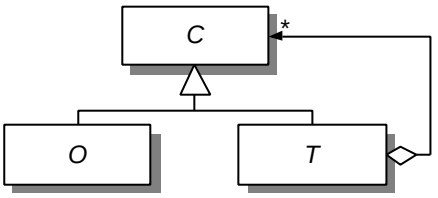
Experiences with object oriented development in PL/SQL

COT/4-18-V1.4



Centre for Object Technology

Centre for
Object Technology

Revision history:	V1.3	2000-05-13	Ready for COT review
	V1.4	2000-02-11	Ready for publication

Author(s): Allan R. Lassen, RAMBØLL
Jacob Steen Due, RAMBØLL

Status: Final

Publication: Public

Summary:

This paper presents experiences gained by introducing object-oriented development on top of a normal procedural language (PL/SQL) and a relational database system (Oracle8). The background for the experiences has been the development a fairly large system.

The Centre for Object Technology (COT) is a three year project concerned with research, application and implementation of object technology in Danish companies. The project is financially supported by The Danish National Centre for IT-Research (CIT) and the Danish Ministry of Industry.

Participants are:
Maersk Line, Maersk Training Centre, Bang & Olufsen, WM-data, Rambøll, Danfoss, Systematic Software Engineering, Odense Steel Shipyard, A.P. Møller, University of Aarhus, Odense University, University of Copenhagen, Danish Technological Institute and Danish Maritime Institute

1. Introduction

Object oriented development has received much acclaim in recent years as the cure for many of the problems that have been seen in software development. However one area that has never received much attention is how systems using relation databases should (and could) take advantage of this paradigm, and how the transition from 'traditional' development to an object oriented development method could be performed.

In this paper we describe our experiences with a hybrid approach addressing the problems in moving towards object oriented development from a more traditional database-centric method. Note that we do not claim to have used object oriented development, but we have introduced aspects of object oriented development into our traditional (functionality decomposition based) approach.

1. The paper will introduce the overall architecture of our system as well as the central concepts we wanted to support.
2. The general introduction is followed by a detailed description of how we implemented the architecture in the PL/SQL programming language.
3. Finally we will discuss our experiences with the approach and evaluate the advantages and disadvantages as we have seen then in practice.

The basis of our experiences was the development of a standard system framework within the field of mobile phone network management. The main functionality was for setting up and maintaining the configuration of the radio network for GSM mobile communication. We had (at RAMBØLL) developed a first version of such a system and were about to embark on the next generation of the system, improving and expanding the functionality. We expected to use approximately 25 persons in almost a year for the development.

Network management is in many ways a very complicated task and the development of such a system faces many obstacles. From our earlier experiences we knew that the requirements for such a system changes and evolves very rapidly, and that we needed to improve our software development process in order to succeed in the market:

- The 'time to market' of any given new feature had to be improved quite dramatically. New features were required at a much faster rate than we were able to supply them leading to a competitive disadvantage. One important aspect of 'time to market' in our situation was the ability to respond quickly to new requirements and to produce several new versions of the system each year. This meant that our new development process had to support an iterative approach.
- Due to the rapidly developing requirements, we needed to employ a high number of developers to work on the system development. This situation created a problem, as the current method became quite inefficient when more than 15 - 20 developers were involved at the same time.

Looking at the main areas of the improvements, we agreed to base the system on a component-based architecture in order to move forward.

Our participation in COT [1] inspired us to move towards an object oriented methodology. We believed that the methodology could further improve the efficiency in the development process by providing better descriptions, and by allowing us to use newer technology in the future - such as object oriented programming languages. We also felt that using an object oriented encapsulation in our design and implementation would help us prepare for change, compared to the functional decomposition based approach we had been using earlier.

However, to move a team towards new technology is not an easy task and the experiences from other projects indicated that going 'all the way' in one step was doomed to failure. In order to minimise the risks we divided the task into two main challenges:

- First, we had to find out who was available for participation in the project. How inexperienced were the developers in using object-orientation, and how could we teach them to learn and use the concepts in a correct and efficient manner. This is referred to as the conceptual challenge in the project.
- Secondly, we had to consider the technological challenge. How did we implement the system? Which tools and languages should we use?

In order to address the challenges we decided to keep the technological platform that the developers were familiar with. In this case we kept the Oracle8 database as the persistent store and PL/SQL and Oracle Developer/2000 as the development tools. Anybody who knows these tools also knows that they are not exactly object oriented. Even though the version of the tools we wanted to use had some object oriented features, we had learned from experience [2] that the maturity of the features wasn't sufficient for us.

The main reason for choosing PL/SQL was not that it is a particularly advanced language, but it has a very good integration to the database, it is portable across platforms (as long as there is an Oracle database on that platform), and as mentioned before we had a good base of developers that were well experienced in using the language.

Therefore, we ended up with the challenge of using object-orientation as conceptual background without using object oriented technology for implementation. This paper describes our experiences.

2. Object oriented development in PL/SQL

Two concepts known from object oriented development were introduced to support the development of our system and the many developers in the project.

- We defined a component-based architecture in order to manage the design and development. Actually we decided we needed to divide the system into components, and then found out that this matched well with object oriented thinking.
- Encapsulation of the database structure was introduced to prevent tight coupling between the components. We switched terminology and used concepts like "object class" and "association" to describe our architectural decisions. This meant that the focus was no longer at the database tables.

This chapter describes how we used the concepts mentioned above.

2.1 Architecture

Our system consists of a number of distinct components called functional areas (FA). Examples of functional areas are general areas like User Management and Error Management. Furthermore specific functional areas are defined for important domains in the system - for instance a key feature of the system is a rule engine and thus we have a Rule Management area.

Our system ended up containing 20 functional areas ranging in size and complexity by several magnitudes. There was no ‘minimum size’ requirement for a functional area and in practice the most experienced member of the team defined the functional areas at a very early stage (based on previous experience). Although a few changes were made subsequently this division actually remained stable throughout the development period.

We feel quite sure that at this point the object oriented 'purists' will object and claim that this is a purely functional division and that it has nothing whatsoever to do with object orientation. And to some extend they would be right, but the point here is that the functional area encapsulated functionality **and** data, and that it functions as a single best home for functionality. The functional areas are in many respects similar to the packages that UML uses and are as such only one step of the way. Inside each functional area are classes that implement the areas public interfaces. So in our process the definition of functional areas is simply the first step.

As can be seen from the above example we use the term ‘component’ in a rather loose fashion. When speaking of components we do not refer to a specific technical specification of the term (e.g. a CORBA object or an EJB). To us components are encapsulated parts of our system, each with its own purpose and behaviour.

In the architecture, each functional area owns a part of the database. The only way to access the functionality in a functional area is through services and view interfaces.

A **service** is a functional task that is being offered to other functional areas. The services were implemented as PL/SQL package procedures and functions.

A **view interface** is a data format that is agreed upon between functional areas. The interfaces should be implemented as database views and in some cases file formats. The introduction of view interfaces implemented as database views were mainly caused by our concern that performance might suffer if we lost the ability to perform joins between tables in different functional areas. However, the use of database views was to be limited to the cases where performance would otherwise be compromised. Interestingly enough we ended up without any view interfaces because performance was not compromised – but when we started the development we were almost sure we would need them.

The physical implementation of tables in the database is encapsulated through the FA public services and view interfaces as illustrated in Figure 1.

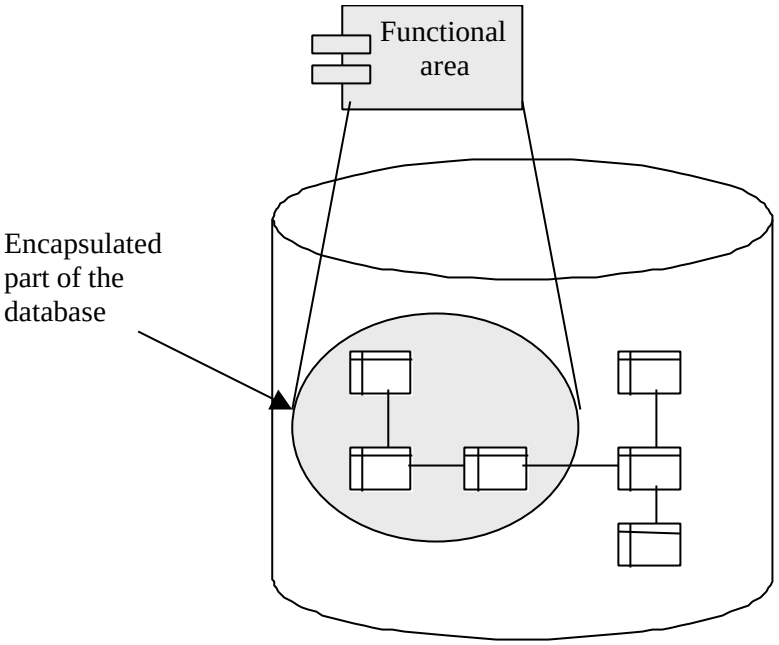


Figure 1: Database implementation is encapsulated by functional area

A functional area consists of all aspects of the functionality i.e. the user interface, client-side functionality, server-side functionality and database elements. One FA owns each dialog in the user interface, but the dialog can use services of other functional areas.

Typically a functional area encapsulated a few tables – 5 or 6 at the most – corresponding to one main object class and with perhaps a few specialised classes (either derived from the main class or as inner classes in the main class).

We wanted a clear division of the processing and followed these rules:

- The primary business logic was implemented in the server, which means that the actual data processing is performed on the server side. Our main goal in doing this was to ensure that it would be possible for us to move the business logic to a third tier later, if we decided to upgrade to technologies like EJB or similar.
- The client side only handles the presentation and capturing of data and events.
- The access to database tables was encapsulated so that change in the database structure would require fewer changes in the code (compared to direct access to the tables).

2.2 Introducing object classes

The architecture is based upon our wish of encapsulating the database structure within a functional area. We would like the communication between functional areas to operate at a logical level and thus hide the physical details on how the data are stored in the relational database. The advantages of the encapsulation are:

- It is easier to use and understand the data from another FA.

- Each FA can restructure their internal implementation including the physical database structure for instance to optimise performance without affecting the other FAs.

We would also like the functional areas to communicate safely and support it using proper data structures. Object classes as known from object oriented development have proven to be well suited for this purpose. Therefore, we introduced techniques in PL/SQL that could produce some of these benefits.

For each functional area we identified their domain object classes and defined PL/SQL record types to represent the attributes of each object class. The basic functionality to create, modify and delete objects was standardised and implemented as stored procedures. On top of this, more complex business logic including the services, were implemented.

The encapsulation with object classes was implemented in various database packages.

2.3 Implementing the architecture

Each functional area was divided into modules as illustrated in Figure 2. Modules on the server side were implemented as database packages. Packages that reside in libraries on the client have been excluded from the figure because they are not important in this context.

Figure 2 shows all the modules in a single FA. The FA in the figure has two user interface parts so the figure does not show how the interaction between different FAs. This is explained in the following but since it is important to understand we will mention it briefly here.

The only access to a FA is through the **Public Service API**. The **Public Data Type** package contains the various types used to describe the classes (see sections 3.1, 3.2 and 3.3), while the Public Service API contains the procedures and function implementing the public methods on those classes.

All other modules mentioned on the figure are **internal** to the FA.

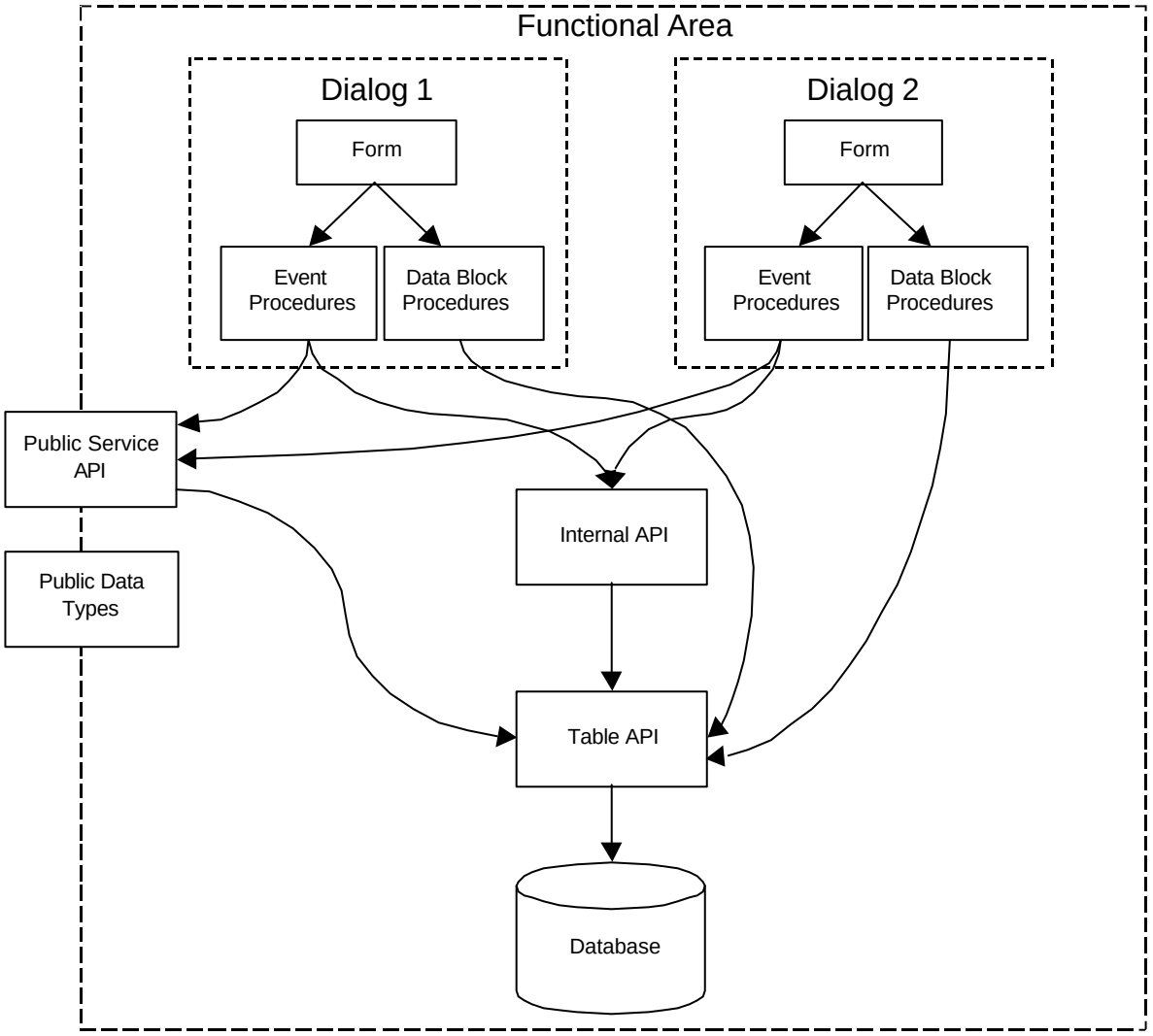


Figure 2: Modules in a functional area

Functional areas communicate solely through the interface specified in the **Public Service API** and the **Public Data Types**. This is illustrated in figure 3 below.

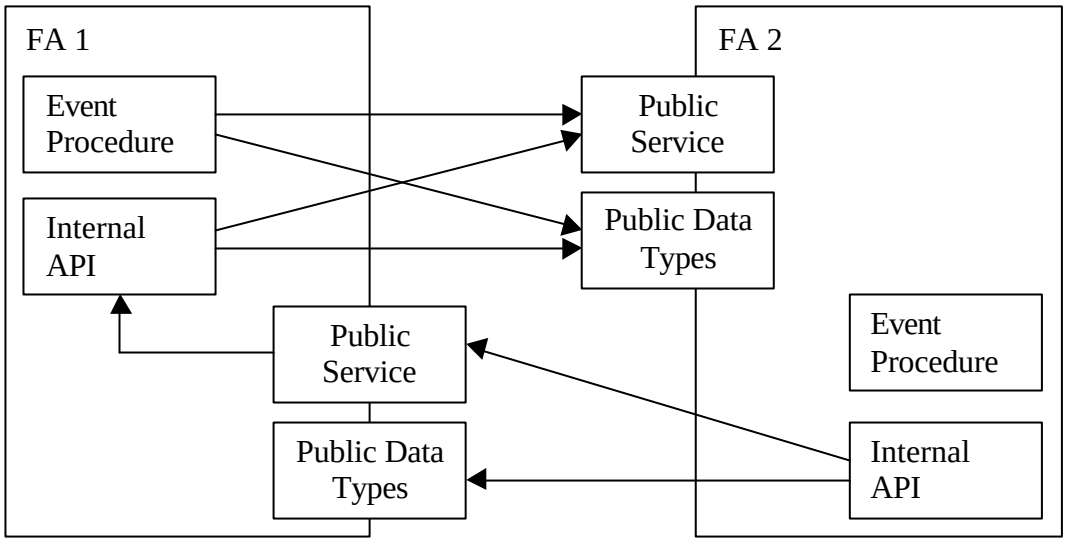


Figure 3 Communication between functional areas

The remainder of this paper deals only with the server related aspects of the architecture. Although we have put quite an effort into defining standards for the user interface side we do not feel they are as fundamental as the changes we have introduced on the server side.

2.3.1 Table API

The primary purpose of the Table API is to encapsulate the database tables inside the functional area. The API is implemented in a database package containing procedures performing basic operations like querying, adding and updating data in a certain table. In principle each procedure maps to a single SQL statement on a certain database table.

The Table API is **internal** to the functional area. The reason is that the procedures are close to implementation of the physical database tables. If they were public, it would put severe restrictions on the ability to change their representation. A functional area usually does not allow other areas to perform all kinds of operations e.g. other areas may not be allowed to delete the data.

2.3.2 Public Data Types

A functional area usually defines new record types to be used in the communication with other areas. These data types are used for parameters in the Public Service API.

The data types correspond to object classes and are built, from a logical point of view, on the data model. We wanted to hide how the logical data model had been transformed into a physical model of database tables, and we certainly did not want to populate the simple record types from the Table API. Introducing data types, collecting data that logically belong together, made the code easier to read.

For readers more familiar with traditional entity-relationship description it is fair to say that the public data types correspond to the entities found in that notation.

The public data types cannot be defined in the Public Service API package because of problems with cross-references (a purely technical problem that is caused by the PL/SQL interpreter). This is exemplified by the following example.

FA1 defines Type1 and uses Type2 defined in FA2
FA2 defines Type2 and uses Type1 defined in FA1

If you try to implement this in Oracle with the types declared in the same packages as the methods using them you will get a very interesting result. If you make a mistake (e.g. a syntax error) in either package specification you will not be able to drop or recreate any of the packages (without the help of Oracle support).

2.3.3 Public Service API

The Public Service API contains the procedures and functions that are public to other functional areas. They include both simple services resulting in a single SQL statement

and more complex services involving several SQL statements. The services correspond to methods on object classes[1].

The Public Service API use procedures from the Table API and Internal API to implement a service. It may also use services from other functional areas.

2.3.4 Internal API

The Internal API contains the procedures and functions that are internal to the functional area in mention. Basically the API follows the same principles as the Public Service API. However PL/SQL does not offer proper facilities to distinguish between the public and internal procedures of a functional area. Therefore, we decided to separate them in different packages and by naming convention[2] we can see if a package is public or internal.

The Internal API will usually use procedures from the Table API to implement the functionality. It may also use services from the Public Service API in its own or other functional areas.

2.3.5 Dialog event procedures

For all user interfaces, the rule is that as little business functionality as possible must be built into the Forms[3] modules.

To ease the mentioned principle, we introduced a separate database package to hold the stored procedures that implement business logic for a certain Forms module. The package is dedicated to the Forms module and other modules do not use it in anyway. It is considered to be an integral part of the dialog. In case of changes in the dialog, the package can be changed in a safe way and without taking other modules into consideration.

When implementing a dialog, the Forms module handles the presentation and capturing of data and events. When processing an event that performs business processing on the data, the Forms module calls a stored procedure in the database package that implements the actual functionality. In principle an event like pressing a pushbutton simply calls a stored procedure.

2.3.6 Dialog data block procedures

The stored procedures to handle data blocks in Forms modules are a special kind of procedures that controls adding, deleting and updating data from the data block to the database tables. They also include a locking strategy. We did not want the Forms modules to access the database tables directly, but to use data block procedures.

[1] When we use the term object class we refer to the generic term, not to the data types called object in Oracle8

[2] In our case internal packages used the string '_int' as the end of the package name.

[3] A Forms module is a user interface module in Oracle Developer/2000.

3. Implementation in PL/SQL

Since we have decided to use PL/SQL as our implementation language we had to devise ways of implementing the encapsulated objects. We needed some way of identifying a specific object as well as the attributes and methods for objects of that particular class.

This section of the paper describes the techniques we used to attain that goal, but since experience shows that an example is often the way towards better understanding we have included short examples in each section[4].

We have decided not to include an introduction to the PL/SQL language in this paper. There are numerous books and presentations on PL/SQL available all of which present the language better than we can do in the limited space available in this paper.

3.1 Identifier record

We define an *identifier record* to identify an object instance of a class in our model. Thus the record is object identification i.e. a kind of pointer to a certain object instance. The definition of the identifier record for a class called `my_class` is like

```
type my_class_type is record (
  internal_id           <internal_id_type>
);
```

where `<internal_id_type>` is a simple data type like a number.

Using identifier records to represent objects provides type safety compared to using simple data types to represent object identification. In the version of PL/SQL we were using it is not possible to get the compiler to distinguish between subtypes of the same atomic type, so implementing the identifier simply as a text would not give any type checking at compile time (or at run time for that matter). Using records however will prevent you from assigning a person object to a department variable.

The internals of the identifier record is for the designer of the FA to define. It must contain sufficient information to retrieve the data in the database belonging to the object. If the data of an object instance is saved in one row in a table it is natural to have a field in the identifier record holding the primary key of that table.

As an example the identifier record definition of an employee could look like this:

```
type employee_type is record (
 employee_id           number);
```

[4] We have dug deep into our creative minds and come up with the example of employee and department FA's (or classes).

3.2 Attributes and property record

For each object class we construct a *property record*. This is a record that holds the attributes of the object class and it only includes the *real* attributes of the object class. In a database table you will find foreign keys that are not real attributes of the object class and they are *not* be included in the property record.

The property record is defined like

```
type my_class_prp_type is record (
  key                   varchar2(10),
  attr1                 attr1_type,
  ...
);
```

In the functional area of the object class `my_class` we provide a function to retrieve the attributes of `my_class`. These attributes are returned in a property record.

Looking at the employee from before the property record might look like the following:

```
type employee_prp_type is record (
  key                   varchar2(20),
  FirstName             varchar2(40),
  LastName              varchar2(40),
  Address               varchar2(200)
);
```

3.2.1 Key string

The property record always contains a `key` field that represents the object identification as a string. The functional area that is responsible for the object class provides functions to map a key to/from an identifier record. Thus it is up to the FA to decide a proper string representation of the identifier record.

The purpose of the key string is to be able to get the identifier record of a certain property record. Due to limitations[5] in PL/SQL we cannot have an identifier record inside the property record. Instead we represent it as a string inspired by the object identifications of CORBA objects. The string is the same as the key returned from the `get_key` function described in section 3.6[6].

3.2.2 Property indicator record

In many situations the developer only wants to work with a subset of the attributes in the property record. In order to be able to express this we construct a *property indicator record*. This record contains a boolean value for each attribute in the object class. The

[5] We cannot create a table of record types with record types.
[6] A typical implementation in our case was would be <FA name><internal ID>, or even just the Internal ID

boolean value indicates whether the attribute in mention is considered by the various operation accepting indicator records.

The introduction of the property record is for performance issues only since this makes it possible to cut down on the amount of data passed around in the system. The property record mechanism is therefore in no way essential to the method.

The property indicator record is defined like

```
type my_class_ind_type is record (
  attr1                boolean,
  ...
);
```

Looking at the employee the property indicator record might look like the following:

```
type employee_prp_type is record (
  key                  boolean,
  FirstName            boolean,
  LastName             boolean,
  Address              boolean
);
```

3.3 Using identifier records and property records

A very important point regarding the identifier record types is that the fields of the record *are to be considered private to the functional area*. In other words the application developer should not mingle with the internals of an identifier record. The reason is mainly to preserve the conceptual integrity – if a developer started using the implementation of the identifier record this would then lock the implementation, and prevent the owning FA from changing it.

The property record is the correct way to access the attributes of an object class. The identifier record must be treated as a black box to the developer whereas the property record is a white box. This rule is illustrated in Figure 4.

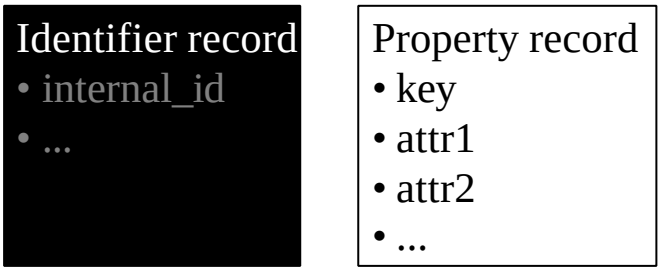


Figure 4: Accessing fields

This means that you use the identifier record to gain access to the attributes in the form of the property record.

3.4 Associations

If object class A has a many-to-one or one-to-one association to object class B a function will be implemented in the functional area of A like:

```
function get_B(p_A in A_type)
  return B_type;
```

For the (by now classical) employee – department example this would mean that the employee class has a function to get the department of that employee:

```
function get_department(pEmployee in employee_type)
  return department_type:
```

If it is allowed to update the association then there will also be a procedure like:

```
procedure set_B(p_A in out A_type, p_B in B_type);
```

If class A has a one-to-many or a many-to-many association to class B there will be a function in the functional area of B like (where `B_tab_type` is a collection of B-class objects):

```
function find_Bs(p_A in A_type)
  return B_tab_type;
```

In our example this means that the employee class has a function for finding all employees in a department:

```
Function find_employees (pDept in department_type)
  Return employee_tab_type;
```

If it is allowed to update the association then there will also be procedures to add and remove associations to B objects like:

```
procedure add_B(p_A in out A_type, p_B in B_type);
procedure remove_B(p_A in out A_type, p_B in B_type);
```

3.5 Aggregation

It is possible to implement aggregations as 1-N associations. However, usually it is desired that the aggregation is closely connected to the parent class, but it is not possible to define a record type with a field that is a PL/SQL table type. Unfortunately the solutions are limited by this fact!

Therefore aggregations are handled like 1-N associations. When passing an object and its aggregation between procedures, we use two parameters.

3.6 References and foreign keys

If object class `some_class` is associated to `my_class`, `some_class` will have a reference to `my_class`. In the database tables the reference will in most cases be a foreign key.

> *The use of foreign keys to object classes in another functional area actually breaks the encapsulation of database structures between the FA's. However, the foreign key concept in relational databases is so strong and beneficial (for data integrity) that we have chosen to make an exception to our rules and accept the consequences.*

In order to make `some_class` able to store a reference to an object of `my_class`, `my_class`'s functional area must supply the following two functions:

```
function get_my_class(p_key in varchar2)
 return my_class_type;


function get_key(p_my_class my_class_type)
  return varchar2;
```

These functions are the inverse of each other. Other FA's may use the `get_my_class` function to create a `my_class_type` identifier record from a key they have stored in their tables. Likewise the function `get_key` may be used to get a key, to store, from a `my_class_type` identifier record. Other functional areas do not use these two functions for any other purposes.

The function `exist_my_class_key` is supplied to check if a certain key exists for an instance of `my_class`. It is implemented as:

```
function exist_my_class_key(p_key in varchar2)
  return boolean;
```

3.7 Implementation details in PL/SQL

The following section contains very PL/SQL specific guidelines that require some working knowledge of PL/SQL. If you do not have such knowledge we recommend that you skip this part and simply take our word for the fact that it is possible to implement the following mechanisms in PL/SQL.

1. Create and manipulate collections of objects
2. Create and delete objects
3. Retrieve and manipulate object attributes
4. Compare objects

3.7.1 Collection of objects

A collection of objects is represented in a PL/SQL table. It is defined as

```
type my_class_tab_type is table of my_class_type;
```

Or as in our example

```
type employee_tab_type is table of employee_type;
```

3.7.2 Collection of object properties

A collection of object properties is represented in a PL/SQL table. It is defined as

```
type my_class_ptab_type is table of my_class_prp_type;
```

Or as in our example

```
type employee_ptab_type is table of employee_prp_type;
```

3.7.3 Creating objects

To create a new object of `my_class` a constructor function is implemented. It looks like:

```
function new_my_class(
  p_my_class_properties in out my_class_prp_type)
  return my_class_type;
```

The constructor function returns the identification of the created object and it also updates the property record.

It may be required to supply information about mandatory associations e.g. a parent object. They are added as extra parameters to the constructor function like

```
function new_my_class(
  p_my_class_properties in out my_class_prp_type,
  p_some_class in some_class_type)
  return my_class_type;
```

In the case of employees they are always assigned to a department so the method would look like this:

```
function new_employeee (
  pEmployee_properties in out employee_prp_type,
  pDepartment          in department_type)
  return employee_type;
```

3.7.4 Deleting objects

To delete an object of `my_class` a destructor procedure is implemented. It looks like:

```
procedure delete_my_class(p_my_class in my_class_type);
```

No example is given here since the employee delete procedure should be self-evident.

Unfortunately, it is impossible to synchronise the delete operation with all the PL/SQL variables referring to the deleted object. The developer must ensure that these variables are not used once the object has been deleted.

3.7.5 Retrieving attributes

The function to retrieve the attributes from `my_class` is implemented as:

```
function get_properties(p_my_class in my_class_type)
  return my_class_prp_type;
```

When only a subset of the attributes is requested a property indicator record is used. An overloaded version of the retrieve function is implemented as:

```
function get_properties(p_my_class in my_class_type,
  p_ind in my_class_ind_type)
  return my_class_prp_type;
```

This function fills in the values of the requested fields in the property record. The key field is always retrieved and therefore it is not possible to specify it in the property indicator record. The developer must ensure that the other fields in the record are *not* used if the record is used to update the object.

For employees the example would look like this:

```
function get_properties(pEmployee  in employee_type,
                        pIndicator in employee_ind_type)
  return employee_prp_type;
```

Additional retrieve functions are implemented as needed. E.g. a function that retrieves a certain attribute, say `my_attr`, is implemented as:

```
function get_my_attr(p_my_class in my_class_type)
  return my_attr_type;
```

In the employee example we often needed just the name of an employee we could add a get_name function to the employee class.

3.7.5.1 Retrieving collections of attributes

Sometimes it is useful to retrieve the attributes of various objects. This is implemented in an overloaded version of the `get_properties` function:

```
function get_properties(p_my_class_tab in my_class_tab_type)
  return my_class_ptab_type;
```

An overloaded version, that uses a property indicator record to specify the subset of requested attributes, is implemented as:

```
function get_properties(p_my_class_tab in my_class_tab_type,
  p_ind in my_class_ind_type)
  return my_class_ptab_type;
```

In the employee example this would become

```
function get_properties(pEmployees in employee_tab_type,
                        pIndicator in employee_ind_type)
return employee_ptab_type;
```

3.7.6 Updating attributes

To update the attributes of an object of `my_class` a general procedure is implemented as:

```
procedure set_properties(
  p_my_class_properties in out my_class_prp_type);
```

When only a subset of the attributes shall be updated, a property indicator record can express it. An overloaded version of the update procedure is implemented as:

```
procedure set_properties(
  p_my_class_properties in out my_class_prp_type,
  p_ind in my_class_ind_type);
```

For employees the example would look like this:

```
procedure set_properties(pEmployees_prp  in employee_prp_type,
                         pIndicator      in employee_ind_type)
  return employee_prp_type;
```

Sometimes you will only update a specific attribute of the object. In that case additional update procedures are implemented. To update the attribute `my_attr` in `my_class` the procedure looks like

```
procedure set_my_attr(p_my_class in my_class_type,
                      p_my_attr in my_attr_type);
```

3.7.6.1 Updating collections of attributes

Sometimes it is useful to update the attributes of various objects. This is implemented in an overloaded version of the `set_properties` function:

```
procedure set_properties(
  p_my_class_ptab in out my_class_ptab_type);
```

An overloaded version, that uses a property indicator record to specify the subset of attributes to update, is implemented as:

```
procedure set_properties(
  p_my_class_ptab in out my_class_ptab_type,
  p_ind in my_class_ind_type);
```

For employees the example would look like this:

```
procedure set_properties(pEmployees_prp  in employee_prp_type,
                         pIndicator      in employee_ind_type)
  return employee_prp_type;
```

3.7.7 Comparing identifier records

A function is implemented in order to compare if two identifier records identify the same object. The function looks like:

```
function is_equal(p_my_class_A in my_class_type,
                  p_my_class_B in my_class_type)
  return boolean;
```

3.8 Summary of encapsulation types and functions

Introducing an object class in a functional area involves implementing various data types and functions. The tables below summarise what is implemented by standard for a certain class `my_class`.

Data type	Description
<my_class>_type	Identifier record for an instance of `my_class`
<my_class>_prp_type	Property record for an instance of `my_class`
<my_class>_ind_type	Property indicator record for `my_class`
<my_class>_tab_type	Collection of objects of `my_class`
<my_class>_ptab_type	Collection of object properties for `my_class`

Function/procedure	Description
new_<my_class>	Constructor for instance of `my_class`
delete_<my_class>	Destructor for instance of `my_class`
get_properties	Retrieve attributes for instance(s) of `my_class`. This function is overloaded to handle both property indicator record and also collection of object attributes.
set_properties	Update attributes for instance(s) of `my_class`. This function is overloaded to handle both property indicator record and also collection of object attributes.
is_equal	Compare if two identifier records points to the same object instance of `my_class`
get_<my_class>	Get identifier record for object instance of `my_class` based on key

get_key	Get key for object instance of `my_class` based on identifier record
exist_<my_class>_key	Check if a key exists for an instance of `my_class`

Additionally, functions are added for each association and aggregation involving `my_class`.

Function/procedure	Description
get_<some_class>	Retrieve associated object instance of `some_class`. This function is implemented when there is at most one associated object.
set_<some_class>	Update association to object instance of `some_class`. This function is implemented when there is at most one associated object.
find_<some_class>s	Retrieve associated object instances of `some_class`. This function is implemented when there are many associated objects.
add_<some_class>	Add an object instance of `some_class` to the associated objects. This function is implemented when there are many associated objects.
remove_<some_class>	Remove an object instance of `some_class` from the associated objects. This function is implemented when there are many associated objects.

4. Experiences

The developers in the project had quite different educational backgrounds, ranging from master degrees in computer science and years of experience to less than a year of formal computer education and little experience in the field of software development.

In the following the experiences are divided into experiences in the design and construction phases of the project. For the time being, we do not have sufficient experience to document the test and implementation phases in this paper.

4.1 Main experiences in the design phase

During the design of the system a number of experiences were gained on the more conceptual aspects of doing systems development in this manner.

4.1.1 The difficulty of abstraction

One of the experiences that we encountered was that, it in general was quite difficult for the developers to think on a more abstract level than they had previously been used to.

Shifting the thinking to be on 'objects' and not on 'records in the database' caused quite some growing pains for a fair percentage of the developers.

Though, it was hard for everybody to think on a more abstract level, the more experienced and those with longer education had it easier than those with less experience. This is not surprising and as the 'unlearning' of old habits began, the problem also began to fade.

However it is an interesting experience that even experienced software developers has some difficulty adjusting to the new principles, and it has strengthened our belief that the conceptual challenge in changing to object oriented development needs to be taken very seriously.

4.1.2 The difficulty of practical encapsulation

One important aspect of the new architecture is the encapsulation of the database from the programs. This caused frustration among the developers because they perceived it as being much more difficult and confusing than the old way. The comment 'Why can't we just select', was mentioned and discussed many times.

For an example of the difference between the new and the ‘old’ way see the example in section 4.1.5.

It is true that the need for different layers seem like a overcomplicating of issues, but in practice this is not the burden one might imagine. For instance it would be obvious to auto-generate the code for the Table API since the code does not contain any functional logic other that the basic database operations.

But again – our experiences shows that when introducing a layered architecture as we have done here, it is important to spend time and effort on explaining its use.

Related to this experience the developers had a tendency to overcomplicate issues. It was very hard to get to the realisation that once you had your model ready; each of the individual actions **was** in fact simple.

4.1.3 Code examples should be present early

One aspect that we (as the project management) had underestimated was the need for small code examples to support the developers. In practice the ability to see things as they will end up is very important and this became clear when a lot of questions disappeared once the developers could see what they would end up with.

Typically developers were very unsure of what to do and how to do it, and in an attempt to make sure they "did it right" they would often spend a long time speculating and complicating the issues. Once we could show them an example the typical comment was - "But is that all there is to it?".

4.1.4 Simplification of operations

As mentioned above the individual action performed in each functional area, did in fact turn out to be quite simple. Of cause the total functionality of the system as such did not simplify, but the description of that functionality on a detailed level did simplify dramatically. This also meant that each FA was a lot easier to understand.

4.1.5 Better quality design documentation

The design documentation was of a much higher quality as well as quantity than before - that is there was more extensive design documentation and the documentation was of a higher quality. Due to better conceptual division as well as each individual operation was easier to describe, it became much easier to read and understand the design descriptions.

Previously it had been common for a design to have to describe rather complex interactions between what were now separate functional areas. Now those descriptions were described in each functional area making the design much more readable.

It is very difficult to give good examples without spending a long time setting up the preconditions and explaining the problem area in detail, but the following simplified example helps illustrate the point. Assume that we are writing a new FA for automatically suggesting salary increases[7]. The algorithm uses as one component a very complex calculation yielding a productivity measurement for each employee and as another component a similar calculation of the department average.

In the 'old' days the design of this new module would describe the algorithm in the main design of the new FA, introducing a lot of concepts that were well known within the Department and Employee FA's but which were not well know by developers that were not domain experts. This would make the design rather hard to read.

The 'new' way would describe the algorithms in the context of the FA's to which they belonged – the Employee FA would be expanded with a new service as would the Department FA. People with domain knowledge could then review each of these changes.

The new design could then be described quite simply as
"For all departments do the following.
1. Calculate the Department average productivity (Department FA, service GetProductivity)
2. For each employee in the department
3. If the employee productivity (Employee FA, service GetProductivity) is higher than the average then bump their salary by 50 %)
4. Otherwise keep the present salary"

The improved design documentation also meant that the review of the design became much less of a strain for the reviewers because the discussions could be on a higher level where content instead of form could be discussed.

[7] Note that there cannot be a decrease – we hope ☺

4.2 Main experiences in the construction phase

Following the design phase we entered the construction phase where another set of experiences were collected.

4.2.1 Stabile design foundation

Once construction began, we naturally had situations where it turned out that there were areas where the detailed design did not solve the problems encountered, or where certain assumptions did not hold. This of cause meant that the design had to be updated, but this turned out to be a surprisingly minor problem.

The model part of the designs was practically always stable and the changes were mostly related to the way a certain service should work internally - or to the fact that an extra service was needed because of demands from other FAs.

4.2.1.1 Encapsulation benefits development in practice

When it became apparent that the implementation of a FA was indeed independent it also became apparent that this is a very beneficial quality in practice. Whereas changes in the implementation in the past have 'rippled' through large parts of the system code, the changes were now (in practice as well as in theory) confined to the FA owning that part of the model.

4.2.2 Performance

When we initially started planning for this method there was some concern that performance would be an issue. This is a traditional worry among those of us used to relation RDBMS's when we look at the way things are done in the object oriented world, but as it is seen for the following this has not been the case.

Interestingly enough many experienced with object oriented development have long claimed that performance was not an issue, but we have none the less kept our scepticism until now.

4.2.2.1 PL/SQL tables are memory hungry

The implementation of collections in our method - the PL/SQL table - turned out to be very efficient in terms of CPU performance, but unfortunately it was not as efficient in terms of memory usage. This meant that the size of collections had to be kept smaller than we had originally envisioned, but in practice this was manageable.

This problem probably occurred in the first place because the developers are used to passing large amounts of data from one part of the system to another by saving temporary data to database tables database tables and using SQL to retrieve it whenever needed. This automatically leads to the need for passing data the same way by using PL/SQL tables.

4.2.2.2 No problem in loosing 'The power of SQL'

One of the great worries we had in the beginning, was the possibility for the encapsulation causing us to loose performance. Would the performance be good enough when selects across FA borders were no longer allowed? To many the ability to do joins

effectively is one of the great benefits of relational databases, and we were denying ourselves that possibility in some cases[8].

This worry was groundless so far - in fact we have seen performance improvements in some cases where we were initially worried that there would be decreases.

At first we were surprised, but on the other hand we feel that this merely reflects the fact that it is very important to design the model in the correct way. It seems that when the model is correct the resulting code will be quite simple and fast.

One aspect of the method that might actually improve performance is that since only a very limited part of the code actually contains SQL against any given table it is much easier to ensure that the SQL is reused to the maximal extend possible.

This makes it a lot easier to write code that the RDBMS keeps cached and therefore need only parse one. A high performance RDBMS (such as Oracle8) will therefore gain some performance by not having to parse SQL statements as often.

4.2.3 Higher quality code

The code being produced is of higher quality, partly because it is much more model near, and partly because there is a much higher degree of readability.

This is of course somewhat of a postulate but the following simple example serves as an indication of what we mean.

Reusing the example from section 4.1.5 (a salary correction FA) the code would typically look something like this in our ‘old’ way of doing things:

```
declare
  cursor getEmployees (lDepartmentID number) is
    select employeeId
     from employees
    where deptID = lDepartmentId
  cursor getDepartments is
    select DeptID
      from departments;
begin
for DeptRec in getDepartments loop
  for EmpRec in getEmployees (DeptRec.DeptID) loop
  -- insert complex calculation of productivity here
  end loop;
end loop;
end;
```

Using the principles in this paper the code would now look like this:

[8] Remember from earlier that this was the main reason behind allowing for the view interfaces between FAs.

```
declare
  Employees employee_tab_type;
  Departments dapartment_tab_type;
begin
  Departments := get_departments();
  for lDepartment in Departments loop
    Employees := find_employees(lDepartment);
    for lEmployee in Employees loop
      if GetProductivity(lEmployee) > GetProductivity(lEmployee) then
       -- do whatever
      end if;
    end loop;
  end loop;
end;
```

Subjectively it can always be argued which of the code examples that is the prettiest – we feel that the new way of doing things makes for a better looking as well as more readable code, but that is of cause open for debate.

However it is not open for debate that there is a number of objective advantages to the new way of doing things:

1. In the first example the new FA (or module) uses explicit knowledge of the employee and department implementation – the names of tables, the name and type of the primary keys etc. The second example does not assume anything about the implementation, but only that certain services exist. This is an objective fact.
2. This means that the implementation of those areas has now been locked, and there is no way of reorganising the areas without rewriting the salary module too. The second example does not lock the implementation of the employee and department FAs – as long as the services are still supported, the department information could for instance be placed in a whole different database, or even in flat files. This also an objective fact.
3. It is our claim that the second example is much easier to read, and that a developer without prior knowledge of the area could understand what is going on much quicker than in the second example.

It is important to note that none of the above is especially difficult to do, but that the method we have devised will **force** the developer to structure the code in such a way as to make these advantages appear. It will be very easy for a code reviewer to locate any violations of the guidelines, and equally clear to the developer that this needs to be corrected.

4.3 Miscellaneous experiences

Because of the component-based architecture, the developers on each FA did in fact become domain experts on the area that they were in charge of. Each participant had

time to learn the FA functionality in detail, and the practice of ‘outsourcing’ functionality to other FAs (as the GetProductivity method in section 4.2.3) meant that the details of other FAs were well shielded.

The developers felt that the new architecture produced a much better overview of the construction within each FA. Again this is caused by the encapsulation shielding unnecessary details from each developer.

We found that the detailed component-based architecture we used made the object oriented concept of roles very important. We often had FAs that owned the general definition of a certain concept and other FAs that expanded the concepts in a role like (as opposed to specialising in an inheritance way) manner. However we have not yet fully understood the implications of this and we therefore will not make more of an issue out of it in this paper.

Another comment - although anecdotal in nature - is that is quite interesting that at least one developer regarded the new architecture as more 'fun'. However this is of course purely anecdotal and perhaps others would find the method cumbersome and boring.

5. Conclusion

Looking at the entire process – which conclusions can be drawn from our experiences?

Our overall experience is that these ways of approaching object oriented development is a good first step. We believe that if we had tried to go all the way, that is to change the analysis, design and implementation platform all at once, in using object orientation we would not have succeeded (and this is also the experience of others whom we have discussed the matter with). Using this approach we have realised a number of advantages.

- We have been able to focus on the conceptual challenge at first. This means that the uncertainty related to using new development tools and technologies has been minimised and thereby the total risks for the project have been substantially reduced.
- Because the underlying technology has been known to all involved we have avoided a period of time where no project progress was occurring.
- The developers have been able to draw upon previous experience to a greater extend.
- Due to the architecture we see a clear upgrade path - moving the system towards a multi-tier architecture seems like a quite feasible possibility, and incorporating new technology such as object oriented languages and frameworks such as Enterprise Java Beans will to a large extend be 'only' a question of technology.
- It is important to note that the 'rules of the game' change slightly because the time spent in the design phase is longer compared to more traditional methods. This was certainly the case for us and we believe that this is inherent to the object oriented approach. However the resulting quality and the level of detail in the design phase products seem to make up for the extra time used. We also feel that a drop in time needed in the construction phase further justifies the extra time spent, but whether the effects cancel each other out is not certain.

- One mistake we made was not to insist more on documenting the object model as early as possible. We feel that if we had done so, we would have had less frustration among the developers, but we mistakenly believed that the best way was to try to 'sneak' these concepts into the development.

So overall we feel that the experience has been a positive one, but there are (of cause) some conditions attached to such a statement - the most important one being that the project depended heavily on the fact that there were a large number of experienced PL/SQL developers.

Another important thing to note is the fact that the problem area was very well known. This had the effect that the division into FAs was done correctly the first time around. There is no doubt that without a correct division into FAs the project would have experienced severe difficulties.

However it is interesting to note that the concepts presented in this paper has been the inspiration for a slightly simplified version of the method used in another project. The method once again proved it usefulness by allowing a fairly large number of people to work in parallel in the design and implementation of a system. Although the project was under great time constraints the parts using the object oriented methods described in this paper made it possible to meet the (fairly optimistic) time schedule with respect to the design and implementation parts of the project.

So the experiences in presented in this paper does not seem like a ‘lucky punch’ but seem to be repeatable in other projects.

It is not possible for us to stipulate whether we end up making a given system for less money than with traditional methods, but we feel that in the total lifecycle of the system there will be a gain. For the first release we do not think there will be a direct financial gain by employing the techniques in this paper, but we do feel it results in a system of a better quality.

In the final analysis we feel that the stepwise approach towards object oriented development has been a success - but in the future we will try to utilise object orientation in all aspects of the development.

6. References

[1] Center for Object Technology, www.cit.dk/COT.

[2] Johnny Olsson, Allan R. Lassen. *Experiences from Object-relational Programming in Oracle8.* COT/4-06.