# OBJECT-ORIENTED DATABASE MANAGEMENT SYSTEM CASE STUDY FOR DECLARATIVE QUERY LANGUAGE

GEORGES AYOUB

A MAJOR REPORT

IN

THE DEPARTMENT

OF

COMPUTER SCIENCE

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE

CONCORDIA UNIVERSITY

MONTREAL, QUEBEC, CANADA

August 1998

0-612-39109-4

## Abstract

OBJECT-ORIENTED DATABASE MANAGEMENT SYSTEM
CASE STUDY FOR
DECLARATIVE QUERY LANGUAGE

Georges Ayoub

Object-Oriented database management systems (OODBMS) combine the abstraction power of objects with the query and performance capabilities of database management systems. Existing query notations were missing many object related features until recently. The introduction of a new query notation, known by *Object Comprehensions*, allows queries to be expressed clearly and processed efficiently. Our work is to establish a testbed for the processing of Object Comprehension Language (OCL) queries using an experimented object-oriented database, Ode. This thesis overviews object-oriented databases evolution, and object query processing, then introduces an object-oriented database system, Ode, whose programming language O++ is based on C++. A university data model is built using O++, stored into the Ode database, and *utilities*, such as bag, list and set, are written to support the processing of OCL queries. The translation of OCL queries into O++ is not part of this thesis, but is part of a related project.

## Acknowledgements

I would like to thank my supervisor, Dr. Gregory Butler, for his patience over the last three years and for his guidance and encouragement to complete this project.

I would like to present this piece of work to the memory of my parents, also to my beloved brother, whose presence is always in mind.

# Contents

## List of Figures

# Chapter 1

## 1 Introduction

Twenty years ago, relational database systems started playing a dominant role in the business and academic sectors, but as software systems become larger and even more complex relational databases proved not to be adequate for the job. Thus, developers have been searching for mechanisms to control that complexity, while maintaining the main goals lowering the cost and improving productivity.

A number of programming paradigms emerged as solutions to the complexity problem. but all indicators point to object orientation as the most promising solution. The object-oriented paradigm started to gain popularity during the past few years. Many see it as an opportunity to rethink programming from an entirely fresh perspective [7].

As object-oriented technology started to gain more acceptance, groups such as ODMG (Object Data Management Group) have made progress towards standardizing data models. The goal is to provide a common architectural setting for object-oriented applications.

Effective adoption of standards, enables the portability of customer software across different products, and encourages software reusability. This in turn will reduce the complexity and lower the costs and improve productivity, as opposite to relational databases where the cost is considered as a disadvantage.

Even though object-oriented databases are a growing field, they are still not widely accepted because most of the market areas are based on relational databases which are widely used due to many facilities the databases provide, such as the query management facility.

The database user retrieves data by formulating a query. The formulation of the query is provided under two options:

- Browsing query used in PC market databases: a user-friendly interface is displayed, allowing the user to select table names and the corresponding fields. Once selected, the user can set up values for some fields needed to be included into the conditions.

  When the query gets executed, the result is displayed in tabulated form.
- Declarative query used in most mini and mainframes systems: users build their own query using the data manipulation language provided with the database.

In both the above formulations, the query processor is used to interpret the online user's query and convert it into an efficient series of operations capable of being sent to the data manager for execution.

After having large satisfaction from using relational queries, developers want to provide queries for object-oriented databases, but inadequacies still exist and are categorized [1] into four groups:

- *Support of object-orientation*: A few object-oriented query languages do not capture the class hierarchy defined by the ISA relationship between classes of the database schema.

- *Structuring power*: refers to the ability to explore and synthesize complex objects which are the components of object-oriented databases. The creation of a new object may require a collection of objects as a parameter. To do so, a query language must provide facilities beyond the standard such as nested queries, and allow orthogonal composition of constructs.

- *Computational power*: Recursion and quantification characterize the computational power of a query language. Traversal recursive queries as well as quantification are supported poorly. Recursive queries with computation are supported even worse.

- *Support of collection*: Usually only one data structure 'Set' is widely supported and its operations are well defined. It is the case for other collection classes such as list and bag.

Therefore a good query notation, having as characteristics the above features is introduced. It is known by *Object Comprehensions*. Object Comprehensions Language (OCL) is based on *List Comprehensions* in [1] which is clear, concise and powerful.

## 1.1 Overview of the OCL Project Components

This report is a part of a team project work, which consists of establishing a testbed for the processing of Object Comprehension Language (OCL) queries using an AT&T object-oriented database, Ode, whose programming language O++ is based on C++. For that reason, OCL queries are to be translated into O++. Translated O++ queries are to be run against a university database based on Ode (Figure 1).

This idea initialized a team work of three interrelated projects: the first project objective is to build and manage a database, this constitutes the *database project.*

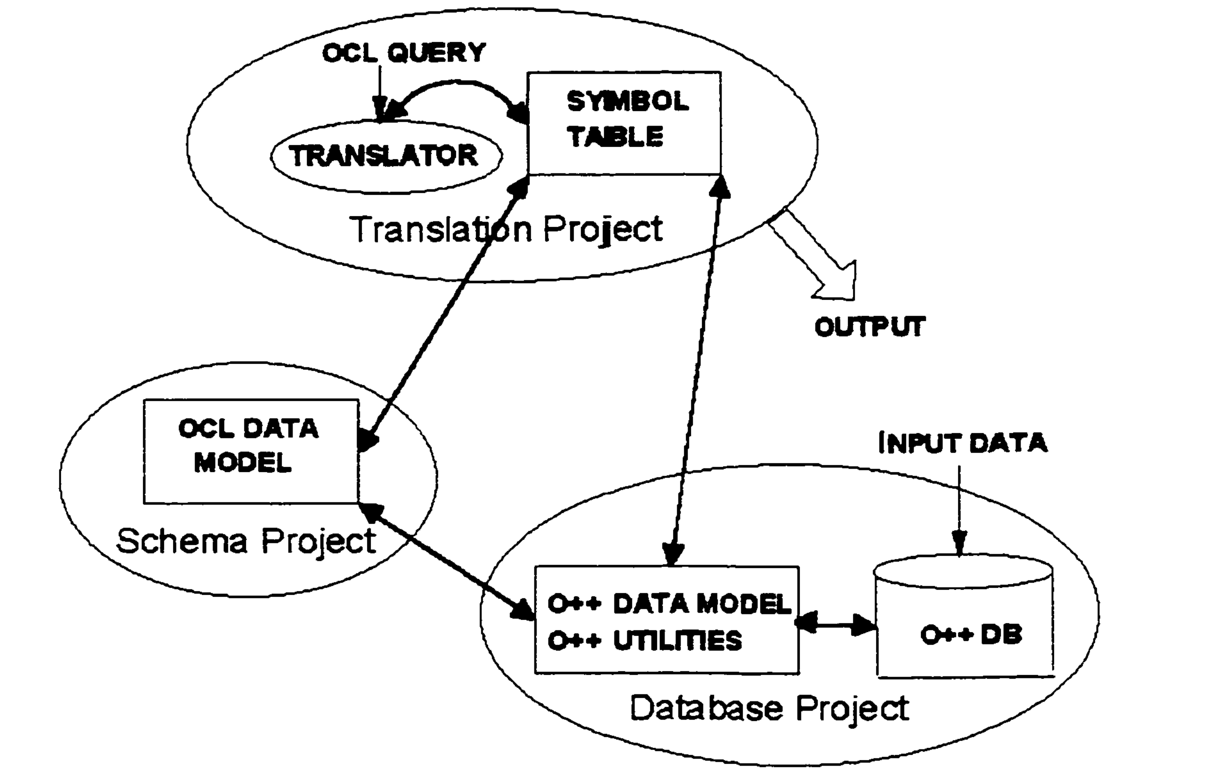

Figure 1 : OCL Query Flow Diagram

A symbol table is required for such a database, which defines another component, the *schema project*. This project is concerned with the implementation of the above mentioned symbol table.

The third team work component is to provide the translation of the user specified OCL queries into an interface query language that could be recognized by and run against the target database created in the database project. This last project is known as the *translator project*.

The following is a closer look at the above interrelated components:

- *Creating an ODE-based university database.*

  This thesis covers the part of the project whose objective is to set up and manage the university database. Many models were available to set up this database.

  A. Building the database using the relational model: the relation is the only construct required to represent the association among the attributes of an entity as well as the relationships among different entities. A relation can be considered as a table name. The relations found in the university database are as follows (Figure 2):

| **Entities** | **Relationships** |
|---|---|
| **Person** (Name) | **Pers_addr** (Name, Street, City) |
| **Address** (Street,City) | **Dept_addr** (Dept_name, Street, City) |
| **Staff** (Name, Salary) | **Belongs** (Name, Dept_name) |
| **Student** (Name, Student_id) | **Teaches** (Name, Code) |
| **Visiting_Staff** (Name) | **Major** (Name, Dept_name) |
| **Tutor** (Name, Salary) | **Takes** (Name, Code) |

**Course** (Code, Credits)

**Department** (Dept_name)

**Assessments** (Code, Value)

**Prerequisites** (Code, Prereq_Code)

**Runby** (Dept_name, Code)

The following is a tabular representation of some of the relations:

**Pers_addr**

| Name | Street | City |
|---|---|---|
| Ron | Guy | Montreal |
| Pam | Young | Toronto |
| Sam | Shepard | Toronto |

**Teaches**

| Name | Code |
|---|---|
| Dave Fisher | Comp 345 |
| Dave Fisher | Comp 520 |
| Johnny Brent | Math 201 |

**Prerequisites**

| Code | Prereq_Code |
|---|---|
| Comp 553 | Comp 536 |
| Comp 553 | Comp 551 |
| Comp 628 | Comp 546 |

Having repeated values into the table as shown above, the representation of a collection of values is implicit in the case of the relational model, it is implemented by adding more records with repetitive values into the tables, as seen in the **Teaches** and **Prerequisites** relationships.

This does not apply to the object-oriented model where different types of collection are set up as described in Chapter 4.

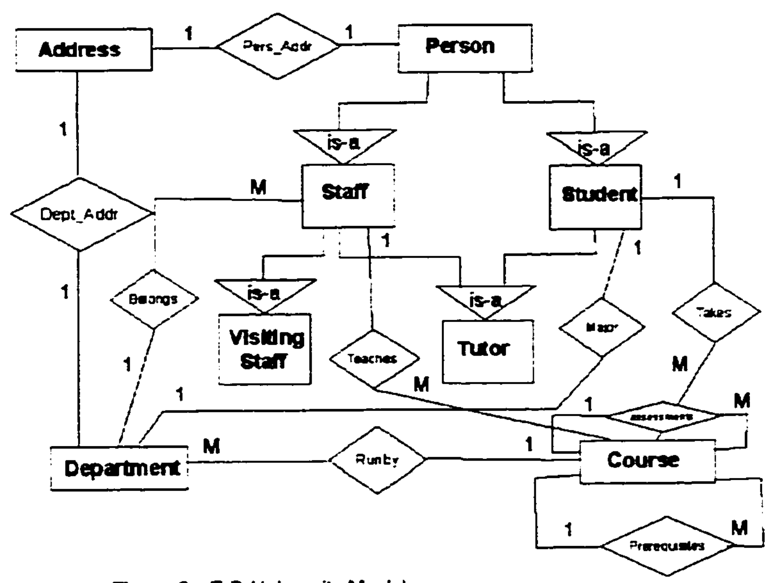

Figure 2 : E-R University Model

B. Building the database using the object-oriented model:

The target database is to be based on Ode, an experimental object-oriented database system from AT&T. It is defined, queried and manipulated in O++, the database interface programming language which is based on C++. The university database covers a university model, a schema of classes such as *Person*, *Department*, *Staff*, *Student*, *Tutor*, *Visiting Staff*, *Course*, and *Address*. The O++ based queries are to be run against the university database. Those queries are built using the Ode built-in query facilities and the O++ data structure classes to support OCL concepts, so called *utilities* for query processing such as Set, List and Bag. The query results then would be printed in the form of a collection of solutions.

- *Implementation of a university database symbol table.*

  The purpose of the schema project is to implement a symbol table for the above university database. It is to look into a BNF grammar for the schema definitions and implement the facility to input views described as schema's for those OCL queries which create objects of new classes.

- *Implementation of the OCL-to-O++ translator.*

  This part covers the translation of OCL queries to O++, an interface language for the Ode database system. Those OCL-to-O++ translated queries are to be run against the university database built in the database project. Therefore, the main objective of this component is to design and implement the OCL translator.

## 1.2 Organization

The organization of this report is as follows. Chapter 2 covers object-oriented database systems and query languages. Chapter 3 presents an overview of Ode and O++. Chapter 4 describes the university data model as a case study. Chapter 5 presents the conclusion. The Bibliography is followed by appendices which contain the code of the university model and utilities.

# Chapter 2

# 2 OODBMS and Query Languages

This chapter covers the advantages provided by object-oriented database systems, while relational database systems seem not to be adequate for certain applications. I describe the evolution of OODBMS, the query language associated to it, the features found in an OODBMS and missing in relational system. Then, I provide a high level description of object query languages, their evolution and the current situation in this field. Finally, a query notation called Object Comprehension Language [1] is studied in detail.

## 2.1 Relational Vs Object-Oriented Database Systems

Relational databases were developed in 1970 and gained market acceptance starting in 1980. The first object database implementation did not begin until 1982 [4]. and became popular in the mid-eighties, as a result of the increased popularity of object-oriented programming languages such as C++.

RDBMS now play a dominant role in the business oriented information sector [3]. Although RDBMSs fit the needs of most business applications, they are not adequate for the needs of information highway, multimedia technologies, time-series and spatial data applications. For instance, the relation schemes for banking enterprise can be defined easily using RDBMS:

```
Branch = (branch-name,asets,branch-city)
Customer = (customer-name,street,customer-city)
Deposit = (branch-name,account-number,customer-name,balance)
Borrow = (branch-name,loan-number,customer-name,amount)
```

But the declarations defined below for polygon, rectangle and table classes are more appropriate to object-oriented design than relational one:

```
Class Polygon
methods
        area (Polygon Type, xvalues, yvalues):->float

Class Rectangle isa Polygon
methods
        xvalues:->Integer
        yvalues:->Integer

Class Table isa Rectangle
methods
        xvalues:->Integer
        yvalues:->Integer
```

Currently, most applications are based on object programming, and programmers want transparent database storage and management of the object data model using inheritance, encapsulation, overriding, versioning, etc. All these features are efficiently handled by OODBMS, and application performance is increased accordingly.

Users want to distribute both data and logic throughout their network to deliver client/server applications: OODBMS provides one solution.

## 2.2 Why We Need Object Oriented Database Systems (OODBMS)

The need for OODBMS increased due to the shortcomings of the RDBMS model in the following areas [11]:

- The relational model needs to decompose each application object over several base relations (tables). For instance, operations, such as rotating an object in space, would require complex calculations over multiple tables, and extremely complex queries in order to represent the object in a query using table joins.

- Another drawback is the need to supply key-attributes to identify each tuple within each relation so relations can be joined together to represent the object through a query. Also, values do not have any identifiers, therefore they cannot be referenced directly. Using the banking information database, a joining procedure is needed to get the account number and address of each customer by having the customer-name as key attribute:

```
select account-number, customer-name, street, customer-city
from deposit, customer
where deposit.customer-name = customer.custmer-name
```

- Referential integrity has to be maintained between relations in order to preserve the integrity of the data. For instance, a foreign key that references a primary key must reference a valid primary key. The database is responsible for ensuring the validity of the references, and managing this responsibility requires extensive overhead:

```
Person_table = (person_id, person_name, address, company_id)

Company_table = (company_id, company_name, address)
```

  person_id is the primary key of Person_table.
  company_id is the primary key of Company_table.
  company_id is a foreign key in Person_table.

- Relational databases have only simple data types, such as strings, Booleans, and integers. Engineering applications require more complex data types, which do not exist in the RDBMS. Also, operations such as rotating an object in space, cannot be maintained by the RDBMS, but have to be stored and maintained in the application code.

## 2.3 OODBMS Evolution

The concept of object-oriented programming originated in the late sixties to early seventies when SIMULA and SMALLTALK-72 languages were developed [4].

Objects provided the initiative for the creation of the fifth-generation database technology. This new generation must be based on conventional database technology and must incorporate solutions to many problems evident in the use of relational databases.

The transition from the fourth-generation to the fifth-generation happened under three approaches:

- Extending an object-oriented programming language (OOPL): This approach is realized by adding persistent storage; concurrent access and transaction support to an OOPL.This approach has become the most popular in the commercial world as illustrated by ObjectStore, Versant, Objectivity, Gemstone, IRIS, and O2 [2].
- Extending a relational DBMS: This approach is created by enhancing an existing RDBMS with object-oriented features such as classes and inheritance, methods and encapsulation. Exemplary systems are Postgres95 and Starburst [7].
- Building an ODBMS from the ground up: This third approach is revolutionary because the whole system is built from scratch, as represented by UniSQL [4] and OpenODB. Orion, a research prototype, belongs to this category. These systems provide their own data models and data manipulation languages [7].

Lately, a new paradigm known as object-relational DBMS (ORDBMS) has emerged. Its objective is to support both relational and object-oriented database applications. Systems in this category are Illustra and DB2/6000 (for extended RDBMS) and OpenODB and UniSQL (for ground-up ODBMS) [7].

The real commercialization of ODBMS technology started in 1992. The early market concentrated on single-user applications, but now object-based multi-user applications are becoming popular. These applications are concentrated around:

- Office information systems: They include text, graphics, video and voice.
- Manufacturing systems: Hierarchical representation of manufacturing processes is related to the object concepts.
- Scientific applications: For example, medical and geographical information.

### 2.3.1 The Object-Oriented Definition

An object-oriented database system must satisfy two criteria:

- It should be a **DBMS**.
- It should be an **object-oriented** system.

A **DBMS** is a database system which allows the definition and manipulation of data and provide services, such as persistence, secondary storage management, concurrency control, recovery, and ad hoc query facility.

An object-oriented system must support objects, classes, inheritance and aggregation, encapsulation and methods.

An OO database is a collection of objects whose *behavior*, *state*, and *relationships* are defined based on the object-oriented **data model** [4].

This data model is defined as a **core model** augmented with semantic modeling concepts such as aggregation and generalization.

The basic components of the object-oriented data model concepts are [3]:

- *Classes*: Objects are instances of classes. Classes should have data members and methods. Data members are also called attributes or instance variables.
- *Methods*: Objects can have behaviors described in the method code.
- *Object and Object Identifier* (OID): Each object should have an unique object identifier.
- *Inheritance*: Classes can inherit members and methods from other classes.
- *Polymorphism*: Inherited methods can be overridden and late-binding of method calls can occur based on the object class.
- *Encapsulation*: Methods and values within objects can be protected.

In addition there are two semantic concepts which are essential for many types of applications:

- *Composite Objects*: known also as *complex objects*, which are built from simpler objects. A complex objects is a heterogeneous set of objects which form a part hierarchy. This is the aggregation relationship.
- *Version*: A version object is a set of objects which are versions of the same original object. A version object consists a hierarchy which builds relationships among various versions of the object.

With the above two concepts, the data model definition is complete and ready to be directly implemented in a database system.

## 2.3.2 OODBMS Advantages

The following advantages are specific to object-oriented databases:

- **Speed**: Queries can be faster because joins (used in relational databases) are often not needed. This is because an object can be retrieved by following object ids.
- **No Impedance Mismatch**: When both the relational data manipulation language (SQL) and the classical programming language do not fit together, an impedance mismatch occurs. This is due to three main causes [2]:

  a. *Type mismatch*: The type system of the programming language and the database system DDL are not the same. Relational databases deal with set of flat tuples, while programming languages often deal with single hierarchical records. Thus, conversion is needed when storing or retrieving data from the database.

  b. *Set versus element programming*: Relational databases operate on sets of tuples such as join, select; whereas programming language deal with one record at a time.

  c. *Declarative versus imperative programming*: Relational query languages are declarative, in contrast to the most widely used programming languages, which are imperative.

  The impedance mismatch has many undesirable consequences, which are manifested while extracting or writing data to databases, so more code has to be written to handle data conversion, and unnecessary communication is established between the programming language and database system.

The mismatch is solved by using persistent programming languages, which use the same paradigm in the programming languages as the database.

In the case of OO databases, the same object paradigm is used by both programming language and database. It is not necessary to do any format conversions when reading the data from disk or writing it to disk.

- **Programmers need to learn only one programming language**: The same programming language is used for both data definition and data manipulation.
- **Complex applications**: The full power of the classes in the database programming language can be used to model the data structures of a complex application.
- **Versions**: Object-oriented databases provide better support for versioning (as we will see later in the case of ODE).
- **Triggers and constraints:** Object-oriented databases (such as ODE) provide support for triggers and constraints. These form the basis of active databases [14].

Finally, all object-oriented applications that have database needs will benefit from using object-oriented databases. Specifically, C++ applications that need a database, can benefit from the use of the Ode database system.

## 2.4 Evolution of the ODBMS Query Language

In the early days of object-oriented database management systems, it was believed that query languages were not needed [3], but later this perception vanished totally as ODBMS acquired an increasing portion of the market place, and the need for certain query language features became crucial.

### 2.4.1 History

A good example of a 'first generation' object-oriented database language supporting queries is the O++ database programming language (details later). O++ is basically an extension of the C++ language with the support of persistent objects. O++ integrates user queries through the programming language itself. These queries were totally non-declarative, using pointers to object members to return results [8]. The queries were specific to the internal representation of the object, thus, they were not portable between object models and violated the encapsulation rule.

### 2.4.2 Next Generation - Object Query Language

With time, it was understood that better object-oriented query languages founded on a well-defined object-oriented data model were needed. The language should be concerned with the conceptual schema of the data model, not the internal representation of the model.

The requirements of object query language over relational languages include [3]:

- *Method Invocation*: to be able to invoke an object's method as a query predicate:

  - Return staff members earning more than $1000.

  ```
  Set [ s <-- StaffMembers ; s.salary > 1000 | s ]     /* salary is the object method */
  ```

- *Object Equality*: A predicate should be able to use several equality comparisons on objects using operator overloading
- *Class Hierarchy*: A query expression should be able to query objects for their place in the inheritance hierarchy, such as the IS-A predicate:

- Return Student living in Glasgow.

```
Set [ s <-- Student ; s.address.city = "Glasgow" | s ]
  /* address is the object method inherited from Person */
```

- *Class Joins*: A query must be able to join together objects based on a predicate and return tuples based on results from joined objects:

  - Return students studying in the same department of SteveJ.

```
Set [ x <-- Students; y <-- Students; x.name = = "SteveJ";
     x.major = = y.major | y ]
```

- *Recursion*: A query must be able to perform recursions, such as all prerequisites of a course, where the prerequisites themselves are courses:

  - Return all direct and indirect prerequisite courses for the 'DB4' course.

```
let f ( cs : Set of Course ) be
  cs UNION Set [ x <-- cs; y <-- f (x.prerequisites) | y ]
  in Set [ c <-- Courses; c.code = 'DB4'; p <-- f (c.prerequisites) | p ]
```

- *Integration*: The language must be able to be integrated with languages such as C++.

### 2.4.3 Current Situation

In 1993, the Object Data Management Group (ODMG) published a set of object languages. They combine the best features of O2, OQL [9] and SQL's DML and DDL languages. The ODMG-93 language is heavily based on programming language standards, in particular C++ and Smalltalk. Three languages are defined in the ODMG-93 standard: OQL (Object Query Language), OML (Object Manipulation Language) and ODL (Object Definition Language).

Another candidate for an object-query language, SQL3 [10], which is being promoted by the SQL legacy, is an attempt to turn the SQL-92 (for relational databases) into an ODBMS query language. While ODMG-93 is considered a true object-oriented language, SQL3 is based on the extended relational model.

## 2.5 The Future of ODBMS and Query Languages

Most likely, future research in ODBMS query languages will take place in industry rather than in academia. This has been the case with RDBMS query languages. For instance, SQL was first developed in academia and then further developed by industry.

The most likely candidates for further research and development are the two previously discussed languages, SQL-3 and ODMG-93 [3]. It seems that most vendors will turn to the object data model definition proposed by the ODMG-93 data model definition. The success of SQL-3 in ODBMS is still not sure. While SQL-3 is considered to be the query language of choice in the extended RDBMS world, it has still to be standardized as the query language of choice for the ODBMS world.

The market for ODBMS is still quite small. There is no doubt that ODBMSs will gain increasing market share and the future looks very promising for ODBMS databases. When we have millions of users working with object-oriented systems in different complex applications, not only will ODBMS gain market share, but the query languages will be more developed and standardized. The popularity of the internet will most definitely encourage the use of object-oriented systems. ODBMSs will not replace relational DBMSs in conventional database markets such as inventory management, finance, airline reservations. Rather, the use of ODBMS will be restricted to complex applications such as design engineering and network management.

Finally, as consumer demands grow and technology evolves, applications will support more complex type information, more relationships and dependencies. All this means, that as time goes on, more applications will move from the domain of traditional databases to that of ODBMSs.

## 2.6 Object Comprehensions: Background

Existing object-oriented query notations have been criticized for being unclear, inefficient and computationally weak. According to [1], the inadequacies are categorized into the following four items:

- *Support of object-orientation*: A few object-oriented query languages do not capture the class hierarchy which is defined by the *ISA* relationship between classes.
- *Structuring Power*: By definition, structuring power refers to the ability to generate complex objects that are components of object-oriented design. To do so, a query language must provide something like nested queries and allow orthogonal composition of constructs.
- *Computational Power*: The power of a query language is characterized by *recursion* and *quantification*. For instance, the traversal of recursive queries is only supported by ORION [4], and system supports recursive queries with computation. The support for quantification is generally poor.

- *Support of Collection*: The *Set* structure is widely supported and its operations are well defined, but this is not the case for other collection classes, and the interaction between different collection classes is not defined.

Therefore, a good query notation is introduced taking into consideration the above weaknesses and the fundamental properties of object-oriented data models. This new query notation is known as *Object Comprehensions*, and is clear, concise, powerful and optimizable. The extension of *List Comprehensions* [16] to *Object Comprehensions* was done by consolidating and improving constructs found in existing query languages.

### 2.6.1 Object Comprehensions Language (OCL)

The standard mathematical notation for sets was the inspiration for *comprehensions*. Thus, *Comprehensions* languages are based on set notation such as the definition of the following set of squares of all the odd numbers which is conventionally written as:

$$\{\text{ square x } \mid x \in \text{s and odd x}\}$$

*Comprehensions* first appeared as *Set Comprehensions* in an early version of the programming language NPL, which later evolved into *Hope* [5] but without *comprehensions*. They were followed by *List Comprehensions* [6].

The above mathematical expression written using *List Comprehensions* is as follows:

[square x | x <-- s; odd x]

where *s* stands for a list instead of a set.

The syntax of *List Comprehensions* is as follows:

Q:: = ... | [ E | Q]

$$Q ::= E \mid P \leftarrow E \mid \wedge \mid Q : Q$$

where E stands for an expression, Q stands for a qualifier, P stands for a pattern, and O stands for an empty qualifier.

The result of evaluating the *comprehension* [E | Q] is a new list, computed from one or more existing lists. The elements of the new list are determined by repeatedly evaluating E, controlled by the qualifier Q. A qualifier is either a filter which is a boolean expression such as *odd x*, or a generator such as the (x <-- s) above, making *x* range over the elements of the list *s*. More generally, a generator of the form (P <-- E) contains a pattern P that binds one or more new variables to components of each element of the list.

Recently, *List Comprehensions* was generalized to *Collection Comprehensions* [1] which provides a uniform notation for expressing queries over different collection classes such as bags, lists, trees, and sets. The major improvement is that only one query notation is needed for all collection classes.

Here are some examples of *object comprehensions* using the notation suggested in [1]:

**Example 1.** Return students having Steve before Bob in their supervisor lists.

```
Set [ s <-- Students; s.supervisedBy.[Steve : Bob] ~ = = List [ ] | s]
```

**Example 2.** Return students taking exactly two courses given by Steve Johnson.

```
Set [ j <-- StaffMembers; j.name = 'Steven Johnson';
        s <-- Students; SOME s.takes = JUST 2 j.teaches | s ]
```

**Example 3.** Return a set of staff members from a certain set (StaffMembers) provided they comply with a specific condition such as earning more than $3000 a month.

```
Set [ s <-- StaffMembers; s.salary > 3000 | s ]
```

It is worth mentioning that *comprehensions* are a declarative specification of a query, and as is shown in [1], are a good query notation for being concise, clear, expressive and easily optimized.

# Chapter 3

## 3 Overview of ODE and O++

ODE (Object Database and Environment) is an object-oriented database system developed for research at AT&T. The primary interface for the Ode database is the database programming language O++ [8], which is based on C++. O++ extends C++ with facilities for creating and manipulating persistent objects stored into clusters, defining and manipulating sets, querying the database, specifying constraints and triggers and running transactions.

The Ode database is based on a client-server architecture. Each application runs as a client of the Ode database. The storage manager used by Ode is EOS [14]. The Ode database handles single user applications that can run without a separate server. It also supports multiple applications which can concurrently access the database.

## 3.1 Database Structure

The Ode database consists of a number of files called *areas* [12]. Each area is organized as a tree structure (Figure 3). A number of databases can exist in a single Ode area. Information about these databases resides in block number two of the area which serves as roots of the trees for each specific database. Each database entry contains information about the address location of its root file and the name and version of this specific database.

There are three basic object types in an area: *file object, data object, and index object.*

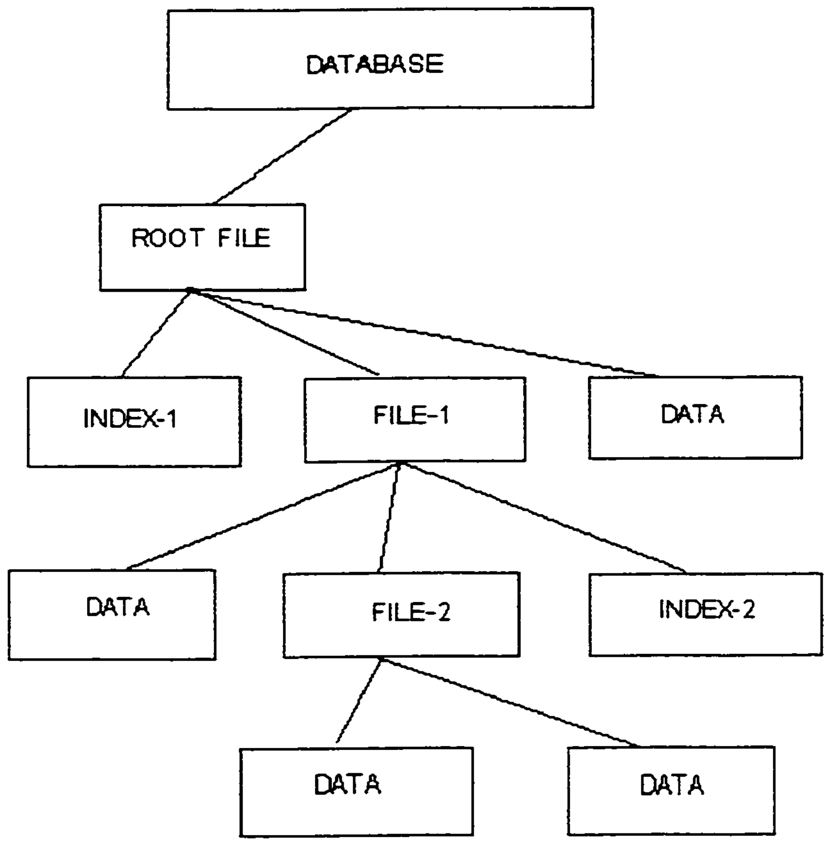

Figure 3: Object Storage

### 3.1.1 File Object

*File objects* are composed of a file header and a list of block numbers. Each block number serves as a pointer to the blocks which contain other objects that belong to the specific file. These objects can be *data objects, index objects,* or other *file objects.*

### 3.1.2 Data Object

*Data objects* contain user data, they are organized into persistent objects identified by a unique object id.

### 3.1.3 Index Object

*Index objects* are composed of an index object header and a list of block numbers, each block number points to a block representing one of the buckets in the hash table. The index object header is at the same time the header of the hash table and contains information such as hash key size, hash value size, key type, etc. Each bucket in the hash table has a bucket header and a list of bucket slots. The bucket header contains information about the number of bucket slots in the bucket, space management, hash key size and value. Each bucket slot indicates the location of key/values pair in the bucket.

*File objects, index objects* or *data object*s that have a size greater than a page (4096 bytes) will be organized into *large objects. Large objects* are also organized as a tree structure; each node in the tree contains a large object header and a list of number/size pairs. The large object header has the information about the level of the tree which indicates the number of nodes that user has to traverse before reaching the final file, index or data object.

## 3.2 Language and Data Model

### 3.2.1 O++ - Database Programming Language for ODE

The Ode database is defined, queried and manipulated using the database programming language O++. The O++ compiler translates an O++ program into a C++ program that contains calls to the ODE object manager library. The library provides facilities for creating and manipulating persistent objects. The translated program is then compiled with the C++ compiler and linked with the object manager library to form a load module ready to execute (Figure 4).

The C++ object model called the *class* is used as the basis for the object model of O++. The class facility supports data encapsulation and multiple inheritance. O++ extends C++ classes by providing facilities for creating and manipulating persistent, volatile objects and their versions, and associating constraints and triggers. Most of the O++ code interacting with the database should be within a transaction block. Also. O++ language provides query facilities on objects found in clusters. The major lack of this query language is that it can't perform 'join' queries, thus O++ alleviates these problems by providing iterators that allow sets of objects to be manipulated as easily as by declarative query languages (SQL). The iteration facility of O++ also allows the expression of recursive queries.

### 3.2.2 Data Model

#### 3.2.2.1 Objects Definition

The O++ object model is based on the C++ object model as created by the *class* facility. A *class* declaration consists of two parts: a specification and a body.

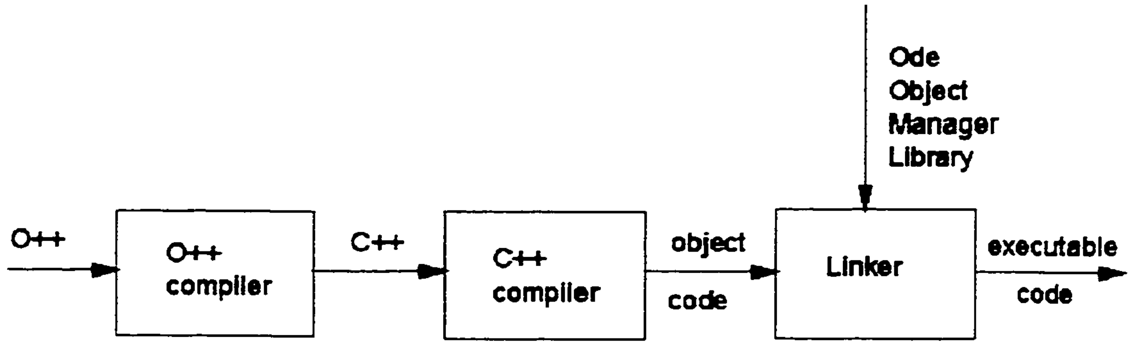

Figure 4 : Compilation of an O++ program

The specification contains the necessary information for the user of a class. For example.

```
Class address {
private:
        char street[20];
        char city[20];
public:
        address(char*,char*);
        char* ret_street( );
        char* ret_city( );
};
```

the body consists of the bodies of methods declared in the class specification. For example,

```
address::address(char *s, char *c)
    {
      strcpy(street,s);
      strcpy(city,c);
    };
```

#### 3.2.2.2 Inheritance

Inheritance allows more specialized objects to inherit properties (data and functions) of more general objects. A derived class is specified by following its name with the name of the superclass. A derived class inherits all data items as well as the member functions of the superclass:

```
class  stockitem: public item  {
          int consumption;  /* qty consumed / year */
          int leadtime;       /* lead time in days */
public:
          int qty;
          double price;
          stockitem( Name iname, double iwt, int xqty, int xconsumption,
                     double xprice, int xleadtime);
};
```

Stockitem is the same as item in addition to other information such as quantity in stock, its consumption per year, its price and the lead time.

Multiple inheritance allows a new class to be derived from multiple classes:

```
class  tutor: public staff, public student {

............

public:

............

};
```

### 3.2.2.3 Named Persistent Objects

O++ considers two types of memory: *volatile* and *persistent* [14]. *Volatile* objects are allocated in volatile memory as those created in any programming languages. *Persistent* objects are allocated in persistent store and continue to exist after the program that created them has terminated. Persistence should be a property of object instances and not types, and it should be possible to allocate objects of any type in either volatile or persistent store.

Also, it is possible to copy *persistent* objects to *volatile* ones and vice-versa to speed up performance.

Each persistent object is identified by a unique identifier, called the object identity (oid). The object identity is referred to as a *pointer to a persistent object.* Persistent storage operators *pnew* and *pdelete* are used as follows:

```
persistent stockitem *psip;  /* persistent pointers */

stockitem *sip;  /* volatile pointers */

psip = pnew stockitem("1m dram", 0.05, 5000, 10000, 7.5, 15);

*sip = *psip;

pdelete psip;
```

psip is a pointer to a persistent stockitem object, then copy the object pointed to by psip to the object pointed to sip, and delete the object pointed to by psip.

### 3.2.2.4 Persistent Objects Clusters

Persistent objects of the same type are grouped together into a *cluster* [13]. The name of a cluster is the same as that of the corresponding type. The Ode object manager views the global database as a collection of local databases called *cluster groups.* Each cluster contains objects of the same type, and they can point to each other within the same cluster. Each *cluster group* manages two copies of persistent objects: the one found on disk and the one found in memory. When a program starts executing and needs to access the required persistent objects, a process copies the persistent objects from disk to memory. This process is called *activation.* The cluster group tracks the objects that have been activated and where in memory they are located. The reverse operation, writing a modified persistent object back to disk, is called *passivation* or *synchronization.*

*Clusters* are implemented using the class template *CLUSTER*, having as parameter the type of the objects the *cluster* will contain. The following steps describe the scenario where a new created object becomes persistent:

- The object manager creates an object by invoking the *pnew* operator.
- To make the object persistent, it must be inserted into a *cluster* (created before).
- The insertion is done by saving a pointer to the required object into the *cluster group*.
- Later, the object is written to disk, and deleted from memory.

## 3.3 Database Facilities

### 3.3.1 Transactions

*Transactions* deal with objects in the database by invoking operations on the objects. Thus, all code interacting with the database (except database opening and closing) must be within a transaction block of the form *trans { ... }* .There are three kind of transactions:

- Update: have the form *trans { ... }*
- Read-only: have the form *readonly trans { ... }*
- Hypothetical transactions allow users to ask 'what if' scenarios and they have the form: *hypothetical trans { ... }*

The following is an example of an *update* transaction:

```
#include <stdio.h>
#include "db.h"  /* classes Name and Addr */
#include "supplier.h"  /* class supplier */

class Item {
        Name nm;
        float wt;
        float pr;
public:
        .......
        float price( );
        .......
};

main ( )

{

persistent Item *pip, *pip1, *pip2;

/* add two new persistent items and get the expensive one */

trans {
        *pip1 = pnew Item("twidleedee",320,125);
        *pip2 = pnew Item("twidleedee",350,75);
```

```
            for (pip of  Item)
                  suchthat(pip->price( ) > 100.00)
              printf(.....);
          }
      }
```

In the above example. the creation of new persistent objects is done within the transaction block. The same applies to the for iteration command.

The ODE database server (EOS) provides transaction management facilities as follows:

- *Concurrency Control*: EOS supports multiple transactions to access a database at the same time through read mode, and only one transaction to write an object. This is accomplished using the MultiGranularity 2-version 2-phase (MG-2V-2P) [15] locking. All transactions acquire locks on data items before they access them, and all locks are released when transactions are finished (committed or aborted). Three types of lock granularities are supported: *page-level*, *file-level* and *database-level*. A *page-level* lock consists of a page locked in shared. exclusive or commit mode. The *file* or *database* can be locked in one of the following modes: no lock, shared. intention shared, exclusive, intention exclusive, shared intention exclusive, commit and intention commit.

  EOS avoids the *starvation* problem associated with 2-version protocol by blocking all new readers when the writer wants to *commit*.
- *Deadlock* is also possible in this locking scheme: deadlock detection is performed each time a lock request is blocked by another transaction which is blocked too. Read-only transactions are less likely to *deadlock* because they request only read locks. The ODE system may also abort transactions to break *deadlocks* by issuing the 'abort' statement which explicitly aborts the transactions.

- *Logging:* EOS maintains two types of logs: *private* and *global* [15]. Each *private log* is associated with one transaction only. The log records are *redo* records, which contain the results (after images) of the updates generated by the corresponding transactions.

  The *global log* contains records that are either commit or checkpoint records. A commit log record contains the committed transaction's id and other information related to the transaction's updates. The checkpoint record contains the ids of the committed transactions at the checkpoint time.

  *Log* files are used by EOS for recovery: The EOS server returns the database to the last consistent state prior to failure as recorded in the *log* files.
- *Transaction Commit*: A transaction is declared committed if the following steps are performed successfully:
  - A. The records, updated by a transaction in its private log, are flushed onto stable storage.
  - B. A *commit* log record is inserted into the global log.
  - C. The global log is flushed onto stable storage.
- *Checkpoint*: To reduce the amount of work the recovery manager has to do, the EOS server periodically issues checkpoints. During a checkpoint, dirty pages found in the shared pool are moved to the stable storage. When the checkpoint process is completed, a checkpoint record is inserted in the global log file.

## 3.3.2 Query Processing

O++ language can query objects found in a cluster, subcluster or set using a *for* loop of the form:

```
for j in s in set-or-cluster-or-subcluster
        [suchthat-clause] [by-clause] statement
```

For instance, to print the name of people whose city address is Ottawa:

```
persistent person  *pers;
persistent address *per_addr;
..............
for (pers in person)
        such that (pers.per_addr.ret_city() == “Ottawa”)
              printf(“%s %s\n”, pers->get_name());
```

This loop will iterate over all *pers* of type person in order to find someone whose city is ‘Ottawa’. The *such that* clause is used to restrict the search of objects that satisfy a boolean expression. This clause is similar to the *where* clause in SQL.

*Joins* can be performed using nested *for loops* or a loop with multiple loop variables as in:

```
for (pe in person; ad in address) {
..........
};
```

Also, using *in all* clause instead of *in* in a ‘for loop’ statement causes the loop to return results consisting of persistent pointers of all objects of a particular type and types derived from it through inheritance:

```
for (pe in all person) such that (pe->get_name() == "GEORGES AYOUB")

        printf(...........);
```

is equivalent to the permutations of the following two loops:

```
for (pe in all staff) such that (pe->get_name() == "GEORGES AYOUB")

        printf(...........);

for (pe in all student) such that (pe->get_name() == "GEORGES AYOUB")

        printf(...........);
```

provided the classes Staff and Student inherit the class Person.

One restriction to the *in all* clause is that it works only when the class declaration use single non-virtual inheritance.

As a more in-depth example of query processing in O++, the following is a manual translation of *object comprehensions* queries into O++ queries of Ode. The OCL sample queries are from chapter 2.

1. Set [ s <— Students; s.supervisedBy.[Steve : Bob] ~ = = List [ ] | s]

```
Set<Students> tempset;

for s in Students
{
    List<StaffMembers> templist;
    if ( s.supervisedBy.ssublist("Steve","Bob") ! = templist)
    tempSet.add(s);
};
```

2. Set [ j <— Staff; j.name = 'Steven Johnson';
    s <— Students; SOME s.takes = JUST 2 j.teaches | s ]

```
Set<Courses> tempset;

for j in StaffMembers
    suchthat (j.name = = String("SteveJohnson"))

for s in Students
    suchthat (s.takes.every(j.teaches))
    tempSet.add(s);
```

3. Set [ s <-- StaffMembers; s.salary > 3000 | s ]

```
Set<StaffMembers> tempset;
for s in StaffMembers
        suchthat (s --> salary > 3000)
        tempSet.add(s);
```

## 3.3.3 Versioning

In Ode, all persistent objects can have *versions* and there is no pre-defined limit on the number of *versions* an object can have [13]. As in persistence, *versioning* is an object property and not a type property. All objects of the same type can be *versioned* and can have different number of *versions*.

An object and all its *versions* are treated as one logical object with one object id. Accessing an object using a logical object id results in access to the current *version* of the object.

A new *version* is created by invoking the macro newversion : v1=newversion(p) where p is a logical object id. Ode recognizes the fact that the object referenced by v1 was 'derived' from the object referenced by p. O++ tracks the relationships between *versions* and provides facilities to traverse them. Given a logical object id, operator pdelete deletes the object and all its *versions*.

## 3.3.4 Constraints

*Constraints* are used to maintain consistency in the Ode database. Providing integrity *constraint* facilities in a database is not a new issue since all commercial databases today provide a certain level of integrity. This new integrity aspect is specific to object-oriented databases, where objects are updated and the database is left in a consistent state. *Constraints* are used to ensure this kind of data integrity.

*Constraints*, which are Boolean conditions, are associated with class definitions. All objects of a class must satisfy all *constraints* associated with the class.

*Constraints* in ODE are of two parts [13]: a predicate and an action (or handler): the action is executed when the predicate (condition) is not satisfied. *Constraints* checking is performed at object level or transaction level, thus ODE supports two types of *constraints*:

- *Hard constraints* are specified in the *constraint* section of a class definition as follows:

```
constraint:
        constraint1: handler1
        constraint2: handler2
        .............
        constraint(n): handler(n)
```

*constraint(n)*: is a Boolean expression that refers to component of the specified class.

*handler(n)*: is a statement that is executed when a *constraint* is violated.

Here is an example of a *constraint* declaration:

```
constraint:
        supplier_state = = Name("NY") | | supplier_state = = Name("");
```

For this type of *constraints*, checking is performed as soon as the object is updated: if any *constraint* associated with an object is not satisfied, and no handler exists to rectify it, then the transaction causing the violation will abort. If there is a handler associated with the *constraint*, then this handler is executed and the *constraint* is reevaluated immediately. If the *constraint* is still not satisfied, then the transaction in question is aborted.

- *Soft constraints* are specified like *hard constraints* except that the keyword *soft* precedes the keyword *constraint.*

  For this type of *constraint*, checking is performed at the transaction - level: that is, checking is deferred until the end of the transaction causing the update. Therefore, whenever a *constraint* violation occurs, it has to be rectified before the transaction causing the violation can commit.

### 3.3.5 Triggers

*Triggers*, like constraints, check the database for certain conditions, except that these conditions are not related to consistency. A *trigger* is specified in the class definition, and can apply only to the specific objects with which they are activated.

A *trigger* consists of two parts [13]: a condition and an action:

```
trigger:
        [perpetual] T1 (parameter-decl1):trigger-body1
        [perpetual] T2 (parameter-decl2):trigger-body2
```

T1,T2 are the *trigger* names. *Triggers* parameters can be used in the *trigger* bodies. There are three types of *triggers*: *once-only* (by default), *perpetual* (specified using the keyword perpetual as the example above), and *timed trigger*.

A *once-only trigger* is automatically deactivated after the *trigger* has 'fired', and must then be reactivated explicitly if desired. A *perpetual trigger* is automatically reactivated after being fired. A *timed trigger* must fire within a specified period of time, otherwise the timeout action is fired. As an example of a *once-only trigger*:

```
class inventitem: public stockitem {
public:
        inventitem (.....);
        void deposit (int n);
        void withdraw (int n);
        ......

trigger:
        order( ): qty < reorderlevel( ) => place_order(this, eoq( ));
                                  /* "this" refers to the object itself */
```

The action associated with the *trigger* order will be executed after its condition becomes true.

*Triggers* are explicitly activated after creating the corresponding associated objects. A *trigger* Ti associated with an object whose id is object-id is activated by the following call:

```
object-id -> Ti(arguments)
```

If successful, the *trigger* activation returns a *trigger* id, otherwise 0 is returned.

*Triggers* may be deactivated explicitly before they have fired as follows:

```
~trigger-id
~object-id -> Ti (arguments)
```

An active *trigger* fires when its condition is true. Firing means that the action associated with the *trigger* is scheduled for action. *Trigger* actions are initiated as separate transactions (*triggered* transactions). They act independently of the transaction causing the *trigger* to be fired (*triggering* transaction). The reason for this is that the *triggering* transaction should be allowed to commit even if the *triggered* transaction aborts for some reason.

Finally in Ode, a *trigger* or *constraint* is said to be *intra-object* if it is associated with a specific object, and the condition associated with it is evaluated only when the object is updated. Otherwise a *constraint* is said to be *inter-object.*

## 3.4 ODE Database Creation and Load

The steps needed to start Ode and build a database, are summarized as follows:

- Ode uses a client-server environment, thus commands are provided to create the initialization files:

  On the server:

  a. Run make_server_init and make_format_init.

  b. Start the server using oderserver.

  On the client:

  1. Run make_client_init and makes_format_init.
  2. Make sure that the EOS_SERVER_HOST in file ~/.eos/clientrc for the client is set to the name of the machine where the server is running.

- When the server is running, the command odeareaformat (*without* the -l) is issued to create a database area. A database area may contain one or more databases. When the server is down, odeareaformat *with* the -l option is used to create an area.

  ```
  odeareaformat -l database_area -o
  ```

  (The size of the *database_area* just created is very big).
- Include the header file ode.h which automatically makes available the class database in order to provide functions for manipulating (open,close, etc.) the database and naming persistent objects.
- Most of the operations interacting with the database are invoked within the transaction block except the open, close and remove database operations which are invoked from outside. Update transactions have the form: trans { ... }, which allow insertion into the database.
- The command to open or create the database should be initiated next outside the transaction body. An O++ database is identified by the name of the file in which the database resides. When a name (with no embedded '/') is used, O++ creates the database in the database area.

  Otherwise, O++ opens an already existing database after specifying the full path name:

  ```
  if ((db = database::open("~/path/datbase_name")) = NULL) {
      cout << "cannot open database_name" << endl;
      exit(1);
      }
  ```
- Define classes whose objects are to be made persistent by using the form:

```
persistent class class-name;

persistent class person { ... };
```

- Create persistent objects using the operators pnew. This operator returns a pointer to persistent objects, known as *persistent* pointers:

  ```
  persistent class_name *pe;    /* persistent pointers */

  pe = pnew person;
  ```

  The creation of persistent objects should be within the transaction block.

# Chapter 4

## 4 Object-Oriented Case Study

This chapter covers the university data model built using O++, provides a detailed description of the corresponding schema of classes and the utilities such as bag, list and set written to support the processing of OCL queries to be run against the university database.

## 4.1 University Data Model

The data model is a simplified university administration system that records information about students, staff members of a university, its departments and courses. The relationships between classes are defined in figure 5 .

The class *Person* has two subclasses: *Staff* and *Student*. *VisitingStaff* is a subclass of *Staff*. *Tutor* inherits from both *Student* and *Staff* in case students can do part-time teaching. The calculation of the salary of a tutor is different from that of a staff member. Every person has an address which is an object of class *Address*. Each student can have many supervisors modeled by the method *supervisedBy* as a list of staff members. Every staff member and student are associated to an object of type *Department* via *department* and *major* respectively. Courses offered by the university, taught by staff members and taken by each student are also recorded. They are represented by set-valued methods, *teaches* and *takes*. A course may have a set of prerequisite courses (*prerequisites*) and is administered by a set of departments (*runBy*). Also, the percentage weights of assessments (*Credits*) is given for each course. A course is an instance of the class *Course*.

It is assumed that the database is made of six set collections:

- *Persons* : Set<**Person**>
- *Departments*: Set<**Department**>
- *Courses:* Set<**Course**>
- *StaffMembers*: Set<**Staff**>
- *Students*: Set<**Student**>
- *Tutors:* Set<**Tutor**>

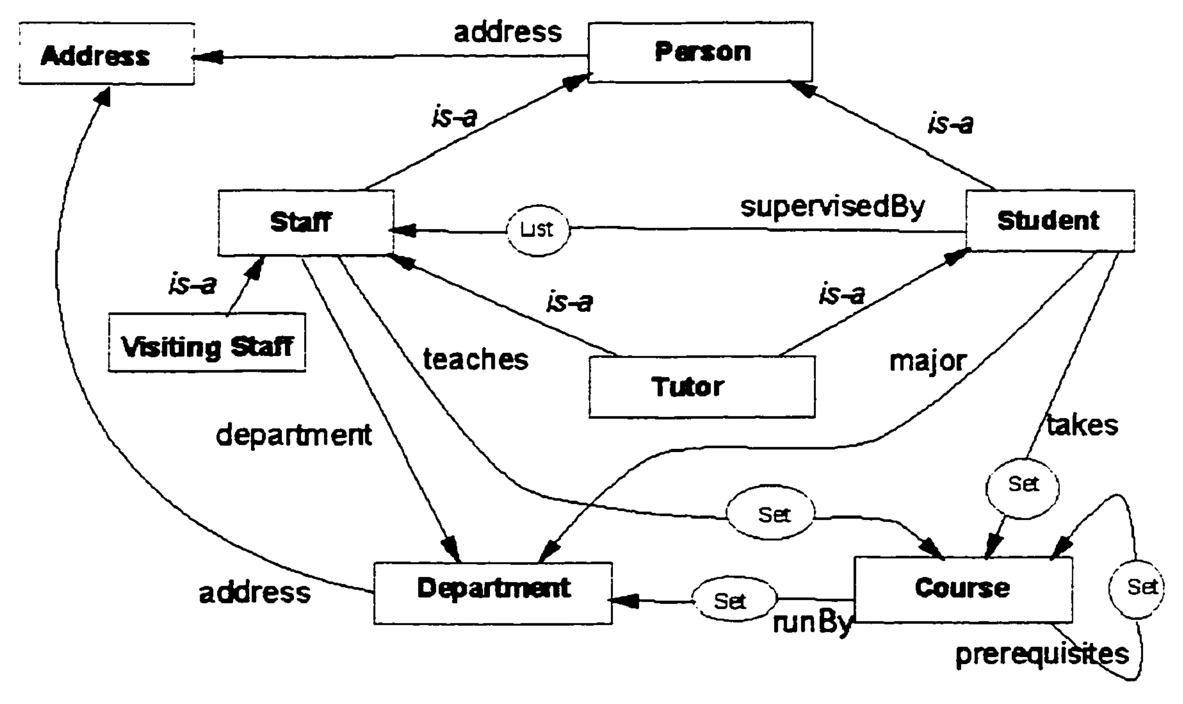

Figure 5 : University Model Diagram

The following is the description of the schema definition of the university database:

```
Class Person isa Entity
methods
   name ->String
   address ->Address

Class Staff isa Person
methods
   department ->Department
   teaches ->Set of Course
   salary ->Integer

Class Student isa Person
methods
   major:->Department
   supervisedBy->List of Staff
   takes ->Set of Course

Class Tutor isa Staff, Student
methods
   salary:->Integer

Class VisitingStaff isa Staff

Class Department isa Entity
methods
   name ->String
   address ->Address

Class Course isa Entity
methods
   code:->String
   runBy:->Set of Department
   prerequisites ->Set of Course
   assessments:->Bag of Integer
   credits:->Integer

Class Address isa Entity
methods
   street:->String
   city ->String
```

## 4.2 Utilities

Utilities are used in *Object Comprehensions* queries and in several object-oriented data models. They are defined as a *collection* of objects grouped as *list, set,* or *bag*:

- *List* is a *Collection* of ordered elements; duplicates allowed.
- *Set* is a *Collection* of elements with no duplicate and no order.
- *Bag* is a *Collection* of elements with possible duplicates and no order.

The class *collection* is defined to manage the above *collection* of objects (*list, set* or *bag*) by declaring methods to be used by all subclasses representing the three types of *collection* objects. In addition, more specialized methods are declared for each one of the subclasses defining the behavior of the different instance variables.

The following is the schema definition of the *collection* classes:

(More detailed description in Appendix A)

```
Template<Type>
Class _node
methods
    put_next->Void
    get_next ->_node
    search ->Integer
    get_info -> <Type>

Template <Type>
Class _Collection
methods
    size ->int
    ret_head ->_Node
    member ->Boolean
    remove ->Void
    intersects ->Boolean
    intersection ->Void
    every ->Boolean
    union_col ->Void
    atleast ->Boolean
    atmost ->Boolean
    just->Boolean
    differ ->Void
    destroy->Void
    display->Void

Template <Type>
Class list isa _Collection
methods
    is_empty->Boolean
    remove ->Void
    del ->Void
    findno ->Int
    insert ->Void
    append ->Void
    frequency->Integer
    show -> <Type>
    isublist ->List of <Type>
    sublist ->List of <Type>

Template <Type>
Class bag isa _collection
methods
    add ->Void
    frequency ->Integer

Template <Type>
Class set isa _collection
methods
    add ->Void
    union_set->Void
```

Many data structures are defined in order to facilitate the access the elements of a *collection*:

- **Iterators**

They are used to access the elements cf a *Collection* sequentially without exposing its internal structure. The order in which the objects are visited depends on the type of the *Collection.* Objects in a sequential *Collection* are accessed in the order in which they are added or sorted. Objects of any other *Collections* are visited in an undefined order. The abstract interface of all iterators is defined by the class *_iter_list*. An iterator object is responsible for keeping track of the current element; it knows which elements have been traversed already.

The following iterator template class is used by the Utilities members in order to visit the elements in any *Collection* type (*set, list, bag*):

```
Template <class Type>
class _iter_list {

private:
        persistent _node<Type> *entry;
        persistent _node<Type> *pos;

public:
        _iter_list( ...)
        persistent _node<Type> *POS() {return pos;}
        void first() {pos=entry;}
        void next() {if (pos) pos=pos()->next;}
        Type *current() {....}

};
```

First, the list to traverse must be supplied. Once the *_iter_list* instance is ready, its elements can be accessed. The *_iter_list* is made up of two items referring to position: The 'entry' and 'pos' which are initialized to point at the head of a list (beginning position).

The Pos() returns the position of the current element in the list.

First() initializes the current element to the first element.

Next() advances the current element to the next element.

Current() returns the current element in the list.

The following example shows how **iteration** is used in the *Destroy()* method which deletes all members of a *Collection()*:

```
_iter_list<Type> iter(this); /* Create an iterator named iter */
iter.First(); /* position the current element pointer to the first element */
cursor=iter.Pos(); /* get the current position and assign it to cursor */

While (cursor) {

        tmp = iter.Pos() ; /* get the cursor position and assign it to tmp */
        iter.next(); /* advances the current element to the next element */
        cursor = iter.Pos() /* get the current position (next elements) and assign it to
                            cursor */
        pdelete tmp; /* delete the previous to current element */

}
```

- **Wrapping Methodology**

Each type (*set, list, bag*) of the *Collection* utility contains instances of the corresponding classes as found in the schema definition.

When a set of <person> is defined, the address which points to the header of a set does not point specifically to the head element, but to a sort of 'wrapper' which envelops each of the object instances of type <person>, <course>, etc... making up the whole set.

The 'wrapping' job is done by using a specific class (_PerPtrWrap), needed when defining the address pointer of each type of the *Collection* utility:

```
Typedef _PerPtrWrap<person> person_wrap;
Typedef set<person_wrap> person_set;
```

Therefore, whenever a searching for a list of objects is needed in the *Collection* type, the 'match' method declared in the class _PerPtrWrap is needed, because objects are accessed through this 'wrapper' only.

## 4.3 University Data Model Creation

The steps needed to build the university data model, are summarized as follows:

- Define class and class templates whose objects are to be made persistent by using the form:

```
persistent class class-name;

persistent class person { ... };

template <...>
persistent class class-template-name;

template <class Type>
persistent class collection { ... };
```

- Create persistent objects using the operators pnew. This operator returns a pointer to persistent objects, known as *persistent* pointers:

```
persistent person *pe;   /* persistent pointers */

pe = pnew person;
```

  Volatile pointers refer to pointers declared without the persistent declaration:

```
class *cl;
```

  Both types of pointers are needed while building the constructors of the classes.

- *Collection* classes are needed for the data model definition:

```
Class Course {

    .......
```

```
        persistent course_set *prereq;
        persistent bag<int> *assess;
    }
```

- The declaration of persistent pointers having as type a *collection (set, bag or list)* of objects, need to use the *wrapper* class as defined before:

```
persistent person_set *sp;
(person_set is defined using the wrapper class)
```

- Need to use the multiple inheritance facility among classes, as is the case for the class *Tutor* which inherits from both *Student* and *Staff.*
- Need to use method overloading for salary, because the calculation of the salary of a tutor is different from that of a staff member.
- Get the values to be inserted, then create a persistent object having as arguments the values just entered, then call a method to insert the object into the database:

```
(
persistent person_set *sp;    /* person_set is defined using the class wrapper */
persistent person *pe;        /* persistent pointer */
const max = 20;

person *pz;                   /* volatile pointer */
getline(fname,max);           /* get first name */
getline(lname,max);           /* get last name */
getline(s, max);              /* get street */
getline(ct,max);              /* get city */

pe = pnew person (fname,lname,s,ct);     /* create persistent object */
pz = (person *) pe                       /*  assign to volatile object */
sp->add(pz);                             /* add the object into the person_set */
```

Iteration is used to add items into the collection in order to avoid duplicates as it is the case for sets and lists.

## 4.4 Problems

Some of the problems encountered during the implementation of collection classes can be defined as follows:

- Persistence: The declaration of pointers using the wrapper was not evident at first, but later after seeing some examples, it was more obvious.
  The difficulties were in matching elements of certain data type using the method match defined into the 'wrapper' class.
  Also I had problems assigning persistent to volatile pointers or vice-versa, but later I found the solution by reading some papers describing the topic.
- Passing arguments: Some difficulties were encountered while passing arguments to insert or find any object. I got the solution after applying the same procedure seen in some examples having the same situation.
- Template: By definition, templates are used to define classes with different type declaration. so no need to define one class per type. The problem I have, is when having templates in the method declaration and how to deal with them when creating items. I got the solution after checking the C++ templates (O++ being an extension of C++) and understood the real usage of them.

# Chapter 5

## 5 Conclusion

The original idea was to study and implement OCL, Object Comprehension Language, a new powerful query notation.

This idea initiated a team work of three interrelated projects. The main goal of the three projects was to test OCL, by allowing a user to run OCL queries against a certain database and view the results. Therefore, the required database to be built, should constitute one of the project components, the database project. Building a database system in turn requires a symbol table for such a database, which constitutes another component, the schema project. The third team has to provide the translation of the user specified OCL queries into an interface query language that could be recognized by and run against the database created in the database project.

The main objective of this thesis is to set up and manage the sample university database. The selected database is to be based on Ode, an experimental object-oriented database system from AT&T. The database is defined, queried and manipulated using O++, the database interface programming language based on the C++ programming language. The University model consists of interrelated objects of O++ classes such as Department, Course, Student, etc.

The O++ query results are obtained by the use of both Ode built-in facilities, in addition to utility classes implemented in O++ to support OCL concepts, such as set, bag, and list. The query results would be printed as a *collection* of solutions.

An overview of object-oriented databases is provided, comparing them to relational databases and how the latter did not seem to be adequate for certain applications such as multimedia technologies. The evolution of object-oriented database is also described.

Then, differences between object-oriented query languages and relational query languages are described. Ode, being an object-oriented database based on C++, is taken as a case study.

The power of Ode consists of facilities for creating and manipulating persistent objects, transaction processing, and utilities for querying a database. The Ode queries are based on O++ (an extension of C++).

A sample university model database is constructed, and utility classes are written to support the model and the OCL queries.

The problems encountered with O++ are as follows:

- The persistence issue: How to store persistent objects into the ODE database, and to remain there. The solution was to use the 'wrapper' class declaration (Appendix A).
- Recursion in class declaration: This situation occurs into the 'Course' class declaration of the university model database where one of the members is of type 'Course' considered as a *set* of prerequisite courses.
- Templates in O++: The rules applied to templates for O++, are the same as for C++, which were a new feature of C++ at the time. This required thoroughly learning C++.

On the other side, these are certain advantages to using Ode and O++ namely:

- The O++ based queries are easy to build, and user friendly, specially with the use of the utilities.
- The iteration method is used in searching the elements of a *collection* sequentially without exposing its internal structure.
- An abstract class *collection* contains all common methods (behavior) of the different types of the data structure utilities (set, bag, list).
- O++ being an extension of C++, many difficulties were resolved only by having the same solutions applied on C++ as well as O++.

The following project goals were achieved:

- Build the university sample database of interrelated objects of O++ classes.
- Build the utility classes in O++ to support OCL queries.
- Studied the evolution of object-oriented database systems compared to relational databases and how the latter are not adequate for certain applications.
- Studied Ode as a case study.

Finally, as this thesis is a part of all three projects, the completion of this part will eventually provide answers as collection of objects, needed by the translated (OCL to O++) queries of the translation component, which relies also on the symbol table of the schema project. It can be seen that all three projects were highly dependent on each other, which required the team members to be in synchronous delivering output and completing the work.

# Bibliography

[1] D. K. C. Chan and P. W. Trinder. Object comprehensions: A query notation for object-oriented databases. In *Proceedings of the 12th British National Conference on Databases*, Guildford, Springler-Verlag, July 1994.

[2] F. Bancilhon, C. Delobel, and P. Kanellakis. Building an Object-Oriented Database System. The Story of O2. Morgan Kaufmann Publishers, 1992.

[3] L. B. Bjornsson and L. Fisk. Object Database Programming Languages. Term paper, May 1997. Http://www.ecst.csuchico.edu/~leifur/doc/ODBMS.html.

[4] W. Kim. Introduction to Object-Oriented Databases. The MIT Press, 1991.

[5] R.M. Burstall, D.B. MacQueen, and D.T. Sanella. Hope: an experimental applicative language. In *In Proceedings of the 1st ACM Lisp Conference*, pages 136-143, ACM Press, 1980.

[6] D.A. Turner. Recursion equation as a programming language. In *Functional Programming and its Applications,*Cambridge University Press, 1981.

[7] B. S. Lee. Object-Oriented Databases: Systems and Standards. In *IEEE Computer Society*, pages 64-66, IEEE press, Oct. 1995.

[8] R. Agrawal, S. Dar and N. Gehani. The O++ Database Programming Language: Implementation and Experience. In *Proceedings IEEE International Conference on Data Engineering*, pages 61-70, Vienna, Austria, April 1993.

[9] A.M. Alashqur, S.Y.W. Su and H. Lam. OQL: A Query Language for Manipulating Object-Oriented Databases. *In Proceedings of the Fifteenth International Conference on Very Large Data Bases*, Amsterdam, Holland. August 1989.

[10] C. Roderic. Object Data Management. *Addison-Wesley Publishing Company Inc.*, Menlo Park, CA, 1994.

[11] M. Kemper. Object-Oriented Database Management. *Prentice Hall*, Englewood Cliffs, New Jersay, 1993.

[12] L. Liu. Odewalker User's Manual. In *AT&T Bell Labs Database Technical Reports.* 1989.

[13] R. Agrawal and N. Gehani. ODE (Object Database and Environment): The Language and the Data Model. In *Proc. ACM-SIGMOD 1989 Int'l Conf. Management of Data*, pages 36-45, May 1989.

[14] N. Gehani and H. Jagadish. ODE as an Active Database: Constraints and Triggers. In *Proc. 17th Int'l Conf. Very Large Data Bases,* pages 326-327, 1991.

[15] B. Alexandros and P. Euthimios. EOS: An Extensible Object Store. In *SIGMOD Conference*, page 517, 1994.

[16] S. Peyton-Jones. The implementation of functional programming languages. In Chapter 7, pages 127-138, *Prentice-Hall*, 1987.

# Appendix A : Collection Definition

## Name

*Collection* - base class for collection of objects.

## Description

The abstract class collection is used to manage a collection of objects. These objects are instances of classes derived from the class _node: It is basic class whose role is to create objects and a persistent link between them. provided the information inside the object and the link to the next object is available.

## Collection Types

The subclasses of **Collection** implement different ways to store and access objects. These different types ease to handle objects in an efficient and flexible manner.

The major three subclasses covered in this study are : **List**. **Bag** and **Set.**

The inheritance relationship among the different classes is as follows:

Class **Collection**

Class **List** : public **_Collection**

Class **Bag** : public **_Collection**

Class **Set** : public **_Collection**

## Enumerating Objects

Classes are always derived from **Collection.**

Class **Collection** is never used directly.

Class **Collection** is abstract.

Baseclasses: object

Subclasses: List, Bag, and Set.

The class **Collection** contains two instance variables declared as *protected*, and also several methods are declared as *public* to be called by the needed objects of different type.

**Instance Variables**

_node<Type> *head:

int total:

**Instance Method List**

| | |
|---|---|
| *Creator* | _Collection, list, bag, set |
| *Destructor* | ~_Collection |
| *Destruction* | Destroy |
| *Accessing* | Size<br>ret_head<br>findnb |
| *Manipulation* | Remove<br>Intersects<br>Intersection<br>Every<br>Union_Col<br>Atleast<br>Atmost<br>Just<br>Differ<br>Del<br>Insert<br>Append<br>Add<br>Union_set |
| *Debugging* | Display<br>Show |
| *Client Interface* | Member<br>Frequency<br>Isublist<br>Sublist |
| *Testing* | Is_empty |

**Name**

Collection::_collection - instance method

**Template**

_collection( )

**Specifiers**

public

**Description**

Collection creator: set the *head* pointer and the *total* counter to 0.

**Arguments**

none

**Return Argument**

none

**Categories**

creator

**First Definition**

class _collection

**Name**

Collection::~_collection - instance method

**Template**

~_collection( )

**Specifiers**

public

**Description**

Collection desctructor: Remove all nodes from the collection type.

**Arguments**

none

**Return Argument**

none

**Categories**

destructor

**First Definition**

class _collection

**Name**

Collection::size - instance method

**Template**

int size( )

**Specifiers**

public

**Description**

Get the total number of elements (nodes) into the collection.

**Arguments**

none

**Return Argument**

int

**Categories**

Accessing

**First Definition**

class _collection

**Name**

Collection::ret_head - instance method

**Template**

_node<type> * ret_head( )

**Specifiers**

public

**Description**

Return the node which is the head of the collection type.

**Arguments**

none

**Return Argument**

_node<Type>

**Categories**

Accessing

**First Definition**

class _collection

**Name**

Collection::member - instance method

**Template**

boolean member(const Type&)

**Specifiers**

public

**Description**

Search for a definit element of the collection.

**Arguments**

const Type&: value contained in the node to search for.

**Return Argument**

boolean values (T or F)

**Categories**

Client Interface

**First Definition**

class _collection

**Name**

Collection::remove - instance method

**Template**

void remove(const Type&)

**Specifiers**

public

**Description**

Search for a definit element of the collection and delete the corresponding node where the element in question resides.

**Arguments**

const Type&: value contained in the node to be removed.

**Return Argument**

void

**Categories**

Client Interface

**First Definition**

class _collection

**Name**

Collection::intersects - instance method

**Template**

boolean intersects(persistent _collection<Type>*)

**Specifiers**

public

**Description**

Check if any two collection of elements have at least one element in common.

**Arguments**

persistent _collection<Type>*: a persistent collection of elements in which the pointer to the header is passed.

**Return Argument**

Boolean

**Categories**

Manipulation

**First Definition**

class _collection

**Name**

Collection::intersection - instance method

**Template**

void intersection(persistent _collection<Type>*)

**Specifiers**

public

**Description**

Find the intersection between the current and _collection<Type> *, and put the solution into the current one by deleting the complementary elements found in the current collection.

**Arguments**

persistent _collection<Type>*: a persistent collection of elements in which the pointer to the header is passed.

**Return Argument**

Void

**Categories**

Manipulation

**First Definition**

class _collection

**Name**

Collection::every - instance method

**Template**

boolean every(persistent _collection<Type>*)

**Specifiers**

public

**Description**

If current set is a part of specified one (_collection<Type> *) then return true. otherwise return false.

**Arguments**

persistent _collection<Type>*: a persistent collection of elements in which the pointer to the header is passed.

**Return Argument**

boolean

**Categories**

Manipulation

**First Definition**

class _collection

**Name**

Collection::union_col - instance method

**Template**

void union_col(persistent _collection<Type>*)

**Specifiers**

public

**Description**

Add all elements of the specified collection (_collection<Type> *) to the current one.

**Arguments**

persistent _collection<Type>*: a persistent collection of elements in which the pointer to the header is passed.

**Return Argument**

void

**Categories**

Manipulation

**First Definition**

class _collection

**Name**

Collection::atleast - instance method

**Template**

boolean atleast(int al, persistent _collection<Type>*)

**Specifiers**

public

**Description**

If *atleast* (al) elements or more of current collection are part of the specified one (_collection<Type> *) then return true, otherwise return false.

**Arguments**

persistent _collection<Type>*: a persistent collection of elements in which the pointer to the header is passed.

**Return Argument**

boolean

**Categories**

Manipulation

**First Definition**

class _collection

**Name**

Collection::atmost - instance method

**Template**

boolean atmost(int am, persistent _collection<Type>*)

**Specifiers**

public

**Description**

If *atmost* (am) elements or less of current collection are part of the specified one (_collection<Type> *) then return true, otherwise return false.

**Arguments**

persistent _collection<Type>*: a persistent collection of elements in which the pointer to the header is passed.

**Return Argument**

boolean

**Categories**

Manipulation

**First Definition**

class _collection

**Name**

Collection::just - instance method

**Template**

boolean just(int js, persistent _collection<Type>*)

**Specifiers**

public

**Description**

If *just* (exactly) (js) elements of current collection are part of the specified one (_collection<Type> *) then return true, otherwise return false.

**Arguments**

persistent _collection<Type>* : a persistent collection of elements in which the pointer to the header is passed.

**Return Argument**

boolean

**Categories**

Manipulation

**First Definition**

class _collection

**Name**

Collection::differ - instance method

**Template**

void differ(persistent _collection<Type>*)

**Specifiers**

public

**Description**

Remove all elements of the specified collection (persistent_collection<Type>* ) from the current one if any.

**Arguments**

persistent _collection<Type>* : a persistent collection of elements in which the pointer to the header is passed.

**Return Argument**

void

**Categories**

Manipulation

**First Definition**

class _collection

**Name**

Collection::destroy - instance method

**Template**

void destroy( )

**Specifiers**

public

**Description**

Remove all elements (nodes) from the current collection.

**Arguments**

none.

**Return Argument**

void

**Categories**

Destruction

**First Definition**

class _collection

**Name**

Collection::display - instance method

**Template**

void display( )

**Specifiers**

public

**Description**

Display the complete elements of the current collection.

**Arguments**

none

**Return Argument**

void

**Categories**

Debugging

**First Definition**

class _collection

**Name**

List::*list* - instance method

**Template**

list( )

**Specifiers**

public

**Description**

Constructor type method: a *list* of type collection.

**Arguments**

none

**Return Argument**

none

**Categories**

Constructor

**First Definition**

class list

**Name**

List::is_empty - instance method

**Template**

boolean is_empty( )

**Specifiers**

public

**Description**

Return true if the current *list* is empty, i.e. total nb of elements is 0.

**Arguments**

none

**Return Argument**

boolean

**Categories**

Testing

**First Definition**

class list

**Name**

List::remove - instance method

**Template**

void remove(const Type &v)

**Specifiers**

public

**Description**

Find the position of the element v in the current *list* then remove the node corresponding to the current position.

**Arguments**

const Type &v: address value of the element to be deleted.

**Return Argument**

void

**Categories**

Manipulation

**First Definition**

class list

**Name**

List::del - instance method

**Template**

void del(int pos)

**Specifiers**

public

**Description**

Delete the element found at position *pos* in the current list.

**Arguments**

const Type &v: address value of the element to be deleted.

**Return Argument**

void

**Categories**

Manipulation

**First Definition**

class list

**Name**

List::findnb - instance method

**Template**

int findnb(const Type& val)

**Specifiers**

public

**Description**

Return the position of element *val* if it exists in the current list.

**Arguments**

const Type &val: address value of the element to detemine its position in the current *list.*

**Return Argument**

int

**Categories**

Accessing

**First Definition**

class list

**Name**

List::insert - instance method

**Template**

void insert(const Type& val,int pos)

**Specifiers**

public

**Description**

Insert a new node containing the element *val* between position *pos* -1 and *pos* of the current *list*. Special case, for a *list* of one element add the node at head, or append the new node to the tail in case *pos* points to the last node of the *list*.

**Arguments**

const Type &val: address value of the element to insert.
int pos: position where to insert.

**Return Argument**

void

**Categories**

Manipulation

**First Definition**

class list

**Name**

List::append - instance method

**Template**

void append(const Type& v)

**Specifiers**

public

**Description**

Append a new node containing the element *v* after the tail position of the *list.*

**Arguments**

const Type &v: address value of the element to append.

**Return Argument**

void

**Categories**

Manipulation

**First Definition**

class list

**Name**

List::frequency - instance method

**Template**

int frequency(const Type& val)

**Specifiers**

public

**Description**

Determine the number of occurrence of the value *val* into the current list.

**Arguments**

const Type &v: address value of the specified element whose number of occurrence must be found.

**Return Argument**

void

**Categories**

Client interface

**First Definition**

class list

**Name**

List::show - instance method

**Template**

Type show(int pos)

**Specifiers**

public

**Description**

Get the value of the element (node) at position *pos* of the current list.

**Arguments**

int pos: an integer value needed to get the content of the node at a certain position *pos* of the list.

**Return Argument**

Type (of the info)

**Categories**

Debugging

**First Definition**

class list

**Name**

List::isublist - instance method

**Template**

list<type>* isublist(int min,int max)

**Specifiers**

public

**Description**

Return a sublist of the current list limited by the above arguments.

**Arguments**

int min. max: two integer values used to delimiter a sublist of the current list. These numbers represent the position this newly sublist starts and ends within the current list.

**Return Argument**

list<Type>*: pointer to a newly created list.

**Categories**

Client interface

**First Definition**

class list

**Name**

List::sublist - instance method

**Template**

list<type>* sublist(const Type& j, const Type &k)

**Specifiers**

public

**Description**

Return a sublist of the current list limited by the *positions* those above data arguments are found into the current list.

**Arguments**

const Type &j , &k: value of two elements which should be checked if they exist into the current list. If this is the case. the corresponding position of these elements is determined, by which the newly created sublist is limited.

**Return Argument**

list<Type>*: pointer to a newly created list.

**Categories**

Client interface

**First Definition**

class list

**Name**

bag::bag - instance method

**Template**

bag( )

**Specifiers**

public

**Description**

Constructor type method: a *bag* of type collection.

**Arguments**

none

**Return Argument**

none

**Categories**

Constructor

**First Definition**

class bag

**Name**

bag::add - instance method

**Template**

void add(const Type& info)

**Specifiers**

public

**Description**

Append a new element *info* into the bag, by creating a new node with *info* content linked to the last node of the current bag.

**Arguments**

const Type &info: value of the element to be added to the current bag.

**Return Argument**

void

**Categories**

Manipulation

**First Definition**

class bag

**Name**

bag::frequency - instance method

**Template**

int frequency(const Type& info)

**Specifiers**

public

**Description**

Determine the number of times the value *info* appeared into the current bag, including the duplicate values.

**Arguments**

const Type &info: value of the element to be checked for times of occurrence.

**Return Argument**

int

**Categories**

Client interface

**First Definition**

class bag

**Name**

set::set - instance method

**Template**

set( )

**Specifiers**

public

**Description**

Constructor type method: a *set* of type collection.

**Arguments**

none

**Return Argument**

none

**Categories**

Constructor

**First Definition**

class set

**Name**

set::add - instance method

**Template**

void add(const Type& vl)

**Specifiers**

public

**Description**

Append a new element *vl* into the bag, by creating a new node with *vl* content linked to the head node of the current set, after checking that no duplicate *vl* values already exist in the current set.

**Arguments**

const Type &info: value of the element to be added to the current set.

**Return Argument**

void

**Categories**

Manipulation

**First Definition**

class set

**Name**

set::union_set - instance method

**Template**

void union_set(set <type>* st)

**Specifiers**

public

**Description**

Get the union of the current and specified set *st* by appending to the current set new elements which do exist into the specified set but not into the current one.

**Arguments**

set <type> * st: pointer to a specified set to make union with.

**Return Argument**

void

**Categories**

Manipulation

**First Definition**

class set

# Appendix B : Header Files *.h

```
typedef _PersPtrWrap<person> person_wrap;
typedef set<person_wrap> person_set;

typedef _PersPtrWrap<department> depart_wrap;
typedef set<depart_wrap> depart_set;

typedef _PersPtrWrap<staff> staff_wrap;
typedef set<staff_wrap> staff_set;

typedef _PersPtrWrap<student> student_wrap;
typedef set<student_wrap> student_set;

typedef _PersPtrWrap<tutor> tutor_wrap;
typedef set<tutor_wrap> tutor_set;
```

# Appendix C : University Data Model

```
#define Max 20

persistent

class address {

private:

        indexable char street [Max];
        char city[Max];

public:

        address(char*, char*);
        char* ret_street( ) {return street;}
        char* ret_city( ) {return city;}

};

address::address(char *s, char *c)

{
        strcpy(street, s);
        strcpy(city, c);

};
```

```
persistent

class person {

private:

        indexable char first_name [Max];
        indexable char last_name [Max];
        persistent address *per_addr;

public:

        person (char*, char*. char*, char*);
        char* get_name( );
        char* get_address( );
        int isequal(person*);
        void print_item( ) {cout << get_name( );}

};


person::person(char* fname. char* lname. char* s. char* c )

{
        strcpy(first_name,fname);
        strcpy(last_name. lname);
        per_addr = pnew address(s,c);
};

char* person::get_name( )

{
        char temp[50];
        char* tpname;
        tpname = new char[strlen(temp) + 1];
        strcpy(tpname, first_name);
        strcat(tpname, " ");
        strcat(tpname, last_name);

        return tpname;

};
```

```
char* person::get_address( )

{

        char temp[50];
        char *tpaddr;

        tpaddr = new char[strlen(temp) + 1];
        strcpy(tpaddr, per_addr->ret_street( ));
        strcat(tpaddr, " ");
        strcat(tpaddr, per_addr->ret_city( ));
        return tpaddr;

};




int person::isequal(person *eq)

{

        return (!strcmp(first_name.eq->first_name) &&
                !strcmp(last_name, eq->last_name)) ? 1 : 0;

};
```

```
persistent

class department{

private:

        indexable char dept_name[Max];
        persistent address *dept_addr;

public:

        department (char*. char*, char*);
        char *dep_name ( ) {return dept_name;}
        char *dep_address ( );
        int isequal (department*);
        void print_item( ) {cout << dep_name( );}

};

department::department (char* dname. char* s, char* s)

{

        strcpy (dept_name. dname);
        dept_addr = pnew address (s,c);

}:


char* department::dep_address( )

{

        char temp[50];
        char* tpaddr;

        tpaddr = new char(strlen(temp) + 1];
        strcpy(tpaddr, dept_addr->ret_street( ));
        strcat(tpaddr, " ");
        strcat(tpaddr, dept_addr->ret_city( ));

        return tpaddr;

};
```

```
int department::isequal(department *eq)

{

        return (!strcmp(dept_name.eq->dept_name)) ? 1 : 0;

};


persistent

class student: public person {

private:

        indexable department* major;
        persistent staff_list* supervised_by;
        persistent course_set* takes;

public:

        student(department*, staff_list*, course_set*)
        department*  get_major( ) {return major;}
        staff_list* get_superv( ) {return supervised_by;}
        course_set* get_courses( ) {return takes;}

};

student::student(department* depart. staff_list* stfl. course_set* crset)
{
        major = depart;
        stfl = supervised_by;
        takes = crset;

};
```

```
Persistent

class course {

private:

        indexable char code[Max];
        persistent depart_set *prunby;
        persistent course_set *prereq;
        persistent bag<int> *assess;
        int credits;

public:

        course (char*, depart_set*, course_set*, int)
        depart_set *get_runby( ) {return prunby;}
        course_set *get_prereq( ) {return prereq;}
        bag<int> *get_assess( ) {return assess;}
        char * get_course( ) {return code;}
        int get_credits( ) {return credits;}

}

course::course (char* name, depart_set* depset, course_set* crset, int nbcredits)

{

        strcpy(code,name);
        credits = nbcredits;
        prunby = depset;
        prereq = crset;
        assess = pnew bag( );

}
```

```
persistent

class tutor: public staff, student {

private:

        indexable int salary;

public:

        tutor(int)
        int get_salary ( ) {return salary;}

};



tutor::tutor(int sal)

{
        salary = sal;

};


Persistent

class VisitingStaff: public Staff {

};
```

```
#include <ode.h>

enum Boolean {false,true};

template<class Type>

persistent class _node {

private:

        Type info;
        persistent _node<Type> *next;

public:

        _node<Type> (const Type& v, persistent _node<Type>* n): info(v), next(n) {}
        void put_next (persistent _node<Type>* n) {next = n;}
        persistent _node<Type> *get_next( ) {return next;}
        int search (const Type& t);
        Type get_info( ) {return info;}

};
```

# Appendix D : Utility Code

```
template <class Type>
persistent class _collection {

protected:
  persistent _node<Type> *head;
  int  total;

public:

 _collection() {total = 0; head = 0;}
 ~_collection() {destroy();}
 int size() {return total;}
 _node<Type> *ret_head() {return (_node<Type>*)(void*) head;}

 virtual Boolean member(const Type&);
 virtual void remove(const Type&);

 virtual Boolean intersects(persistent _collection<Type>*);
 virtual void intersection(persistent _collection<Type>*);
 virtual Boolean every(persistent _collection<Type>*);
 virtual void union_col(persistent _collection<Type>*);
 virtual Boolean atleast(int,persistent _collection<Type>*);
 virtual Boolean atmost(int, persistent _collection<Type>*);
 virtual Boolean just(int,persistent _collection<Type>*);
 virtual void differ(persistent _collection<Type>*);

 virtual void destroy(); // remove all items

 void display();

};

/* ====================================================================*/

template<class Type>
class _PersPtrWrap {
private:
  persistent Type *info;
public:
   persistent Type* wrap_info() {return info;}
  _PersPtrWrap(persistent Type *inf) : info(inf) { }
  _PersPtrWrap() : info(NULL) { }
  _PersPtrWrap(const _PersPtrWrap<Type>& wrap)
         {info = wrap.info;}
  int match (const _PersPtrWrap<Type> &wrap) {return ((Type*)
(void*)info)->isequal((Type*)(void*)wrap.info);}
  void print() const {
     ((Type*)(void*)info)->print_item();
  }
};

/* ================================================================ */

template <class Type>
class _iter_list {
private:
  persistent _node<Type> *entry;
  persistent _node<Type> *pos;
public:
  _iter_list(_collection<Type> *lst) {
      entry = lst->head;
      pos   = entry;
    }
  persistent _node<Type> *POS() {return pos;}
  void first() {pos = entry;}
  void next() {if (pos) pos=POS()->next;}
  Type *current(){
   Type* t= NULL;
   if (pos) t=&(POS()->info);
     return t;
    }

};
/*================================================================ */


template <class Type>
persistent class list : public _collection<Type> {

public:

 list() : _collection<Type>() {};
 Boolean is_empty();
 void remove(const Type &v) {del(findnb(v));}
 void del(int);
 int findnb(const Type&);
```

```
  void insert(const Type&, int);
  void append(const Type& v) {insert(v, total+1);}
  int  frequency(const Type&);
  Type show(int);
  list<Type>* isublist(int,int);
  list<Type>* sublist(const Type& j, const Type& k) {return isublist(findnb(j),
findnb(k));}
};

/* ========================================================================== */


template <class Type>
persistent class bag : public _collection<Type> {

public:
  bag() : _collection<Type>() {};

  void add(const Type&);

  int frequency(const Type&);

};

/*=========================================================================== */


template <class Type>
persistent class set : public _collection<Type> {

public:

  set() : _collection<Type>() {}

  void add(const Type& vl);

  void union_set(set<Type>*);


};

/*=========================================================================== */

template<class Type>
int _node<Type>::search(const Type& t)
```

```
{
persistent _node<Type>* cursor = pthis;


 while (cursor){

   if (cursor->info.match(t))
      return  1;
   else
       cursor = cursor->get_next();

 }

 return 0;

};

/*================================================================= */

template<class Type>
Boolean list<Type>::is_empty()
{
 return total == 0 ? true : false;
};

/*================================================================= */

template<class Type>
Boolean _collection<Type>::member(const Type& val)
{
if (!head)
  return false;

 persistent _node<Type> *cursor = head;

 if (cursor->search(val))
   return true;

 return false;

};

/*================================================================= */

template<class Type>
```

```
void _collection<Type>::destroy()
{
 persistent _node<Type>* tmp;
 persistent _node<Type>* cursor;

 _iter_list<Type> iter(this);
 iter.first();
 cursor = iter.POS():

 while(cursor) {

    tmp = iter.POS();
    iter.next():
    cursor = iter.POS():
    pdelete tmp;
 }
 cout << "SET DESTROYED!!" << endl:
 head = 0;
 total = 0;
};

/*================================================================= */

template<class Type>
void list<Type>::del(int pos)
{
  persistent _node<Type> *prev, *cursor = head;

 if (pos == 1) {
   head = cursor->get_next();
   pdelete cursor;
   total--;
   if (total == 0)
      head = 0;
  }
 else
    if (pos > 1 && pos <= total)
    {
    for(int i = 1; (i < pos) ; i++) {
      prev = cursor;
      cursor = cursor->get_next();
    }
    prev->put_next(cursor->get_next());
    pdelete cursor;
    total--;
   }
};

/*=================================================================== */

template<class Type>
void list<Type>::insert(const Type& val, int pos)
{
 persistent _node<Type> *prev, *cursor = head, *temp = pnew _node<Type>(val);
 /*  check above */

 if (!head) {
    head = temp;
    total++;
    }
 else
    if (pos == 1) { // insert at head
       temp->put_next(head);
       head = temp;
       total++;
    }
    else
        if (pos > 1)
        {
        for(int i = 1; (i < pos) && cursor->get_next(); i++) {
          prev = cursor;
          cursor = cursor->get_next();
        }

        if (!cursor->get_next() && (i < pos))
          cursor->put_next(temp);
        else {
             temp->put_next(cursor);
             prev->put_next(temp);
           }
        total++;
        }
};

/*=================================================================== */


template<class Type>
int list<Type>::findnb(const Type& val)
```

```
{
 int cnt = 0;

 if (!head)
   return cnt;

 persistent _node<Type> *cursor = head;

 while(cursor) {
  cnt++;
  if (cursor->info.match(val))
     return cnt;
  cursor = cursor->get_next();
 }

 return 0;
};

/*=================================================================== */

template<class Type>
int list<Type>::frequency(const Type& val)
{
  persistent _node<Type> *cursor = head;

  int count = 0;

 if (!head)
    cout << "List is empty!" << endl;
 else {
  while (cursor) {
    if (cursor->info.match(val))
           count++;
    cursor = cursor->get_next();
     }
  }
  return count;
};

/*=================================================================== */

template<class Type>
Type list<Type>::show(int pos)
{
```

```
persistent _node<Type> *cursor = head:


if((pos <= total) && (cursor))
{
 for (int i=1; i < pos;i++)
    cursor = cursor->get_next();

 return(cursor->get_info());
}

return 0:

};

/*=====================================================================*/


template<class Type>
list<Type>* list<Type>::isublist(int min, int max)
{

 list<Type> *lst:

 lst = new list<Type>:

  persistent _node<Type> *cursor = head:


 if((!head) || (min > max))
   return lst:

 for (int i=1; (i <= max) && cursor: i++){
    if((i >= min) && (i <= max))
      lst->append(cursor->get_info());
    cursor = cursor->get_next();
   }
 return lst;

};

/*=====================================================================*/

template <class Type>
void _collection<Type>::intersection(persistent _collection<Type> *lst)
```

```
{
  persistent _node<Type> *crs1 = head, *temp, *prev = head; // current one
  persistent _node<Type> *crs2 = lst->head;

  int flag = 1;

 while(crs1) {
  if (crs2->search(crs1->info.wrap_info()))
   flag = 0;

  if (flag){
     prev->put_next(crs1->get_next());
     temp = crs1;
     crs1 = crs1->get_next();
     if (temp == head)
        head = crs1;
     pdelete temp;
     total--;
     if (total == 0)
       head = 0;
     }
  else
     {
     prev = crs1;
     crs1 = crs1->get_next();
     }

  crs2 = lst->head;
  flag = 1;
 }

};

/*==================================================================== */

template <class Type>
Boolean _collection<Type>::intersects(persistent _collection<Type> *lst)
{
  persistent _node<Type> *cursor1 = head;
  persistent _node<Type> *crs2 = lst->head;

 while(cursor1) {
  if (crs2->search(cursor1->info.wrap_info()))
     return true;
  cursor1 = cursor1->get_next();
  crs2 = lst->head:
 }
return false:

}:

/*=================================================================== */

template <class Type>
void _collection<Type>::differ(persistent _collection<Type> *lst)
{
  persistent _node<Type> *crs1 = head, *temp, *prev = head; // current one
  persistent _node<Type> *crs2 = lst->head;

  int flag = 1;

 while(crs1) {
 if (crs2->search(crs1->info.wrap_info()))
   flag = 0:

 if (!flag){
     prev->put_next(crs1->get_next());
     temp = crs1;
     crs1 = crs1->get_next();
     if (temp == head && crs1)
        head = crs1:
     pdelete temp;
     total--;
     if (total == 0)
       head = 0;
     }
 else
    {
     prev = crs1;
     crs1 = crs1->get_next():
    }

 crs2 = lst->head;
 flag = 1;
 }

};

/*=================================================================== */
```

```
template <class Type>
void _collection<Type>::union_col(persistent _collection<Type> *lst)
{
  persistent _node<Type> *cursor1 = head;
  persistent _node<Type> *cursor2 = lst->head;

 while(cursor1->get_next())
   cursor1 = cursor1->get_next();

 cursor1->put_next(cursor2);
 total = total + lst->total;

};

/*================================================================== */

template <class Type>
void set<Type>::union_set(set<Type> *st)
{
  persistent _node<Type> *cursor1 = head, *prev = head;
  persistent _node<Type> *crs2 = st->head;

  int flag = 1;

 while(crs2) {

  while(cursor1 && flag) {
  if (cursor1->info.match(crs2->info.wrap_info()))
      flag = 0;
   else
      prev = cursor1;
      cursor1 = cursor1->get_next();

 }

  if (flag) {
     prev->put_next(crs2);
     total++;
  }

  cursor1 = head;
  prev = head;
  crs2 = crs2->get_next();
  flag = 1;
}

};

/*===================================================================================*/

template <class Type>
Boolean _collection<Type>::every(persistent _collection<Type> *lst)
{
  persistent _node<Type> *cursor1 = head;
  persistent _node<Type> *crs2 = lst->head;

  int flag = 1;

 while(cursor1) {
  if (crs2->search(cursor1->info.wrap_info()))
    flag = 0;
  if (flag)
    return false;
  else {
    cursor1 = cursor1->get_next();
    crs2 = lst->head;
    flag = 1;
 }

}

return true;

};

/*===================================================================================*/

template <class Type>
Boolean _collection<Type>::atleast(int al, persistent _collection<Type> *lst)
{

  persistent _node<Type> *cursor1 = head;
  persistent _node<Type> *crs2 = lst->head;

  int flag = 1;
  int cnt  = 0;
```

```
while(cursor1) {
  if (crs2->search(cursor1->info.wrap_info())) {
    cnt++;
    flag = 0;
  }

  cursor1 = cursor1->get_next();
  crs2 = lst->head;
  flag = 1;
 }

if (cnt >= al)
  return true;
else
  return false;

};


/*========================================================================= */

template <class Type>
Boolean _collection<Type>::atmost(int am, persistent _collection<Type> *lst)
{

  persistent _node<Type> *cursor1 = head;
  persistent _node<Type> *crs2 = lst->head;

  int flag = 1;
  int cnt  = 0;

while(cursor1) {
  if (crs2->search(cursor1->info.wrap_info())) {
    cnt++;
    flag = 0;
  }

  cursor1 = cursor1->get_next();
  crs2 = lst->head;
  flag = 1;
 }
```

```
if (cnt <= am)
  return true;
else
  return false;

};


/*================================================================================ */

template <class Type>
Boolean _collection<Type>::just(int js, persistent _collection<Type> *lst)
{

   persistent _node<Type> *cursor1 = head;
   persistent _node<Type> *crs2 = lst->head;

   int flag = 1;
   int cnt  = 0;

while(cursor1) {
   if (crs2->search(cursor1->info.wrap_info())) {
     cnt++;
     flag = 0;
   }

   cursor1 = cursor1->get_next();
   crs2 = lst->head;
   flag = 1;
 }

if (cnt == js)
  return true;
else
  return false;

};

/*================================================================================ */

template <class Type>
void _collection<Type>::display()
{
   _iter_list<Type> iter(this);
   iter.first();
   Type* t;

   for (; t=iter.current();iter.next()) {
      cout << "*";
      t->print(); cout << endl;
    }

};


/*============================================================================= */

template <class Type>
void _collection<Type>::remove(const Type& info)
{
 persistent _node<Type> *prev = head, *cursor = head, *temp;

 int flag = 1;


 while (cursor && flag) {
   if (!(cursor->search(info))){
      prev = cursor;
      cursor = cursor->get_next();
     }
   else
     {
      prev->put_next(cursor->get_next());
      temp = cursor;
      cursor = cursor->get_next();
      if (temp == head && cursor)
         head = cursor;
      pdelete temp;
      total--;
      if (total == 0)
        head = 0;
      flag = 0;
     }
 }
};


/*============================================================================= */
```

```
template <class Type>
void bag<Type>::add(const Type &info)
{
  persistent _node<Type> *cursor = head, *crs = head;

  persistent _node<Type> *temp = pnew _node<Type>(info);

 if (!head) {
    head = temp;
 }
 else
    {
     while (cursor->get_next()) {
          cursor = cursor->get_next();
     }
     cursor->put_next(temp);

    }
 total++;
};

/*================================================================*/

template <class Type>
int bag<Type>::frequency(const Type& info)
{
 persistent _node<Type> *cursor = head;

 int count = 0;

 if (!head)
    cout << "List is empty!" << endl;
 else
  {
     while (cursor)
     {
        if (cursor->info.match(info))
           count++;
     cursor = cursor->get_next();
     }
  }
  return count;
};

/*================================================================*/
```

```
template <class Type>
void set<Type>::add(const Type& vl)
{
  persistent _node<Type>* cursor = head;

   if (!cursor->search(vl))
     {
     head = pnew _node<Type>(vl,head);
     total++;
     }

};

/*===================================================================== */
```