ORACLE White Paper

Designer/2000
C++ Object Layer Generator

Product Overview

VERSION 1.0: DECEMBER, 1995

Part C

Part C

December 1995

Author: Dai Clegg

Contributors:

Table of Contents

Management Summary

The Need for Object & Relational Co-existence

Object oriented (OO) programming languages, such as C++, are being chosen for many development projects. They have expanded their traditional stronghold in embedded systems via telecommunications and financial applications into a broad range of commercial applications. For example:

- for middleware,
- where a purely diagrammatic user interface is needed,
- where existing C++ components are to be integrated,

or simply as a way for an organization to broaden its capabilities and acquire experience in object technologies (OT). Programmers using an OO language such as C++ with Oracle7, or any relational database, face the task of writing code to store objects in, and retrieve objects from the database. This is time-consuming and error-prone, a fact that results from the so-called 'impedance mismatch' between set-based relational processing and object-based processing. One response is to use an OO database (OODBMS). In many cases this is impractical since the very reason that the programmer needs to store or retrieve data is to share it with other applications, frequently relational-based enterprise applications.

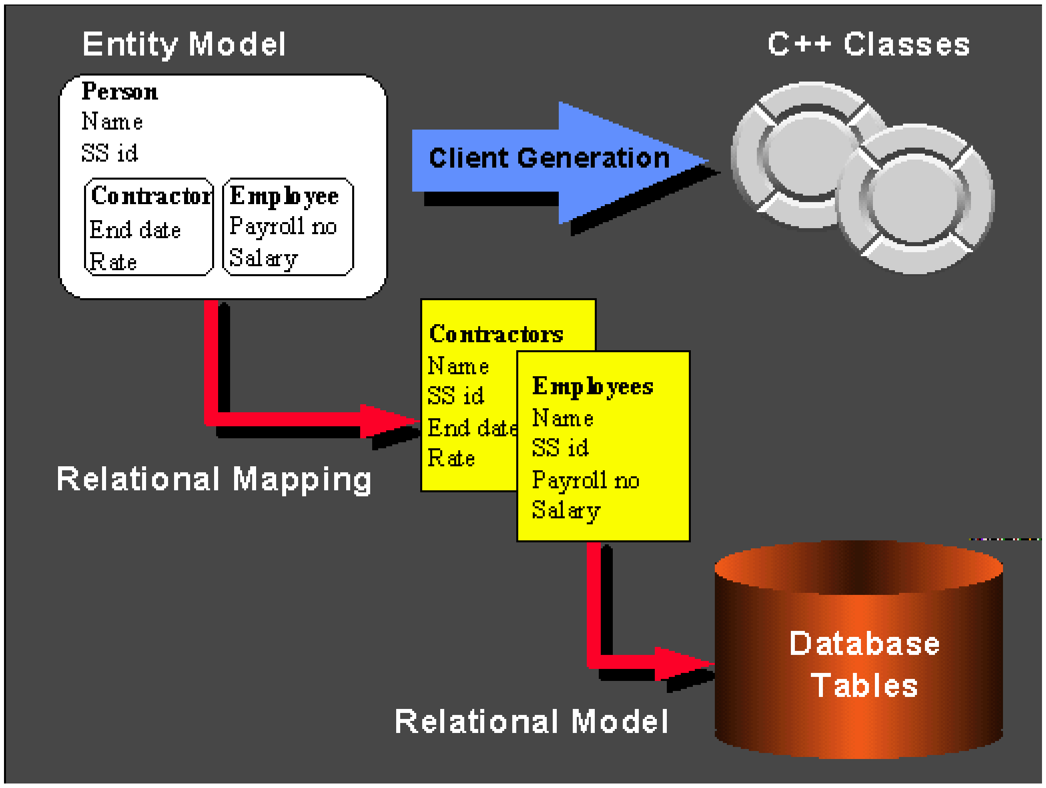
The need for object oriented programming to co-exist with relational databases is real today and growing, although it will eventually diminish as successful relational databases extend their capability to manage objects directly (for example Oracle8). The C++ Object Layer Generator exists now for C++ programmers who wish to access Oracle7 databases.

Overcoming the Impedance Mismatch

Overcoming the impedance mismatch by mapping from one world-view to the other is not impossible. In many cases it is relatively trivial (for example a Customer object might correspond to a row in the Customers table), but it is tedious and needs to be maintained as the data structure evolves (for example new columns are added to the Customer table). Now C++ developers can use Designer/2000 graphical tools to describe their class structures or type model (the object structure) and how it should map onto the physical database (the relational structure). They use the Server Generator to generate the database definition and the C++ Object Layer Generator to create the client-side representation of those objects.

Figure 1, opposite, illustrates how the C++ Object Layer Generator uses Designer/2000 to create usable objects for C++ programmers. The Entity Relationship (ER) Diagrammer represents the class structure, using sub-types, and nested sub-types to represent the hierarchies of classes normally found in C++ programs. The Data Diagrammer represents the relational schema as implemented. The Data Design Wizard creates a relational schema from an ER diagram according to the developer's preferred mapping. The Server Generator implements the relational schema as a database. The reverse engineering facilities in Designer/2000 create a data diagram and a default ER diagram from an already implemented database. In either case the result is a diagrammatic mapping from object structure to relational data structure. Just as the Server Generator will create the database, the C++ Object Layer Generator creates C++ class definitions for all the object types or classes defined in the ER diagram.

Figure 1: Generating Classes in the Client and Tables on the Server



The C++ Object Layer Generator

The C++ Object Layer Generator is different to the other Generators in Designer/2000. The other generators create either server-side data and procedural definitions (the Server Generator) or client-side program code (the code generators for such targets as Oracle Forms, Visual Basic$^{(tm)}$, etc.). The C++ Object Layer Generator creates and manages **components**, used by C++ programmers. It generates

- class definitions for persistent types,
- public methods to manage storage in and retrieval from Oracle7 for these types
- private methods to manage client-server interaction efficiently.

The Value Proposition

Many commercial enterprises and government agencies run Oracle7 databases that are vital for their effective operations. Increasingly those organizations want to use tools such as C++ to access that data. The Designer/2000 C++ Object Layer Generator makes that cost effective.

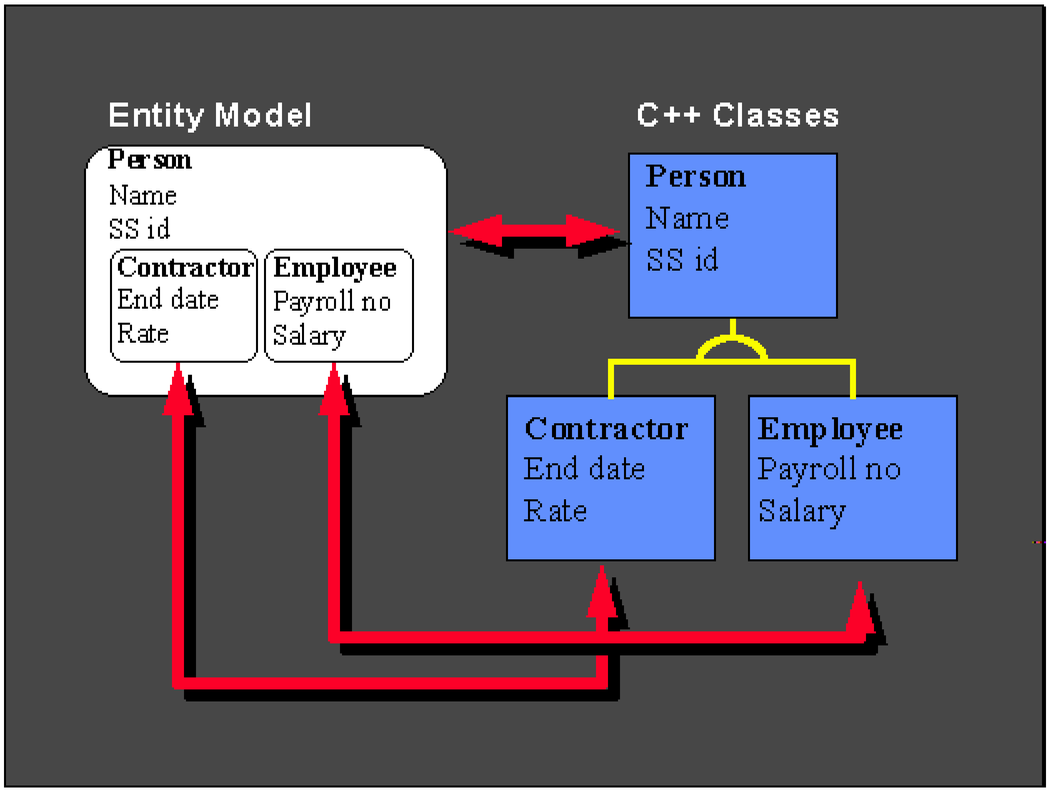
Many developers with an otherwise strong commitment to object technology are concerned that OODBMSs are limited in the scalability, interoperability and reliability necessary for mainstream applications. The C++ Object Layer Generator makes Oracle7, which has no such limitations, a viable store for those developers.

Objects, Entities, Classes and Tables

Classes are the building blocks of OO programming. The C++ Object Layer Generator needs to know what classes it is to generate and how these map onto the relational schema. Designer/2000 provides the tools to define these inputs. The classes are defined by using the Entity Relationship Diagrammer. An entity definition can be regarded as a sub-set of a full object definition. Entity Relationship modelling does not provide facilities for capturing behaviour and lifecycle, in traditional modelling these are not as well integrated as in Object modelling. However, Entity Relationship modelling **does** capture data, structure and association.

The C++ Object Layer Generator generates code to manage the persistency of objects, that is to say it is concerned with storing and retrieving the data and data structures. To do this, data and structure are precisely what it needs. So although Entity Relationship modelling is not as rich as Object modelling, it is rich enough to drive the C++ Object Layer Generator. Figure 2, below, shows how entities and entity sub-types correspond to classes, or types.

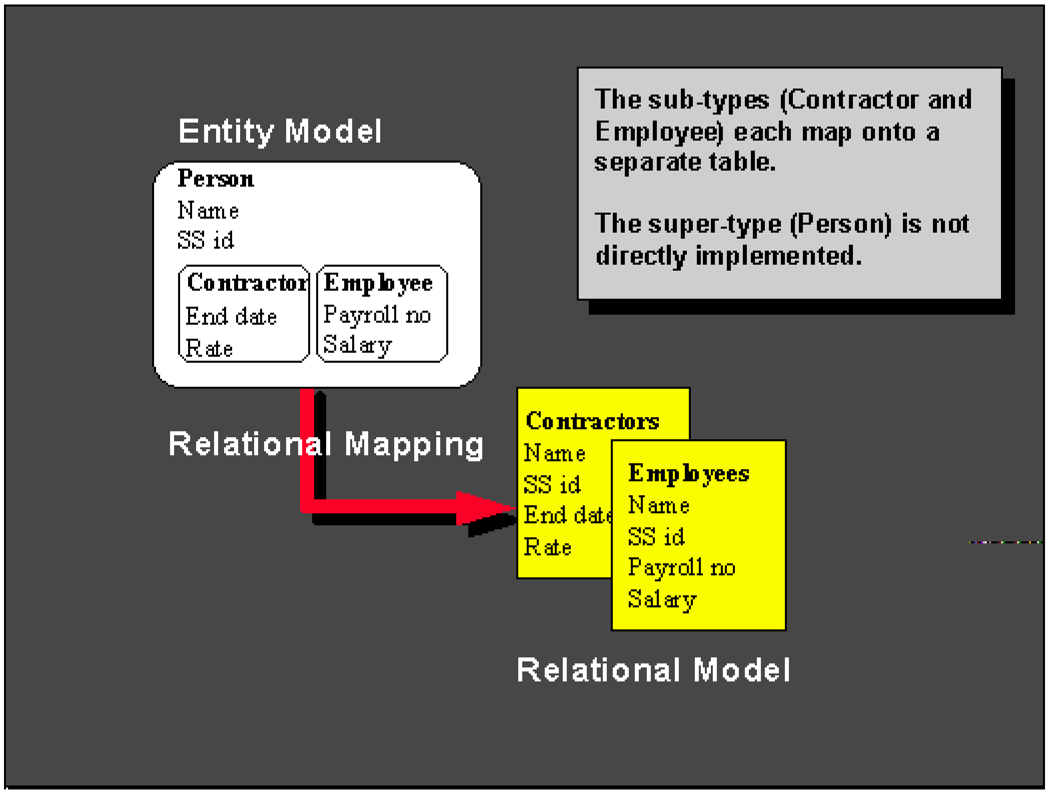
Figure 2: Identifying Entities with Classes



Designer/2000 hence provides basic, but adequate, object structure modelling. It also provides, in the Data Diagrammer, a much more comprehensive modelling environment for relational structures. In addition, the Designer/2000 repository maintains the mapping from entities to relational implementation. Unlike the correspondence of classes to entities, which is one-to-one, the mapping from entities onto the relational structures that implement them need not be one-to-one. This is vital because the end-user wishes to see the data in terms of her business perspective and the database designer wants to implement it in the most efficient way available. By supporting alternative implementations of the entity model, Designer/2000 is allowing the business view of the world to be different from the implemented database schema. So the C++ programmer constructs an effective user interface, using classes based on the end-user view point, and the hard work of mapping this onto the implemented database schema is finessed by the C++ Object Layer Generator.

Figure 3, overleaf, illustrates an example mapping.

Figure 3: An Example Implementation Mapping



Even for the simple example in

Figure 3 there are alternative implementations:

- A single table for all super- and sub-types.
- A separate table for each super- and sub-type.
- One table for the super-type and one each for the sub-types.

Depending on the mix of applications, each might be the best from the database designer's viewpoint, but for each of the alternatives the user has the same viewpoint.

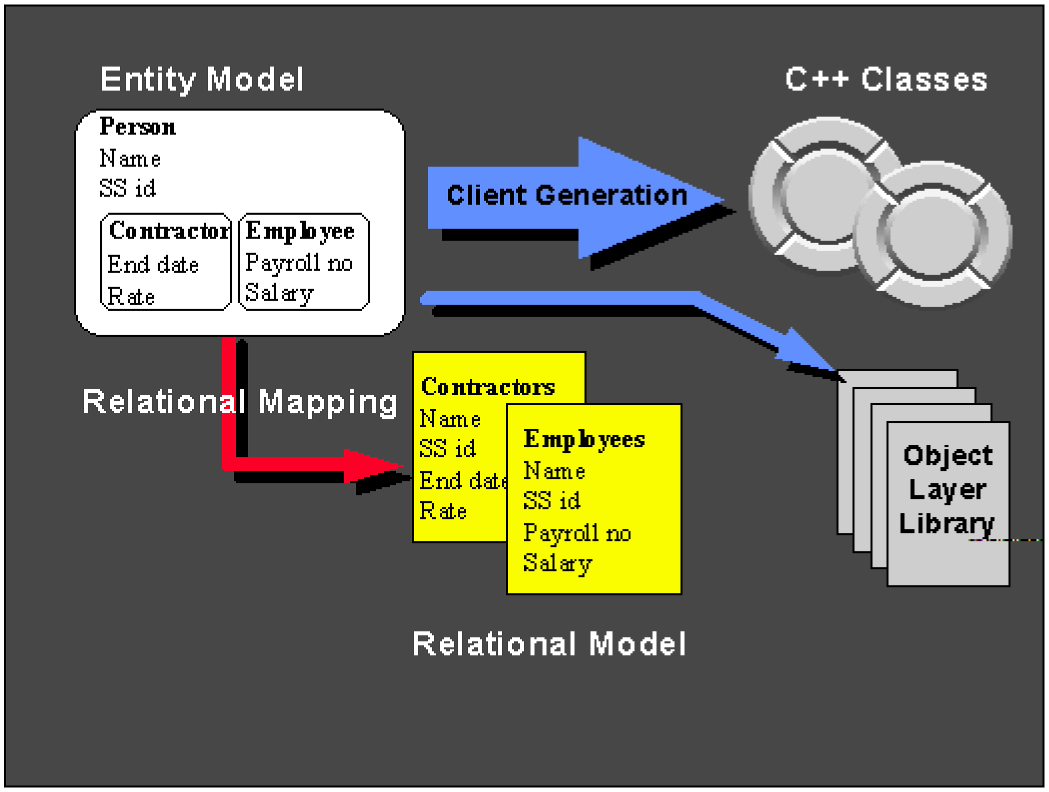
Transparent Client/Server Processing

Class definitions are generated from entity definitions. These classes have public methods for storage and retrieval operations. There is also library code that implements the mapping from classes to tables and optimizes the client/server transfers.

For example a program may create a new *Employee*. This is not written to the database unless explicitly requested. The program then assigns the *Employee* to a *Project*. If the database row representing that specific *Project* is already in the client-side cache, that is if it has already been retrieved for some other purpose, the association is made in the cache, but still no network or database overhead is incurred. Finally another *Employee* is allocated to the *Project*. This *Employee* is not in the cache, so a network transfer is required and all the buffered operations outstanding are effected in a single message pair.

From the view point of the C++ programmer this was a series of operations on a number of objects. Whether or not any of the objects had to be fetched from the server in order to complete the operation is not apparent. At what point updates are transferred to the server is irrelevant. The library code manages all this transparently. The generation of this library is dependent on the Entity-to-Table mappings, associations between entities (relationships) and the corresponding associations between the implemented tables (foreign keys). A more detailed view of the **client-side** generation process is illustrated in Figure 4, below.

Figure 4: Generating the Object Layer Library



The Development Process

Pre-requisite Definitions and Delivered Components

The Designer/2000 C++ Object Layer Generator allows C++ programmers to use Oracle7 as a persistent store. It also provides a transparent means of using that store. In order to exploit this capability, there are three definitions that must be created:

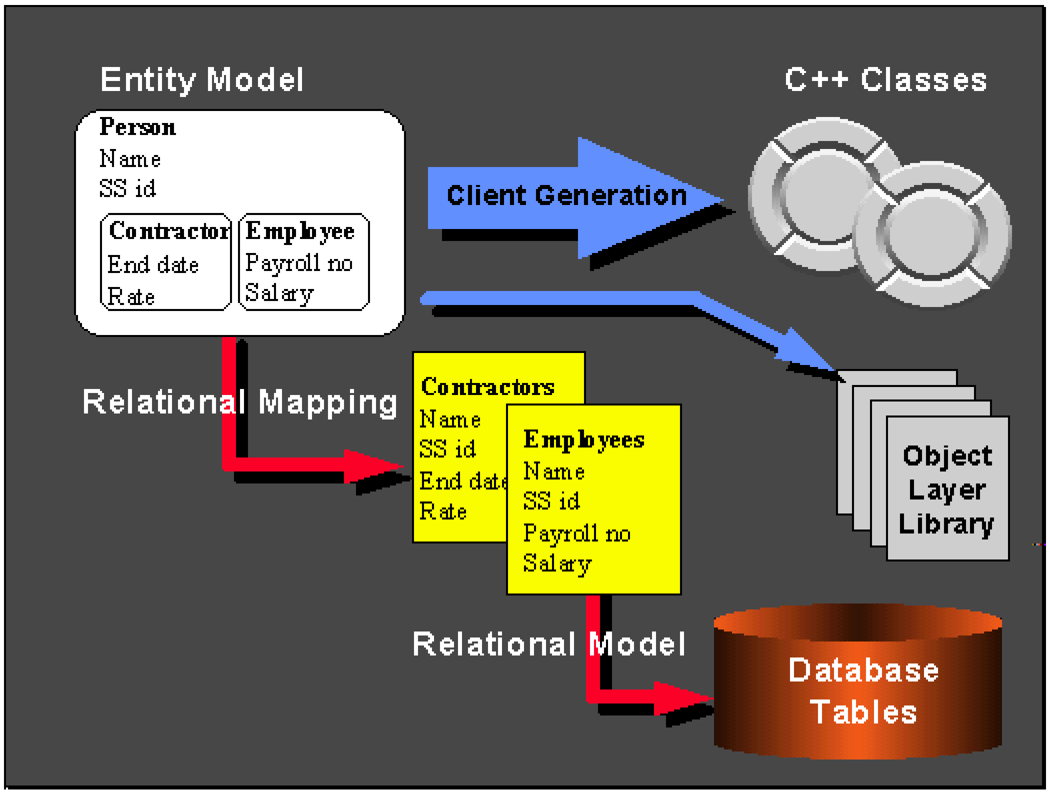
- an entity relationship model to represent the persistent classes,
- a relational schema to represent the persistent store,
- a mapping from one to the other.

From these definitions Designer/2000's Server Generator and C++ Object Layer Generator create the client and server components:

- C++ native classes with full function public methods for persistency,
- C++ object layer library code to process client/server efficiently and transparently,
- Oracle7 database definitions to store the object instances.

Figure 5 below, shows the definitions, transformations and delivered components.

Figure 5: The Full Picture



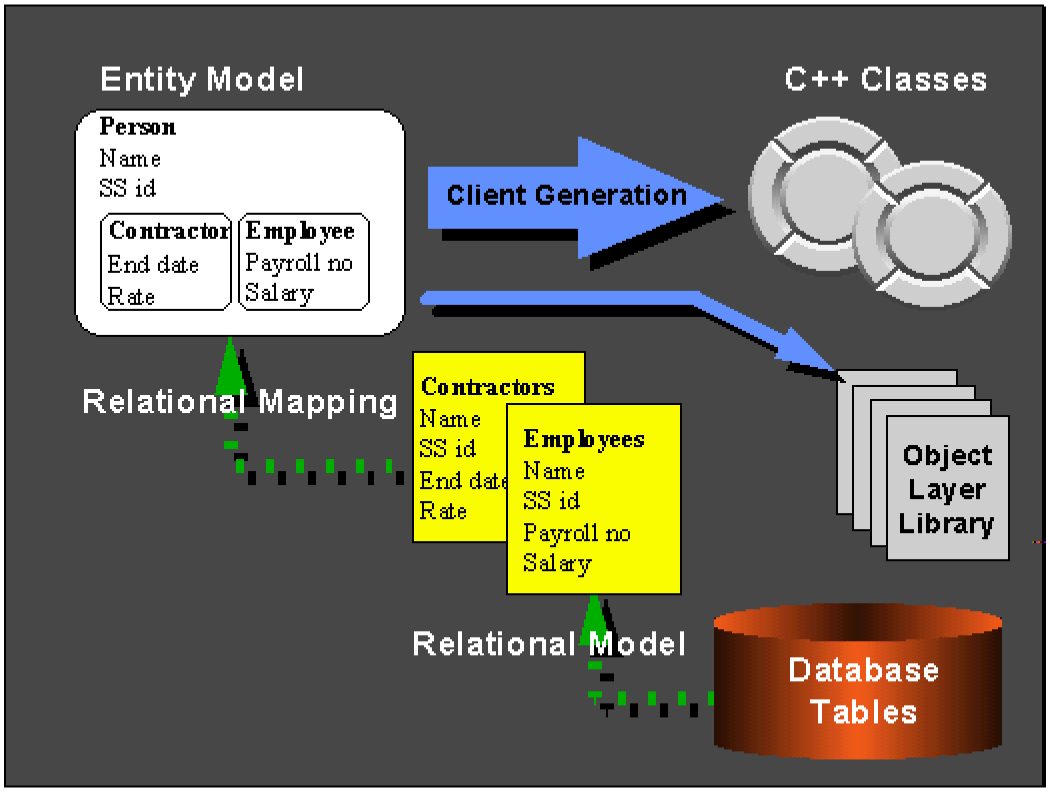
Development Tasks and Dependencies

There are inter-dependencies indicated by Figure 5. For example we need all the definitions before we can generate the client-side components. These dependencies imply a sequence of tasks, but in reality there is more than one way to arrive at the required definitions.

The Server Generator's reverse engineering capabilities create a relational schema model from an already implemented Oracle7 database. The retrofit capability of the Data Design Wizard creates an entity model from a relational schema definition. By using these facilities, developers can create the definitions necessary in reverse order. Clearly, the entity model created in this way represents only a simplistic mapping from entity (class) to table, but developers can refine to the mapping to match the user model more accurately.

The design recovery facilities mean that Designer/2000 and the C++ Object Layer Generator can support developers adding new C++ applications to an existing database as well as supporting development of application and persistent store in parallel. Figure 6 below, shows this alternative development process.

Figure 6: Design Recovery



Evolving the Schema and Object Model

Creating and synchronising the C++ object model and the Oracle7 persistent model is necessary to initiate development. However, in all non-trivial development projects the models will change. Users will require:

- changes to match their world-view more accurately,
- new data to be stored and new formats for data already defined.

Developers will identify

- improvements to tune the database design,
- more efficient ways to deliver the features that users demand.

All these can cause changes to the object model and the underlying relational schema. Typically by the time the changes have been identified the generated class definitions will have been customised to add user-interface behaviour and other processing. It would not be acceptable to lose these modifications, so the C++ Object Layer Generator can recreate its own code without altering custom code.

There will be such changes where custom code is affected, for example if the definition of the US numeric Zip Code in a postal address is broadened to accommodate European alphanumeric postal codes. Manual changes will be necessary in any client-side methods that validate, display and edit postal codes. However the persistency of the new definition is managed automatically by the C++ Object Layer Generator and the Server Generator, which generates changes to existing databases.

Full Life-Cycle Support

Projects starting in a ‘green-field’ situation; projects adding to existing applications; projects maintaining existing code. All these can benefit from the Designer/2000 C++ Object Layer Generator.

Using the Generated Components

The C++ Object Layer Generator Transparency Layer

There are many interfaces available to a C++ programmer to access an Oracle7 database. These include:

- Pro*C,
- Pro*C++,
- OCI,
- ODBC,
- Oracle Objects for OLE,
- Microsoft DAO

These are programmatic interfaces. Programmatic interfaces simplify the task of accessing a set based SQL database, such as Oracle7, from a procedural language, such as C++. They achieve this by hiding or bypassing the need to handle such complexities as bind variables and cursors directly.
The transparency layer of the C++ Object Layer Generator has a higher level of ambition. For specific operations, it seeks to make the presence of the database **transparent** to the developer. One way to achieve this would be to cache all objects on the client at runtime. This would be impossible in memory requirements and transfer times, so the transparency layer manages the same effect by maintaining a client-side cache. Objects are retrieved into the cache only as necessary. Cache objects are updated - insertion, deletion, properties and associations - corresponding to program operations. Some operations that the transparency layer covers are:

- Creating objects,
- Deleting objects,
- Updating object properties (e.g. DepartmentName),
- Updating object associations (e.g. The Department that the Employee works in),
- Traversing associations (e.g. through the set of Employees in a Department).

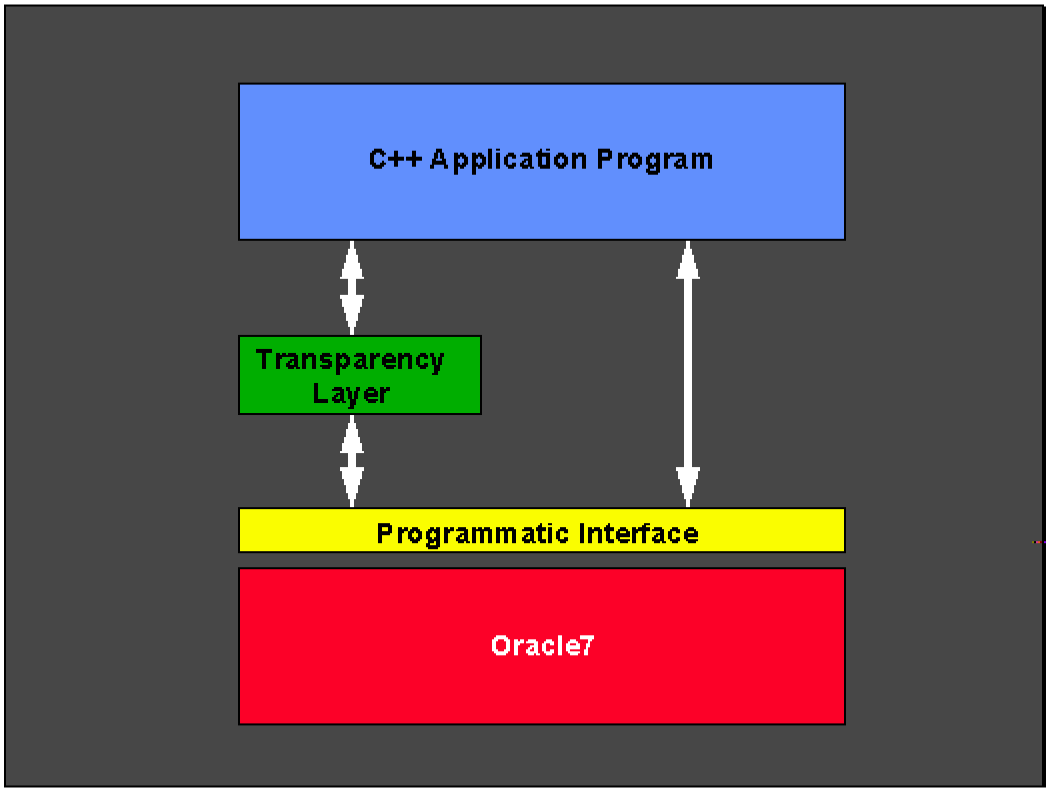
Other functionality is provided as part of the interface, functionality that **does** recognise the database. This includes:

- Database connection and disconnection
- Server-based query of objects based on some criteria (e.g. Age > 30)
- Explicit object locking
- Transaction Control

The transparency layer in the C++ Object Layer Generator does not completely replace a programmatic interface. Some database operations cannot be transparent. These are typically the ones that affect the database itself, for example data definition language operations such as create new table. So an application will normally use both the C++ Object Layer Generator and a programmatic

interface. The transparency layer itself uses a programmatic interface to access Oracle7, a C++ encapsulation of OCI, and this interface is available to application programmers.
Figure 7, below, shows the architecture of a typical application program:

Figure 7: Application Program Architecture

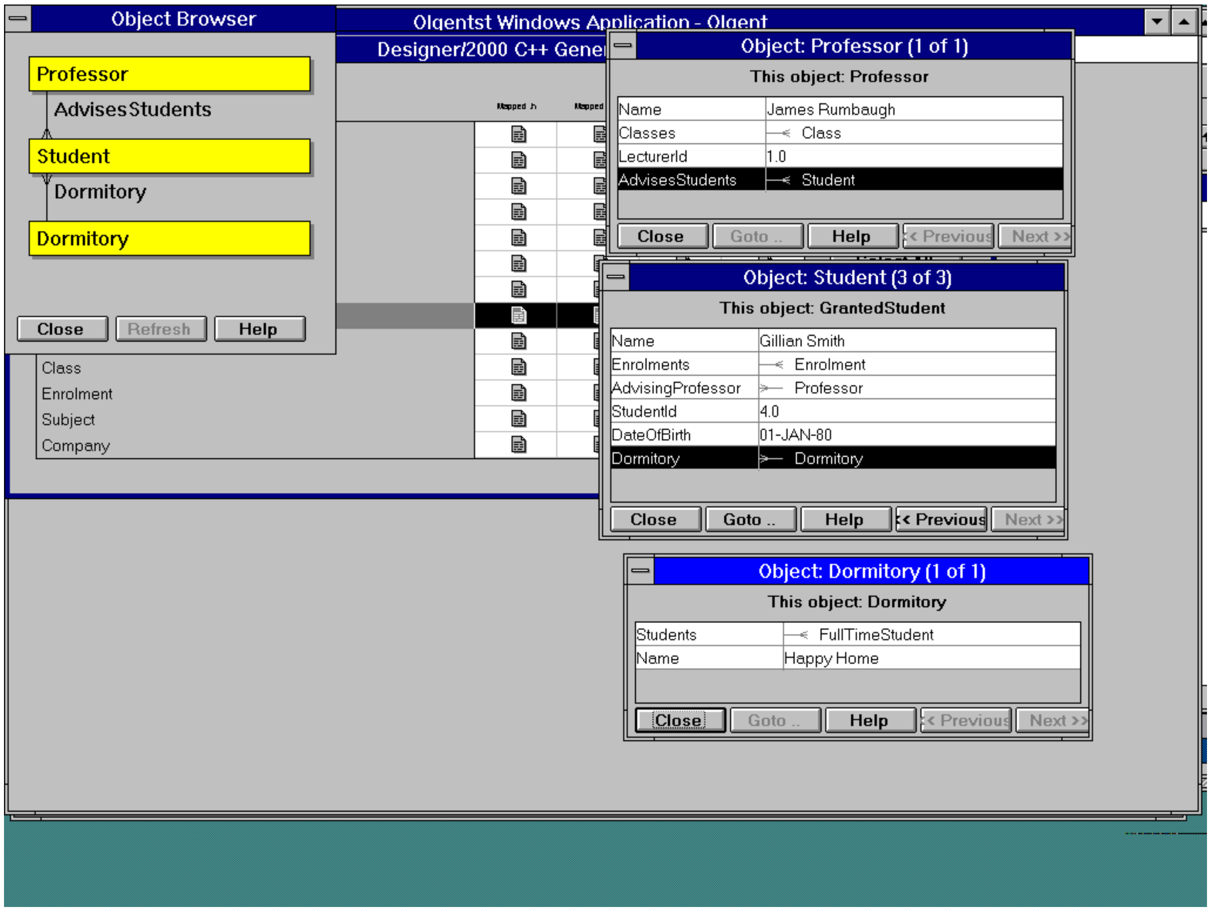


Browsing the Stored Instances

The mapping from generated C++ classes to implemented tables may not be isomorphic, indeed it may be very complex. Standard browsing tools (for example Discoverer/2000 or SQL Plus) query the database as implemented, so the data retrieved by such browsers may not be meaningful to a C++ programmer. For example in Figure 2 on page 4, all the classes might have been implemented in a single table. The programmer trying to check that the property values of a newly created instance of *Contractor* will be unable to find any database object corresponding to *Contractor*. To resolve this difficulty the C++ Object Layer Generator includes an Instance Browser that retrieves from the relational tables, but presents classes to the user.

Figure 8 below, shows the user interface to the Instance Browser.

Figure 8: The Instance Browser



This object: Professor

Name	James Rumbaugh
Classes	Class
LecturerId	1.0
AdvisesStudents	Student

This object: GrantedStudent

Name	Gillian Smith
Enrolments	Enrolment
AdvisingProfessor	Professor
StudentId	4.0
DateOfBirth	01-JAN-80
Dormitory	Dormitory

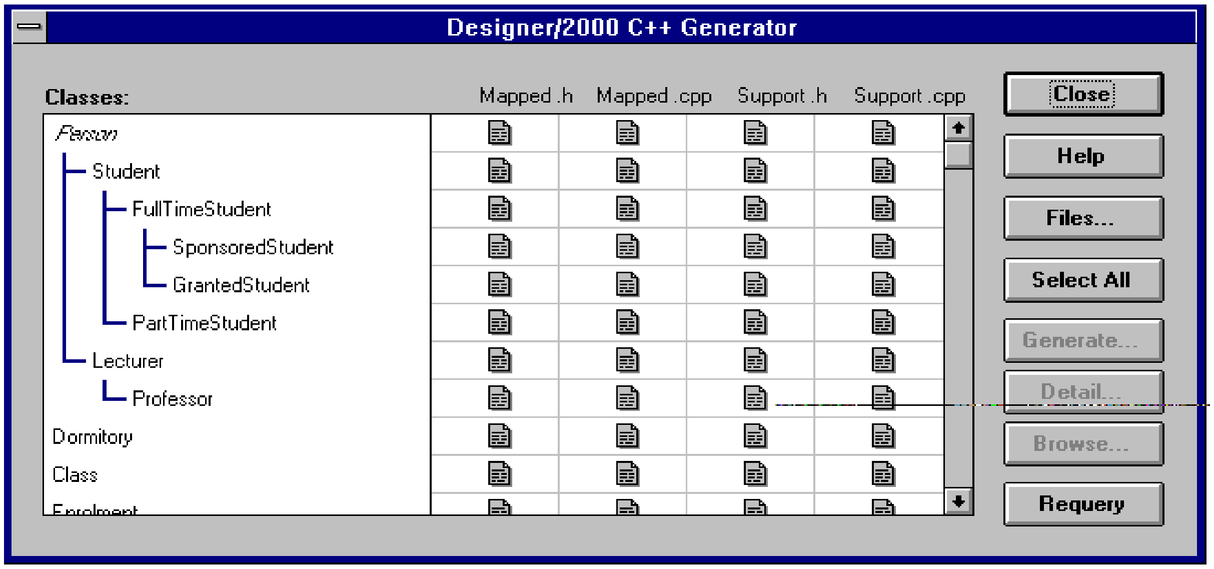
This object: Dormitory

Students	FullTimeStudent
Name	Happy Home

Controlling the Generation Process

The C++ Object Layer Generator operates on fragments of code, using template files as its source and inserting and replacing code in appropriate source files. In a major project, such as is most likely to use the C++ Object Layer Generator, the management of source files and adherence to house style for code is of great importance. For this reason, at generate time, the user can specify which types of code are to be inserted into which files. The C++ Object Layer Generator keeps track of what has been generated and into which files. However, this may be overridden by the user, and source files can be browsed as part of the generation process to ensure that the appropriate actions are being taken, or have been taken.Figure 9, below, shows the interface the developer uses to partition source code amongst source files, on a class by class basis, if required.

Figure 9: Directing the Generation Process



Conclusions

The Designer/2000 C++ Object Layer Generator can be useful to any C++ programming project that needs a persistent store for objects. Because the persistent store is an Oracle7 database, C++ programmers have access to the industry leading database, giving them scalability and security, and data sharing with all the other applications implemented on any connected database:

1. **C++ developers using Designer/2000 for business modelling and implementation** already have, in the form of their entity relationship model and data model, a basic type model and relational schema mapping.
2. **C++ developers who wish to access an existing Oracle7 database,** can reverse engineer the database schema and default type model into the Designer/2000 repository.
3. **C++ developers who need a reliable persistent store**, can use Designer/2000 to define types and their relational mapping before generating the database definition.

They can each then generate native C++ classes that will have all the necessary persistency methods, optimised for client/server operation. The regenerate capability means that even though developers will add their own methods (for example for user interface, for client-side integrity checking etc.), when the schema or object models change, the generator will ripple these changes through to the class definitions without jeopardising user custom code. Control remains with the developer but the generator takes on much of the hard work.