
    DatabaseFile *dbf_file;
    dbf_field *dbf_fld[MAX_fields];
    FilterCollection dbf_filter;
public :
    // functions that provide access to the database buffer
    void w_field(uchar *field, uchar *str);
    void w_field(uchar *field, double val);
    void w_field(uchar *field, long val);
    void r_field(uchar *field, uchar *str);
    void r_field(uchar *field, double *value);
    void r_field(uchar *field, long *value);
    DataSet(name = NULL);
    DataSet();
    uchar *who() {return dbf_file->filename();}
    int open();
    int close();
    int erase();
    int set_index (const dbf_field& fld);
    virtual int seek (dbf_field &fld, void *var);
    virtual int skip(long n);
    virtual int go(long n);
    virtual int go_top(); // true if success
    virtual int go_bottom();
    virtual int append();
    virtual int write();
```
The internal structure and functionality of class *DataSet* have been extensively validated and verified. Its inherent reliability and its simple interface ensure that the developer can fully concentrate on the essence of his own problem, without having to care about or to be burdened with routine operations. All this enhances the speed of the software development process significantly, and makes is it far more reliable.

**Database fields**

One of the fundamental design issues is how objects and their constituent subobjects (members) are to be stored in the database. Although the actual storage mechanism should be of no concern to the developer, its impact on the design of the objects, specific to the application, must be carefully considered, in view of reusability and portability.

A number of available class libraries (see e.g. Gorlen et al. [2]) and programming languages require each class to provide a read() and write() method. This is enforced by deriving all classes from a single base class (the traditional *Object*), which implements virtual read() and write() functions. Besides the burden of having to provide read() and write() functions for all classes, this approach has some major drawbacks (see Ellis et al. [3]).

As the standard C and C++ data types (int, double, etc.) have no read or write functions, special provisions will have to be made to store these objects in the database. Further, if third party libraries are used, the probability that these libraries share our common base object is minimal. This means that they too will have no read or write functions that can be accessed via polymorphism used on *Object*. Moreover, if the third party library has also a common base object, name conflicts and unexpected side-effects are likely to occur. It can also be argued that read() and write() functions have in fact very little to do with the object itself. Consider classes such as *Date* or *Point*. Although it can often be necessary to store dates or points in a database, the classes themselves are not directly associated with databases. It is therefore not really their job to deal with writing and reading. Moreover, the implementation of the storage of complex objects (such as images) is often a job for database specialists (see e.g. Hughes [4]), who are not necessarily the same people who develop the classes to be used in an application program.

For all these reasons we opted for a alternative method. With each class, instances of which are to be stored in the database, a separate field class is associated. This specifies how the class must be read from or written to the
database. All these field classes are derived from a common baseclass `dbf_field`, the interface of which is shown below.

```cpp
class dbf_field {
protected :
    char *name;
    FIELD_TYPE type;
    int width;
    int dec;
    PH_INDEX_REF indexkey;
    DataSet *dbf;
public :
    dbf_field (DataSet *base, char *nme, FIELD_TYPE ftyp,
               int wdth,
               int dec, char *ndx);
    dbf_field (dbf_field& fid);
    char* fieldname() {return name;)
    virtual write()=0;
    virtual read()=0;
    virtual seek(void *var);
    // Specify how to write, read and seek a field
    virtual operator char*() const
    // etc...
};
```

For each storable data type, a corresponding field class is derived and the member functions `read()` and `write()` are overloaded. Consider for example the class `char_field`, which specifies how to write strings to the database:

```cpp
class char_field : public dbf_field {
    char *str;
public :
    char_field (char *txt, DataSet *base, char *nme, int
               wdth, char *ndx = NULL): dbf_field (base, nme,
               F_CHAR, wdth, 0, ndx)
    {
        str = txt;
    }
    int write() {
        dbf->w_field(fieldname(), str);
        return 1;
    }
    int read() {
        dbf->r_field(fieldname(), str);
        return 1;
    }
    virtual operator char*() const {return str;}
};
```

This approach has several advantages. As the storage method is completely separated from the object itself, we can use our technique also for the built-in data types and for classes found in third party libraries. By grouping all storage methods in their own dedicated class hierarchy, we can also extend these classes with other database related functions (e.g. `seek()`, `say_as_string()`, etc.) without overcomplicating the related object. The result is a concise and
uniform framework, which allows the developer to describe formally how application specific objects are being stored and retrieved, without having to care about the internal details of the actual storage mechanism. The next subsection explains in some more detail how this is achieved.

Reading and writing objects
The reading and writing of objects from/to storage is abstracted by the member functions `DataSet::read()` and `DataSet::write()`, which send in a polymorphic way a read (or write) message to each (derived) `dbf_field` that is coupled to this `DataSet` object. All these derived `dbf_fields` have now their own overloaded read function that specifies how this object must be read from the buffer. The complete details of this reading and writing process are beyond the scope of this paper. It should be noted, however, that the overloaded `dbf_field::read()` member functions can vary from extremely simple implementations (e.g. for a `char_field`) to very sophisticated ones, where e.g. links are made with several other `DataSet` objects. An example of such a read function will be presented in a subsequent section.

THE USE OF THE DATABASE INTERFACE

While in the previous section, the main components of the object-oriented database interface where presented, we shall now illustrate how it can be used to build robust and portable applications.

Direct versus derived instances of `DataSet`
Class `DataSet` class could be used directly to access a database, by creating instances of it. This is demonstrated in the following example, where a customer database is accessed.

```cpp
void test ()
{
    char name[31];
    char street[31];
    DataSet dbf ("customer");
    char_field f_name (name, &dbf, "NAME", 30);
    char_field f_street (street, &dbf, "STREET", 30);
    dbf.go_top();
    for (int i=0; i<200; i++) // list first 200 customers
    {
        cout << i << " : " << name << " " << street << endl;
    dbf.skip(1);
    }
    
Suppose we are interested in the customer's name and street, which we want to access through the variables `name` and `street`. By using two `char_fields` `f_name` and `f_street`, those variables are linked to the `DataSet` instance `dbf`. In the loop, which lists the first 200 customers, there is no visible reference to the
database anymore, except for an extremely simple skip instruction. Each iteration, the system automatically makes sure that the variables *name* and *street* contain the correct values stored for the next customer. There was also no need to explicitly open or close the database. A lot of routine tasks are done automatically.

So far, the implementation details of the database were hidden (a similar approach was presented by Murphy [5]), but we are still confronted with the fact that we are working with a database. In order to hide the database concept itself, no direct instances of class *DataSet* should be created. It should rather be used as a base class for object-oriented-database objects. In this way it becomes not only possible to hide the database concept, but also to apply the principle of inheritance to databases and to create persistent inheritance trees, as shown by Heinckiens [6] and as will be described later in this paper. As an example of a derived *DataSet* object, consider a class *Customer*.

```cpp
class Customer : public DataSet {
  private:
    char _id[7];
    char _name[31];
    char _street[31];
    City _city;
    char_field f_id;
    char_field f_name;
    char_field f_street;
    city_field f_city;
  // etc...

  public:
    Customer() :
      DataSet("customer"),
      f_id(_id, this, "ID", 6),
      f_name(_name, this, "NAME", 30),
      f_street(_street, this, "STREET", 30),
      f_city(_city, this, "ZIP", "CITY")

    char* id() { return _id; }
    char* name() { return _name; }
    char* street() { return _street; }
    int id(char* str);
    int name(char* str);
    int get();
    istream& operator>>(istream& strm) { get(); return strm; }
  // etc...
};
```

Now, the dbf_fields and the variables have become members of the derived *DataSet* class. The buffer variables and the dbf_fields are hidden in the private part of the class and an application using the *Customer* class sees no longer how this class gets its data. It does not even have to know that the class is coupled to a database. For example, to specify a customer either by his id code or by his name, special functions are provided, which by themselves are straightforward to implement. As the following code fragment demonstrates, a
simple seek statement is sufficient to set all variables for a customer. The search operation that is performed, goes unnoticed.

```c
int Customer::id (char *str) // Specify a Customer by id.
{    return ( seek (f_id, str) == FOUND );
}

int Customer::name (char *str) // Specify by name.
{    return ( seek (f_name, str) == FOUND );
}

int Customer::get() // Let user enter a Customer.
{    if (no_empty_get(_id) // returns false if empty
        return ( seek (f_id, _id) == FOUND );
    else
        return select_in_menu (f_name);
    }
```

Once derived *DataSet* classes are implemented and validated for all objects, relevant to the problem domain of the application, they can be used as easily as built-in data types. As the following program fragment demonstrates, the programmer sees no longer any difference between an instance of a *Customer* and an ordinary data type, such as *double*.

```c
void customer_test ()
{    double number;
    char some_text[31];
    Customer _cust;
    cin >> number >> some_text >> cust;
}
```

When this function is executed and the user is prompted to enter a customer, the following happens: first the user is asked to enter the customer's id. If an empty code is entered, a scroll box appears with all known customers (these are fetched from the database). The user can now make his selection. The corresponding customer is automatically fetched from the database. Was the id code not empty, then the corresponding customer is searched for. It is important to notice that this whole input process is hidden from the application programmer. This is achieved by overloading the >> operator.

**Inheritance of DataSets**

Inheritance is maybe the most important features of object-orientation which largely enhances software reusability, and in consequence reliability. It allows to express common behaviour and structure of the constituent objects of different, but closely related applications. Object orientation makes it possible to implement and test that common portion only once. Later changes to the
software, including bug fixes, are then automatically propagated to all applications based on it. This practice stands in sharp contrast with the more traditional approach, where an application which is similar to one already existing, is created by copying the existing code and modifying it, a strategy that leads to huge maintenance and life-cycle development problems.

Inheritance in object-oriented programming means that common object behaviour is implemented in a base class, from which more specialised classes are derived. The latter implement specific behaviour, while inheriting everything that was already implemented in the base class. Due to the concept of a **DataSet** class, inheritance rules can be applied also to databases. In this way, persistent inheritance trees can be created, and all the benefits of inheritance, as well as the resulting reusability, become available at the database level.

Consider for example class **Customer**, presented earlier. It contains all functionality that is common for all kinds of customers. If a more specialised **Customer** object is needed for a dedicated application (e.g. for dentists, hairdressers), it can simply be derived from the more generic customer class, as is shown in the following code fragment.

```cpp
class DentistPatient : public Customer {
    Teeth _teeth;
    Date _birth;
    Teeth_field f_teeth;
    Date_field f_birth;
public :
    // functionality specific for a DentistPatient.
};
```

### Multiple DataSet instances

The level of abstraction provided by the **DataSet** class has some far reaching consequences. As the programmer is no longer aware of the fact he is using a database, he also does not know when tables might be opened and closed. Another extremely useful characteristic is that, as is the case with "ordinary" variables, several instances of a same **DataSet** class can be created (and thus be active) at the same time, each of course with its own view on the underlying database file. Consider for example:

```cpp
void test ()
{
    Customer a, b;
    a = "Fiers";
    b = "Pauwels";
    do
    {
        cout << a.name() << " " << b.name() << endl;
    }
    while ( (a.skip(1)!=dEOF) && (b.skip(1)!=dEOF) );
}
```
The implementation of the underlying functionality is certainly not straightforward. Both \( a \) and \( b \) access the same database file. As this happens in the same process, and as each process has only one record pointer in a database table, it is clear that \( b \) moves also \( a \)'s physical record pointer (as it is the same one). This means that a multiple stream capability had to be implemented, where each stream represents an instance of the same \( DataSet \) class.

**Classes DatabaseFile and DatabaseManager**

The possibility of having multiple instances of the same \( DataSet \) implies that the same database table may be opened several times simultaneously. Many operating systems put a limit on the number of files that may be open at the same time. As the application programmer no longer knows which variables are linked to a database and which are not, and as he certainly does not know how all these database tables are linked together, he has no way to keep track of this number. Therefore, two special classes \( DatabaseFile \) and \( DatabaseManager \) were introduced to automatically take care of all these bookkeeping operations. These two classes are invisible for the application programmer, who has only access to the \( DataSet \) class. Both classes will be discussed in somewhat more detail, in order to illustrate how the functionality described above is implemented in a robust way.

With each database table corresponds exactly one instance of the class \( DatabaseFile \). This class keeps track of all details associated with a database table and abstracts all operations that can be performed on it (e.g. opening, closing,...). With each instance of \( DatabaseFile \), several instances of \( DataSet \) can be associated. It is the \( DatabaseManager \) that takes care of assigning a correct \( DatabaseFile \) to each \( DataSet \). This is illustrated in fig.1. The functionality and information, common for all instances of a certain \( DataSet \), is grouped in the \( DatabaseFile \) object (e.g. the index tags, the number of records, etc.). On the other hand, each \( DataSet \) instance can have a different record pointer, a different record filter, a different index tag selected, etc.

Class \( DatabaseManager \) manages all active database tables. When a new instance of a \( DataSet \) is created, it issues a request to the \( DatabaseManager \) to associate it with a \( DatabaseFile \). The \( DatabaseManager \) also makes sure that all operating system restrictions are met at all times. This means for example that it keeps track of the number of open files and that it implements a least recently used algorithm to close "old" files in order to open new ones.
NAVIGATING THROUGH DATASETS

After the database interface, and its direct usage have been discussed, we are now ready to present a class that was built on top of the database interface, and combines its functionality with that of a graphical user interface. The result is a general-purpose data management and entry system, which offers a high level of abstraction to the application developer, both in view of the manipulation of persistent objects, and of the presentation of data towards the end user.

A frequent operation in data manipulation is navigation through a DataSet. All applications need at least the following functionality: skipping forwards and backwards, searching, editing, deleting, updating, and adding data records. Given the requirements of reusability, it makes sense to abstract that functionality by a common base class **DataNavigator**. Due to the interaction with the end user in the operations mentioned above, a user interface must also be incorporated in class **DataNavigator**. The interface of **DataNavigator** is shown below:

```java
class DataNavigator : public DataSet {

private :

    void do_selection(int selection);
    int _new_record;
    int _records_deleted;

public :

    DataNavigator(char *dbf_name);
    virtual init() {return 0;} // blank the record
```
virtual void show() const {}; // display the record
virtual void defaults() {} ; // set a record to it's default value
virtual edit() {return 0;} // edit the record
virtual search() {return 0;} // seek a record
virtual execute(uchar *w_name);

protected :

prev_record();
next_record();
seek_record();
virtual delete_record();
edit_record();
new_record();

If a specific class is derived from DataNavigator, only the functions show() and edit() must be re implemented. These functions specify how to display and to get an object. They can use generalised get() and put() functions, which abstract the way an object is displayed on the screen. Class DataNavigator also provides for the rest of the user interface functions (e.g. opening the window, the menu functions, etc.). By hiding most of the user interface concepts, class DataNavigator not only simplifies the program development, but it also improves portability across different GUI-platforms. As an example, consider once again another implementation of class Customer, but this time as a child of DataNavigator.

class Customer : public DataNavigator {
private :
  char _name[31];
  char _street[31];
  City _city;
  char_field f_name;
  char_field f_street;
  city_field f_city;
  // etc...
public :
  Customer() :
    DataNavigator ("customer"),
    f_name (_name, this, "NAME", 30),
    f_street (_street, this, "STREET", 30),
    f_city (&_city, this, "ZIP", "CITY")
  {}
  char* name() { return _name; }
  char* street() { return _street; }

  void show() const;
  void edit();
};

Customer::show() const
{
  say (win_customer.name, name());
  say (win_customer.street, street());
  say (win_customer.city, _city.name());
  // etc...
As can be seen, all what had to be done was to specialise the user interface functionality, by overloading two very simple functions show() and edit(). What the end user sees on the screen as a result is shown in fig.2.

It is clear that an extremely limited amount of code, based on a robust and independently verified set of classes, can exhibit a very rich behaviour, the reliability of which is based on a set of certified software technologies.

![Fig.2. Illustration of the user interface implemented in class DataNavigator.](image)

**SAMPLE APPLICATION : AN INVOICING SYSTEM**

Another, more extensive, example of the usage of the basic components of the database interface will be presented, more precisely an invoicing system that allows to set up invoices, to be sent out to customers. Abstracts of annotated...
C++ code will be used to illustrate how the application developer can rapidly write the code needed to build a reliable application.

For all objects in the problem domain, which are to be stored in the database, appropriate derived *dbf_field* classes should be defined. The outline of a customer field class is shown below.

```cpp
class customer_field : public dbf_field {
    Customer *cust;

public:
    customer_field(Customer *var, DataSet *base, char *nme, char *ndx = NULL)
        : dbf_field(base, nme, F_CHAR, 6, 0, ndx) {cust = var;}

    write() {
        dbf->w_field(fieldname(), cust->number());
    }

    read();  // etc...
};
```

In the implementation of `customer_field::read()`, the link to the customer object is made through the customer's ID. In (1) the customer's ID is fetched from the internal record buffer, and in (2) we seek the customer object with this ID. Two important points must be noted here:

- in (1), due to the use of `r_field()` (a member of `DataSet`), `customer_field::read()` is totally independent of the database system used;
- the application programmer does not know that any link with another `DataSet` is made, thus certainly not that it is made through the ID. All he knows is that the invoice-`DataSet` contains a customer object. Nowhere he has to bother how exactly this customer is stored in the `DataSet`.

The use of *Customer* objects as members of *Invoices* is illustrated by the following code fragment.

```cpp
// Invoicing program (simplified version)
class Invoice : public DataSet {
    double total;
    long number;
    Date date;
    Customer customer;    // (1)
    Stock item[Max_items]; // (2)
    long_field f_number;   // (3)
    date_field f_date;
    customer_field f_customer;
    stock_field f_item[Max_items];
```
double_field f_total;
public:
    Invoice();
    Create();
    List (Customer& cust);
    // etc...
};
Invoice::Invoice () :
    DataSet ("invoice"),
    f_number (&number, this, "NUMBER", 6),
    f_date (&date, this, "DATE"),
    f_customer (&customer, this, "CUSTOMER")
{}  
Invoice::Create()
{
    if (customer.get())
    {
        while (item.get())
        {
            total += item.get_price();
        }
        customer.update_statistics (total);
        writeO; // (4) make the invoice persistent
    }
} // list all invoices sent to a certain customer
Invoice::List (Customer& cust);
{
    if (search (f_customer, &cust) == FOUND) // (5)
    {
        cout << "Invoices of " << cust.name () << endl;
        while (customer == cust) // (6)
        {
            cout << number << " : " << total << endl;
            skip();
        }
    }
}

When looking at this code fragment more closely, a number of things can be noted. By using Customer and Stock objects in (1) and (2), the programmer uses DataSets without being aware of it. Moreover, in (2) he uses multiple instances of the same DataSet. In (4) the invoice object is made persistent, but nowhere we had to specify how to do this. This is all done automatically by using derived dbf fields (3). Although the search operation in (5) is relatively complicated, we didn't have to specify how to perform it. We didn't even have to select the right index key. In (6) two Customers are compared, each of which is an instance of the same DataSet.

CONCLUSION

Up to now, a significant number of practical applications have been developed using the database interface, described above. Most of them are situated in the data processing sector, some are expert systems. The feedback obtained from these projects let the system evolve to a product that has been proven to be useful in practical software development.
During the development of these applications, a few remarkable observations were made. The development time was reduced by a factor 10 or more. The size application program source was reduced by a factor 8 to 9. The extendibility was simplified a lot. The bug rate went down enormously. We obtained a reusability of about 80%. The system proved to be very well suited for multimedia applications: due to the concept of dbf_fields, it was as easy to store pictures and sound as it was to store ordinary data.

A number of features, offered by the database interface, where not discussed in this paper, such as queries, multi-user capabilities, filters. It has been shown, however, how object-orientation in general, and the database interface in particular, can contribute to the development of robust and portable software, that exhibits inherent quality.

REFERENCES