PERSISTENCIA, EVOLUCIÓN DEL ESQUEMA Y RENDIMIENTO EN EL MODELO BASADO EN CONTENEDORES

(*Persistence, Schema Evolution and Performance in the Container-based Model*)

J. Baltasar García Perez-Schofield

TESIS DOCTORAL

presentada para optar al grado de Doctor en Ingeniería Informática

"What do you need persistence for, provided you have memory-mapped files?". *Anonymous referee.*

Noviembre 2002

Área de Lenguajes y Sistemas Informáticos

Departamento de Informática

Universidad de Vigo

Directores de la presente Tesis Doctoral

Supervisors of this PhD thesis

Dr. Manuel Pérez Cota
Department of Informatics
University of Vigo
España (Spain)

Dr. Timothy B. Cooper
Director of Smarts, Pty
Sydney, Australia

A mis padres, cuyo aliento y apoyo incondicional han sido constantes durante toda mi vida.

A mi familia.

Agradecimientos

El simple hecho de comenzar a escribir esta sección es un acontecimiento difícil, cargado de recuerdos y sensaciones contradictorias, por muchas razones. En primer lugar, está ese detalle de ser una "sección" de la tesis, que se coloca al principio pero que inevitablemente se escribe al final. Esa "sección", en la que uno no puede apoyarse en resultados anteriores, ni en la bibliografía disponible, y donde no es necesario probar nada. Y sobre todo, está esa sensación de terminar este trabajo, ponerle el punto final, lo que lleva inevitablemente a pensar en el principio del camino, y en el mismo camino recorrido, una investigación que culmina ... en la escritura de esta página.

En primer lugar, deseo agradecerle a Tim Cooper todo lo que ha hecho y todo su interés por este trabajo. Sencillamente, él es el padre de Barbados y el que además aceptó dirigir el trabajo del autor de esta Tesis Doctoral (a pesar de ser también el director de Smarts, Pty). No puedo evitar una sonrisa pensando en cómo, a través del correo electrónico, trabamos relación, y más tarde, comenzamos la andadura como director y estudiante de doctorado. No deja de ser curioso que una persona en España le pida a otra en Australia que le codirija la Tesis Doctoral por correo electrónico, ¡aunque creo que es aún más curioso que él aceptara!. Una estancia de investigación en Australia era inevitable, y durante la misma, además de trabajar con Tim, conocí a otra gente muy interesante, que hicieron el tiempo allí mucho más agradable, como Kieran, Audris y Andi. También quiero agradecer la ayuda de Gernot Heiser (otra "cita" por correo electrónico), de la Universidad de Nueva Gales del Sur, quien escuchó amablemente el desarrollo de Barbados y me ofreció interesantes sugerencias.

A Manuel Pérez Cota, el director de esta tesis, le agradezco su constante apoyo, su plena confianza en mi trabajo y la libertad de la que disfruté realizando el mismo.

El trabajo duro, siempre es mucho más llevadero si se comparte con otros, presentes en esas anécdotas, en las alegrías y presentes sobre todo cuando tienes esos momentos bajos en los que piensas que nunca vas a terminar, que el prototipo nunca acabará de funcionar bien, o que la burocracia va a acabar contigo. Ésos otros son mis compañeros de trabajo, y muy especialmente, los del tercer piso. Vaya un agradecimiento especial a Lourdes y a María, por sus ánimos y esas conversaciones "seminocturnas" alrededor de una caja de *donettes*, a Emilio, por su apoyo y amistad, a Eva y a Arno, por esos cafés tan agradables, a Jacinto, por esos favores de última hora, a Orge, por tan brillante portada, y ... a todos mis compañeros en general.

Finalmente, a Joaquín le agradezco esas conversaciones de madrugada en La Coruña, en la que hablábamos de tantas cosas, incluyendo la Tesis Doctoral. A él le debo varias ideas sobre cómo comenzar un trabajo de investigación.

Está hecho.

Orense, 27 de Septiembre de 2002

Abstract

The field of Persistent Systems has been one of the big open areas of research since 1970, with the appearance of the first object-oriented database management systems. For many reasons (performance, working philosophy ... etc.) persistent systems have not become commonplace.

The first part of this PhD thesis will present a study of existing models of persistence and existing persistent systems. A new model of persistence, based on previous ones and called the "container-based model" will be developed. This model is contrasted with the orthogonal model of persistence (currently the most popular model), and has important benefits in performance.

The container-based model will be applied to the Barbados prototype, a C++-based persistent programming environment. This prototype will serve as a performance benchmark in order to verify that the assumptions made in our model are correct and valid.

Finally, Schema Evolution presents one of the most important sub-problems in this field of research. A complete design of Schema Evolution for the container-based model will be included, which therefore would be directly implementable in the prototype.

Resumen

Los sistemas persistentes se configuran como uno de los grandes campos de investigación desde 1970, con la venida de las bases de datos orientadas a objetos. Varias razones (rendimiento, filosofía de trabajo ... etc) han impedido que estos sistemas sustituyan a los sistemas actuales.

En esta tesis, se presentará en primer lugar un estudio sobre los modelos de persistencia y los sistemas persistentes ya existentes. Se desarrollará un nuevo modelo de persistencia, basado en los anteriores, llamado el modelo de persistencia basado en contenedores. Este modelo se contrastará con el modelo de persistencia ortogonal (el imperante actualmente), y se demostrará como el primero conlleva significativas mejoras en el rendimiento sobre el segundo.

El modelo basado en contenedores se aplicará al prototipo Barbados, un sistema de programación persistente basado en C++, del que se obtendrá una evaluación del rendimiento real, comprobando si las asunciones del modelo son válidas y correctas.

Finalmente, la evolución del esquema presenta uno de las subáreas más importantes en este campo. Se incluirá un diseño completo de la evolución del esquema aplicable al modelo de contenedores, y por tanto, aplicable directamente al prototipo.

Contents

Detailed Index

Chapter 1: Introduction

1 Introduction

Persistence is a term which has been widely used from the 1970s (with the advent of the Object Oriented Database Managers) until today (Brown, 1991). Even today, it is many times confused with simple, raw serialisation to a file; serialisation could be therefore understood as the main outcome from persistence for current systems.

The objective of persistence is to free the programmer or user of the extra work which supposes to save the data the process is using at the end of its execution and restore it again in the next execution; this task is widely accepted as an error-prone task. With persistence, this task is automated. This way, the programmer or user has all the data the process needs available as soon as the process is in memory. In many persistent systems, this data is available for other processes, and not only for the one which created it; this makes many persistent system to work somehow as an object oriented database manager (although many of them don't support object oriented features such as transactions).

Although the field of persistence has had a lot of life as a scientific research field in computer science, it has been practically closed some time ago. The basic problems of persistence are: (a) the deep change in working philosophy for programmers, no more founded on files, (b) the strictness which comes with any automaton, which supposes added problems in practice, such as the one known as schema evolution, and (c) the performance problems which derive from using this kind of systems; persistent systems have failed in offering a performance similar to traditional systems. Another, secondary, problem is the inexistence (at least, at the beginning of research in this

field) of efforts trying to approach this kind of systems to the final user, for example supporting widely-known programming languages in persistent programming systems.

Although a study of the reasons of the decline of interest of research in persistence is out of the scope of this PhD thesis, part of this work will be to present a persistent programming system, Barbados, which has many characteristics that try to cope with that problems; a commonly used language such as C++ is supported, which is expected could attract many programmers to use a final product based in this prototype. A design for schema evolution support is also presented in detail, too; this is known as one of the big trials of research in persistence. Finally, the theoretical model which Barbados follows is expected to obtain several advantages from the merging of the characteristics of a file system and a persistent storage.

The justification of this PhD is that a) it presents a new model of persistence, different of the widely used one, the orthogonal model of persistence; b) it presents a new, novel mechanism of schema evolution which takes advantage of the model of containers; and c) it provides the reader with a set of performance tests on the prototype, which make possible to him or her to evaluate whether this work has achieved the expected results.

This PhD thesis is organised as follows: firstly, the container-based model of persistence is presented in chapter 2, followed by a thorough comparison with the state of the art in chapter 3. The chapter 4 is dedicated exclusively to the matter of the design of schema evolution in the container-based model. The chapter 5 presents valuable performance results, and finally, the chapter 6 contents all conclusions the author has obtained during the development of this research work.

Chapter 2: The Container-based Model of Persistence

2 The Container-based Model of Persistence

2.1 Introduction

The theoretical container-based model of persistence is the basis of this work, as this model provides the framework which allowed us to implement Barbados, with its characteristics.

In this section, the container-based model is presented briefly in a typical session with the prototype; secondly in its theoretical aspects, and finally, in its implementation aspects in the Barbados prototype.

2.2 A practical introduction to the model

The ideas in this chapter related to containers are implemented in a system called Barbados. Barbados is a persistent programming system which relies on an object oriented layered architecture (Álvarez Gutiérrez *et al.*, 1999). This layered arquitecture is implemented on the top of the Windows API (Win32)[1] but could run as a stand-alone operating system (since the goal of persistence is effectively the merging of the operating system with the programming language, Barbados is actually a full C++ (Ellis & Stroustrup, 1990) programming environment completed

[1] Copyright Microsoft Intl. From (Microsoft, 1998a).

with editor, compiler and debugger). Barbados offers a C++ command-line interface which provides incremental and interactive compilation and thereby C++ acts as a shell-level language as well as a programming language. Barbados can be therefore described as similar to UNIX but where instead of operating inside a directory hierarchy of files, users operate inside a directory hierarchy of C++ objects.

Barbados will be used in this section to show the working philosophy of a persistent programming system. The application built with the prototype will be a tiny example about a **Car Shop**. The answers of the system will be shown in inverse video.

```
cd(/);
mkdir(carshop);
carshop: directory
cd(carshop);
mkdir(bin);
bin:directory
mkdir(data);
data: directory
cd(data);
/Common/Containers/listOfPointers people;
/Common/Containers/listOfPointers cars;
```

The user has created here the container which will be holding the application: the '*bin*' and the '*data*' directory under the '*carshop*' directory. The '*bin*' container will hold the application code and class definitions. The application data will be in the '*data*' container. Two lists are created there, in order to hold pointers to instances of cars and clients. The class '`listOfPointers`' is part of the standard library in Barbados.

```
cd(..);
cd(bin);
#define MAX 150

class Item {
public:
          virtual char* getDescription(void) = 0;
          virtual char* getKey(void) = 0;
};
class Item {}
```

The class `Item` is the top class of the hierarchy. It will be the parent class of the two main classes of the system: `person` and `car`.

```
class person : public Item {
          private:
                    char name[MAX];
                    char surname[MAX];
                    char address[MAX];
                    char buffer[MAX];
          public:
                    char *getName()    { return name; };
                    char *getSurname() { return surname; };
                    char *getAddress() { return address; };
```

```
                    person(char *surname, char* name, char *address)
                    {
                              strcpy(this->surname, surname);
                              strcpy(this->name, name);
                              strcpy(this->address, address);
                    }

                    char *getDescription() {
                              sprintf(buffer, "%s, %s. %s.", surname, name,
                                        address);

                              return buffer;
                    }

                    char *getKey() { return getSurname(); }
};
class person {}
```

The class *person* will represent any client or potential client of the second-hand car shop.

```
class car : public Item {
          private:
                    char plate[MAX];
                    char color[MAX];
                    int year;
                    char buffer[MAX];
                    person *boughtby;
                    int price;
          public:
                    car()
                    { *plate = 0; year = 0; boughtby = NULL; }
                    car(char *plate, int year)
                    {
                           this->year = year; strcpy(this->plate, plate);
                           boughtby = NULL;
                    }

                    bool isAvailable() { return (boughtby == NULL); }
                    person *getBuyer() { return boughtby; }
                    void buyCar(person *p, int price) {
                              if (isAvailable())
                              {
                                        boughtby = p; this->price = price;
                              }
                    }

                    int getPrice() { return price; }
                    void modifyPrice(int price) { if (isAvailable())
                                                  this->price = price; }

                    char *getPlate() { return plate; }
                    char *getColor() { return color; }
                    char *setColor(char *color) { return
```

```
                                strcpy(this->color, color); }
                int getYear()    { return year; }
                char *getDescription() {
                        if (year==0)
                                return NULL;
                        else {
                                sprintf(buffer,
                        "Car %s: Year: %d Color:'%s'"
                        " Price:$%d\n",
                         plate, year, color, price);
                                if (!isAvailable())
                                {
                                        strcat(buffer,
                                                "\tbought by ");
                                        strcat(buffer,
                                      getBuyer()->getDescription());
                                }

                                return buffer;
                        }
                }
                char *getKey() { return getPlate(); }
};
class car {};
```

The class *car* represents all cars for sale in the second-hand car shop. It stores a pointer to the client that bought it. If the pointer is NULL (for example, when the object is built), then it has not been sold. The `isAvailable`() method determines (comparing the pointer with NULL) whether the car has been sold or not. Also, there is a method, `buyCar`(), that accepts a pointer to a `person` and a *price*, and sets the values appropriately. After the car has been sold, it is not possible to modify the price.

The user has, until now, created the set of classes needed in this example. The next step is to create the set of functions that will use that classes in order to provide the management functionalities for the second-hand **Car Shop**.

As has been seen, the user doesn't need to have any special knowledge about Barbados, apart from the extensions for directory management. The rest is plain C++.

```
void createCar(void)
{
        int year;
        char buf[MAX];
        char plate[MAX];
        char color[MAX];
        int price;
        car *newcar;


        // Ask for data
        cout << "Please enter a plate: " << endl;
        gets(plate);

        cout << "Please enter year of made: " << endl;
```

```
            gets(buf);
            sscanf(buf, "%d", &year);

            cout << "Please enter the color of the car: " << endl;
            gets(color);

            cout << "Please set initial price: " << endl;
            gets(buf);
            sscanf(buf, "%d", &price);

            // Create a car
            newcar = new car(plate, year);
            newcar->setColor(color);
            newcar->modifyPrice(price);

            // Store it
            /carshop/data/cars.appendItem(newcar);

      cout << "End of creation." << endl;
}
createCar: function (void) returning void;
```

The function `createCar`() creates an instance of the class *car* in the '*data*' container, that's why this function doesn't return anything and doesn't take anything. The *data* container is the one which holds the lists of cars and people.

```
void createClient(void)
{
            char name[MAX];
            char surname[MAX];
            char address[MAX];
            person *p;


            // Ask for data
            cout << "Please enter a surname: " << endl;
            gets(surname);

            cout << "Please enter a name: " << endl;
            gets(name);

            cout << "Please enter the address: " << endl;
            gets(address);

            // Create a person
            p = new person(surname, name, address);

            // Store it
            /carshop/data/people.appendItem(p);

      cout << "End of creation." << endl;
}
createClient: function (void) returning (void);
```

The function `createClient()` also creates an instance of the class *person* in the '*data*' container, through a dialog with the user. It would be possible to create a visual interface, but anyway this was not done for the sake of simplicity.

```
Item *locateItem(/Common/Containers/listOfPointers *l, char * key)
{
          /Common/Containers/listOfPointers::iterator it = l->begin();
          Item * x;

          while(it != NULL)
          {
                    x  = ((Item *) l->getData(it));

                    if (!strcmp(x->getKey(), key))
                              break;

                    it = l->next(it);
          }

          return x;
}
locateItem: function(listOfPointers *, char *) returning Item *;
```

`locateItem()` is a general method able to find instances of `car` or `person` through the key, in any of the both lists in the container '*data*'.

```
void list(/Common/Containers/listOfPointers *l)
{
            Item *x;
        /Common/Containers/list::iterator it = l->begin();

        cout << "List\n"
                "=====================\n\n" << endl;

        while(it != NULL)
        {
                x = (Item *) l->getData(it);

                cout << x->getDescription() << endl;

                it = l->next(it);
        }

        cout << "=========================================\n"
             << endl;
}
list: function (listOfPointers *) returning (void);
```

The function `list()` dumps the contents of a list containing "Item's" in the display. We can this way display either cars or people.

As has been said, these two functions above list the contents of the lists in the '*data*' container. They will be used as an option in the menu, which is coded below.

```
void carshopMenu()
{
          int option;
          char buffer[MAX];
          Item *it;
          person *buyer;
          car *boughtcar;
          int price;

          do {
           do {
            cout << "1. Create car\n"
                 << "2. Create client\n"
                 << "3. Buy car\n"
                 << "4. Find client by surname\n"
                 << "5. Find car by plate\n"
                 << "6. List people\n"
                 << "7. List cars\n"
                 << "8. Exit\n" << endl;

            cout << "Please, enter an option: " << endl;
            gets(buffer);
            sscanf(buffer, "%d", &option);
           } while(option < 1 || option > 8);

           switch(option) {
                    case 1: createCar();
                              break;
                    case 2: createClient();
                              break;
                    case 3:
                        cout << "Please enter client's surname: " << endl;
                              gets(buffer);
                              if ((it = locateItem(&/carshop/data/people,
                                   buffer)) != NULL)
                              {
                                        cout << it->getDescription();
                                        buyer = (person *) it;
                              }
                              else    { cout << "sorry, not found."; break; }

                              cout << "\nPlease enter car's plate: " << endl;
                              gets(buffer);
                              if ((it = locateItem(&/carshop/data/cars,
                                   buffer)) != NULL)
                              {
                                        cout << it->getDescription();
                                        boughtcar = (car *) it;
                              }
                              else    { cout << "sorry, not found."; break; }

                              if (!(boughtcar->isAvailable()))
```

```
                                {
                                        cout << "Car already sold." << endl;
                                        break;
                                }

                                cout << "\nPrice of this car: "
                                     << boughtcar->getPrice() << endl;
                                cout << "\nPlease enter final price: " << endl;
                                gets(buffer);
                                sscanf(buffer, "%d", &price);

                                boughtcar->buyCar(buyer, price);
                                break;
                        case 4: cout << "Please enter client's surname: "
                                   << endl;
                                gets(buffer);
                                if ((it =
                                     locateItem(&/carshop/data/people, buffer))
                                     != NULL)
                                        cout << it->getDescription() << endl;
                                else    cout << "\nsorry, not found." << endl;

                                break;
                        case 5: cout << "Please enter car's plate: " << endl;
                                gets(buffer);
                                if ((it =
                                      locateItem(&/carshop/data/cars, buffer))
                                      != NULL)
                                        cout << it->getDescription() << endl;
                                else    cout << "\nsorry, not found." << endl;

                                break;
                    case 6: list(&/carshop/data/people);
                            break;
                    case 7: list(&/carshop/data/cars);
                            break;
                    case 8: break;
                         default:
                            cout << "Invalid option." << endl;
                 }
                } while(option < 8);

        cout << "\nOption: " << option << "\nFinished." << endl;
}
carshopMenu(): function (void) returning void;
```

The `carshopMenu`() function is the whole interface for the user, using the functions defined before it. From the menu, it is possible to create a new car or a person, locate one of them or list all persons and all cars, and mark a car as sold by a given client.

```
cd(../data);
void menu()
```

```
{
        ../bin/carshopMenu();
}
menu: function (void) returning void;
```

The user will enter in the '*data*' directory in order to work in the *carshop* application. The whole code is in the '*bin*' directory, so instead typing each time `../bin/carshopMenu`(), it is possible to create a function in charge of calling the appropriate function in the '*bin*' directory. The user now, only has to type `menu`(); in order to start the application.

```
menu();

1. Create car
2. Create client
3. Buy car
4. Find client by surname
5. Find car by plate
6. List people
7. List cars
8. Exit

Please enter an option:
1

Please enter a plate:
ou8954v
Please year of made:
1999
Please enter the color of the car:
green
Please set initial price:
6000

1. Create car
2. Create client
3. Buy car
4. Find client by surname
5. Find car by plate
6. List people
7. List cars
8. Exit

Please enter an option:
2

Please enter a surname:
García
Please enter a name:
J. Baltasar
Please enter the address:
Edif. Politécnico, s/n Campus de Ourense 32004 Ourense

1. Create car
2. Create client
3. Buy car
4. Find client by surname
```

```
5. Find car by plate
6. List people
7. List cars
8. Exit

Please enter an option:
6

List
==================


García, J. Baltasar. Edif. Politécnico, s/n Campus de Ourense 32004 Ourense
==================

1. Create car
2. Create client
3. Buy car
4. Find client by surname
5. Find car by plate
6. List people
7. List cars
8. Exit

Please enter an option:
3

Please enter client's surname:
García

García, J. Baltasar. Edif. Politécnico, s/n Campus de Ourense 32004 Ourense

Please enter car's plate:
ou8954v

Car: ou8954v. Year: 1999. Color: green. Price: $6000

Price of this car: 6000
Please enter final price: 5995

1. Create car
2. Create client
3. Buy car
4. Find client by surname
5. Find car by plate
6. List people
7. List cars
8. Exit

Please enter an option:
7

List
==================

Car: ou8954v. Year: 1999. Color: green. Price: $5995,
       bought by: García, J. Baltasar. Edif. Politécnico, s/n Campus de Ourense 32004
Ourense
==================
```

```
1. Create car
2. Create client
3. Buy car
4. Find client by surname
5. Find car by plate
6. List people
7. List cars
8. Exit

Please enter an option:
8
Option: 8. Finished.
```

As has been shown, the work in Barbados is done in an high interactive way. Classes and functions are compiled immediately as they are entered, as well as object creations. The editor distinguishes between declarations, definitions and executions, allowing the user to enter a function (such as `cd`(), or `menu`()), and execute them immediately, giving the impression that they are commands –system commands, in the case of `cd`()-.

The programmer doesn't need to create functions loading/saving the objects in the lists, as that is transparently managed by the system. The program is only needed to be entered one time, as the program also persist. All the user has to do in successive sessions with the program is to go to the *'/carshop/data'* container and type "`menu`();" in order to start execution. The data will be there as it was the last time he or she worked on it.

`menu()` is, in this context, the entry point of the program (Pettriek, 1994), as the `main()` function is the normal entry function in standard C++ (Ellis & Stroustrup, 1990). In Barbados C++, any function can be an entry point of an application, and any application can have more than one entry point.

Also, the lists in the '*data*' container can be inspected by any process, which doesn't need to be the application in the '*bin*' container. Even the user can do it interactively from the prompt. This makes Barbados to somehow resemble an object oriented database; however, in order to be considered as a true database manager, Barbados should offer services such as for example transactions, and it doesn't.

2.3 The theoretical containers-based model

The unit of transfer between user programs and the persistent store in this model is the container. A FGO (*Fine Grained Object*) is an object which can be defined in the language supported by the persistent system. Every container has an integer identifier (the *container_id*), which uniquely identifies it in the PS. A container is composed by a set of FGO's, which are reachable by a special object called *the root*. This root is always an object of the *directory* class. Directories are organised in a tree structure, similar to the one which can be encountered in any traditional hierarchical file system, as a directory can have subdirectories.

A subset of these FGO's is the set of *interface* objects, which are the only objects which can be referenced directly from other containers.

Containers are also the clustering system used in order to manage and organise the PS. Furthermore, they perform two additional (and related) functions: one of a greater level of abstraction, to the user, as a support for directories in which language-level objects are stored, and

another one, of a lower level of abstraction, as the unit of organization of the PS.

The existence of the 'containers' concept as sub-structure for the persistent store (*clustering*) prevents us from classifying Barbados as an orthogonal persistent model (Atkinson & Morrison, 1995). Instead of this category, the containers model (Cooper & Wise, 1995; García Perez-Schofield, 2001b) can support *type*-orthogonal persistent systems, just like the Barbados prototype. However, we the authors claim that the concept of containers, which mixes the best characteristics of traditional file systems (access rights, copy, migration), with persistence, is meaningful enough for the user that the loss of orthogonality (and therefore, transparency) in persistence is actually an advantage. Moreover, it gives important advantages in performance (García Perez-Schofield, *et al.*, 2001c).

The raising of the clustering system to the conceptual level, i.e., visible by the programmer, presents some valuable advantages. The mechanism of clustering used is directly managed by the user, who, through the abstraction of directories, indicates which objects must be stored together in the same *cluster* (container). This saves the system from having to use expensive automatic clustering techniques, which downgrade the performance of the whole system; other non-automatic ones are available, as well, however, they are not valid for all situations (Darmon *et al.*, 2000).

Currently, *swizzling* (the conversion of objects' OID to pointers and vice versa, between the PS and memory) in orthogonal persistent systems is normally done at the level of pages: this way, when an *unswizzled* pointer is detected, the system loads the referenced object from the cache or the referenced page to main memory from the PS (Morrison, 1999). This mechanism tries to minimise the number of *swizzlings* to be done, as well as the number of disk accesses, too. So, the system carries out the task of finding referenced objects or even load new *clusters,* whether necessary. This forces the system to assign special values to *unswizzled* pointers, permitting the system to distinguish *unswizzled* pointers from *swizzled* ones.

By contrast, containers in the container-based model are always loaded under the user's demand, and all the FGO's are loaded into memory in the one operation. *Swizzling* is done automatically, but is only needed if the container couldn't be put in the same memory address as it was previously. In this case, all pointers are found and swizzled, by adding to them the delta between the old memory address of the container and the new one. This mechanism is incidentally very similar to the way in which the same problem is solved in DLL's (*Dynamic Link Libraries*, Pettrieck, 1994).

About memory protection, the container-based model allows programmers to create data-structures which are in some sense autonomous (Cooper & Wise, 1995). Restricted data sharing among containers is allowed: *interface* objects –but only them-, can be directly referenced (using plain C++ pointers) from another container. This way it is possible to support a non type-safe programming language without the memory protection problems that would otherwise make it infeasible, as it is really difficult for an error to be propagated outside its container, or to risk the whole PS. Moreover, when a container is loaded in memory, the other containers with objects referenced from it are loaded in read-only mode (unless of course they are already in memory in read/write mode).

By contrast, supporting type-unsafe programming languages is not possible in an orthogonal persistent system: there is no way to cope with the related errors (such as errors derived from pointer arithmetic facilities, and uncontrolled casting), and as there is no division in the PS, it would be possible to corrupt the whole store. Although the choice of type-safe languages is attractive from

a theoretical point of view, in practice it results in a great limitation in the use of common tools and programming languages. Barbados currently supports C++ (Ellis & Stroustrup, 1990), a type-unsafe language.

In Barbados, identification of persistent objects is done by reachability when each container is closed (for example, when a `cd()` to another container is done). Garbage collection and compaction is done at this point. Note that this garbage collection doesn't affect the normal behaviour of *malloc/free* and *new/delete* C++ commands, i.e. we are not trying to redefine the C++ language, rather this reachability algorithm is designed solely as a means to identify the persistent objects.

2.3.1 Building Large Data-Structures out of Containers

Containers are not isolated data-structures. We expect containers to be in the order of kilobytes or megabytes, and any data-structure too large to be stored in a single container must be constructed from multiple containers. There are in fact 2 ways to build (really) large data-structures out of multiple containers: (a) Container references and (b) a special feature called Container/Name Swizzling ("C-N Swizzling").

2.3.2 Container References

Containers are identified by a value called a '*container id*'. *Container id*'s are simple integers internally, however the '*container id*' is a special type introduced into the Barbados variant of the C++ language, identified by the keyword '*container*'. This is one of the very few enhancements we have made to the language.

If an application program has a 'container' value, then it can use it to open and close the container using the '`OpenContainer`()' and '`CloseContainer`()' system calls. Now, under the view of the orthogonal persistence model, this reeks of non-orthogonality, it seems this model merely replaces the '`fopen`()/`fclose`()' functions with something equivalent but with a different name, in violation of the aims of the field of Persistence. However, although users have to open containers by themselves, there is not any need to specify anything else: the complex task of rebuilding a data-structure from a flat file, or creating objects and save them is automatically and transparently carried out by the system.

Application programmers are expected to write programs which traverse large data-structures by opening and closing the relevant containers as needed.

Note that the `OpenContainer`() and `CloseContainer`() system calls can be hidden inside other functions, as they are for example in the '`cd`()' call (i.e. change current directory), which in many cases will alleviate the need to use these calls.

2.3.3 The Ownership relationship: A subset of the Container References relationship

The '*Container Reference*' relationship actually contains an important subset: some of the reference to other containers are actually '*ownership*' references. This relationship can be represented as a tree of containers (as shown in figure 1), where the root container (its container identifier is always the number 1, the only special case) is always at the top of the hierarchy. Relations are unidirectional: if container **A** is a child directory of container **B,** then there is an ownership relation between **A** and **B**, going from **A** to **B** (i.e., **B** is the owner). Each container-directory can have references to any number of child containers, while a given container can have

only one parent container (owner).

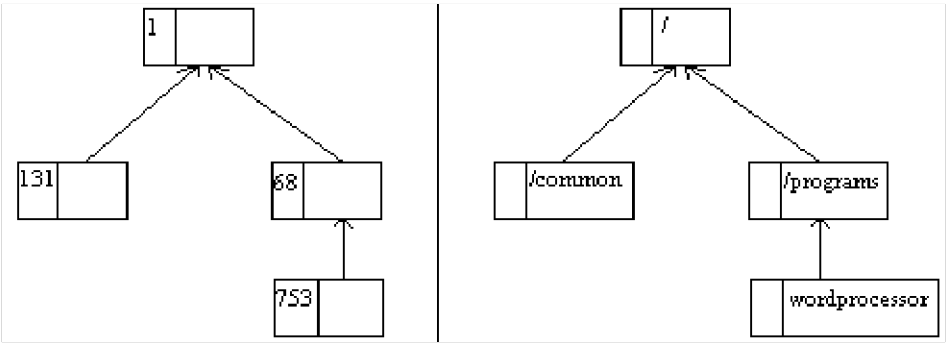

Figure 1. Ownership relation

The root container is guaranteed to exist, always. The user can therefore create and delete any directory, excepting the root one.

Let *U* be the universe of all containers in the system, and *r* the root one.

Then, always, at least, $\exists c \in U | c \equiv r$, so U is never empty, $U \neq \Phi$,and the ownership relation is defined such as $\forall c \in U, c \neq r \Rightarrow \exists c' \in U | c \rightarrow c'$, meaning that *c'* is the parent of *c*. This relation is transitive, meaning that $(\forall c, c', c'' \in U | c \neq r \wedge c' \neq r, c \rightarrow c' \wedge c' \rightarrow c'') \Rightarrow c \rightarrow c''$, and finally, the top of the hierarchy is the root: $\forall c \in U, c \neq r \Rightarrow \exists c'1, \ldots, c'n \in U | c \rightarrow c'1 \rightarrow \ldots \rightarrow c'n \rightarrow r$.

It is possible to easily define the subset of container-directories which are subdirectories of a given container-directory –this subset can be empty-. This is defined as: $\forall c \in U \Rightarrow (Sc \subset U | (\forall c' \in U, , c' \rightarrow c) \Rightarrow c' \in Sc)$.

2.3.4 Container-Name Swizzling

This kind of reference is called the **C-N Link relationship**: C-N stands for Container-Name. Relationships are set from an FGO in a given container, to a different, public FGO (through its name) in another container (through its *container_id* key value), conforming a pair (*container_id, name*). The relation between that containers is set, while containers are in memory, through plain C++ pointers (or references) by the user, and stored in an special format by the system, when the involved containers are saved to disk. For example, it can be achieved by creating a reference to another container:

```
…
cd(/Statistics);

void showSydneyPopulation(void)
{
  cout     << "Population of  Sydney: "
           << /Cities/Sydney/population
           << endl;
}
```

In the above example, there is a link between the container `/Statistics` and the container `/Cities/Sydney`, because the path of the variable `population` leads to the later container. Concretely, this will be stored as an absolute pointer (as it will be seen in next sections), in the C-N

Swizzling table at the very end of the container. Another example is shown in figure 2.

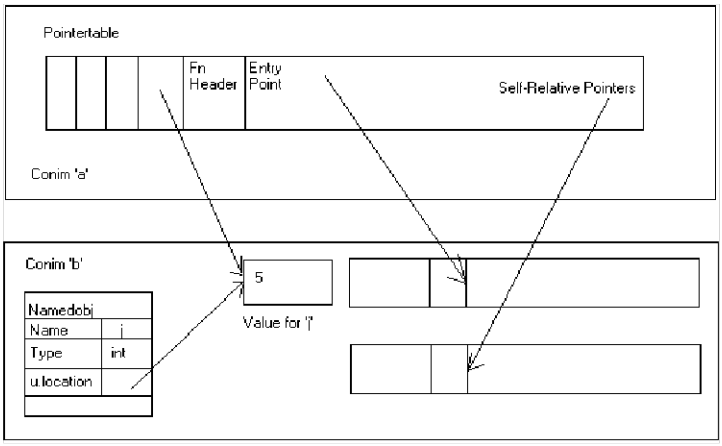

Figure 2. *CN-Swizzling between two containers: a function in the container* ***a*** *refers to a public of type* int *('i') in container* ***b****. There are other relations (self-relative pointers) to another two functions in the second container.*

C-N Swizzling is intended to serve as a vehicle for sharing among containers. Mainly the objective of this sharing is software, allowing a container to use functions or classes defined in another –library- container. The abuse of the C-N Swizzling mechanism would lead to a PS in which all containers are related: this would mean to load and save the whole PS each time a container is loaded, which would be sub-optimal.

C-N Links between containers can always be represented by a directed graph, sometimes – in fact probably very often - with cycles. Graphs with cycles present the problem which happens when an object in container ***x*** is referencing an interface object in container ***y***, and an object in ***y*** also references an interface object in container ***x***. If this happens, they will potentially not be unloaded from memory ever (apart from the system being shutdown). This is an interesting though easy problem which had to be solved in the *Container-Management* layer of the architecture. Ideally, this problem shouldn't happen, as in this case the programmer is suggested to put the contents of that two containers together (as in fact this two containers are behaving as only one).

Another kind of swizzling, as has been seen, happens when the programmer simply references one container by another. By this low-level mechanism, the programmer, directly through the API calls `OpenContainer()` and `CloseContainer()`, is able to store very huge amounts of data, scattered in different containers. Programmers are in charge of calling `OpenContainer()` and `CloseContainer()` in the appropriate order and the appropriate times. This low-level mechanism can always be avoided, by the use of the high-level mechanisms (directories) described above, provided the need of coping with really huge data structures doesn't exist.

Again, let's assume that all containers (sets of FGO's) in the system pertain to a U universe. A container, (which can be understood as a set of FGO's), can be found within the system in main memory or in the Persistent Store, but not in both of them at the same time $U = M \cup PS$. $\forall C \subset U \Rightarrow c \subset Mó, c \subset PS \mid M \cap PS = \phi$.

As can be seen in figures 2 and 3, a container must save all information needed in order to

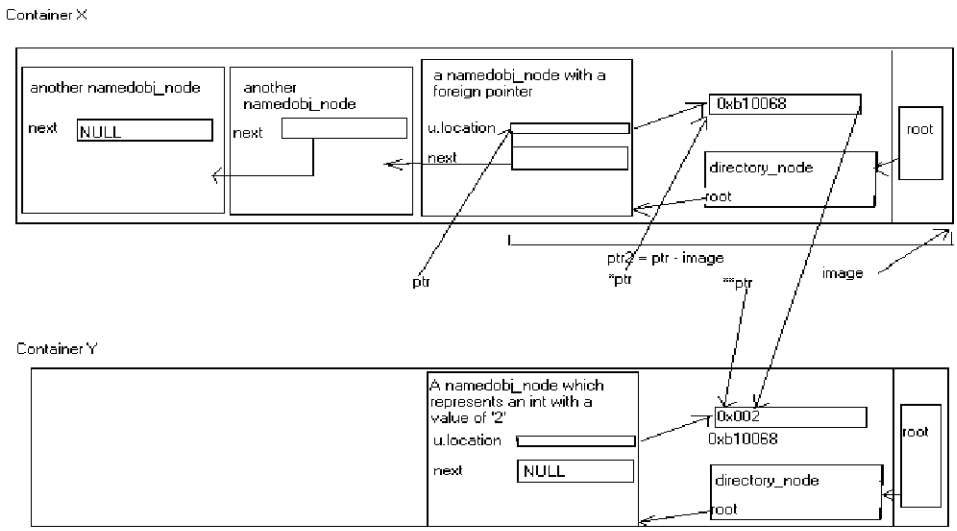

Figure 3. *Reference to objects in other containers from member data of interface objects.In Container X, an FGO has a pointer to an interface FGO in another object. That object is an integer of value 2, with an appropriate* namedobj *pointing to it.*

restore all relations with its prerequisite (i.e. the related ones by C-N links) containers. This information is stored in the C-N table, which generated by the system each time the container is saved in secondary memory. The basis for this CN table is the information generated by the compiler: the position of all pointers inside the compiled code; and the pointers present in the structures which represent classes. For example, pointers to other containers inside functions[2]. As the compiler emits code for functions (member functions or functions, in the concrete case of C++), it also generates a table in which the compiler stores the location of pointers inside the code.

Pointers inside code can be of two types: self-relative ones, which are resolved just adding their value to the address of the pointer (this was introduced because of 'CALL' instructions, as it provides more efficient machine-code and less pointers to swizzle at load time); and absolute ones, which are memory addresses storing referenced data. Finally, the *classdef-type* ones are absolute pointers which reference the definition of a class in a container different from the one in which the objects is defined. Figures 2 and 3 shows how these two kind of jumps can reference objects in other containers.

On the other hand, any object in a container can reference *interface objects* (necessarily, they must be interface ones), from other container. *Interface objects* are identified by CN pairs. This pair (*container_id, name_of_namedobj*), uniquely identifies an object in the whole PS; and this is also the way they are referenced in the CN table.

A relation between containers *c,c'* is defined therefore as a pointer between an object *o* of the set of FGO's of any container to a public object *p* of the set of public objects *P* of another container. It is relevant that if a container only has one object, this one must be necessarily public. Otherwise, it wouldn't be possible to relate any two containers (although this is not always necessary).

[2] Barbados emits native Intel code. There is not a bytecode language nor an interpreter, so the dependencies must be resolved before executing any function.

So, $\forall C \subset U \mid C \neq \phi \Rightarrow (\{\exists Pc \subset C \mid Pc \neq \phi\}, , \exists Ic \subset Pc \subset C)$, which allows to define the transitive relation $(\forall o \in C, \forall p \in Ic' \subset Pc' \subset C' \mid o \rightarrow p) \Leftrightarrow C \rightarrow C'$, so, $(\forall C, C', C'' \subset U \mid C \rightarrow C', C' \rightarrow C'') \Rightarrow C \rightarrow C''$.

As has been said, it is possible to represent the relations among containers as directed graph with cycles. It is also clear that the individual CN Link relations don't happen in both directions; they happen in the same direction that the pointer to the foreign data is signaling, and a reverse relationship doesn't happen. This way, if there is a link between containers A and B, i.e., A->B (A to B), then the B->A (B to A) relation is not implied, as A requires data in B but B doesn't need necessarily any data in A. This doesn't prevent the possibility of having a second CN Link relation from B to A. Precisely this latter case is an example of why the system will need to consider the possibility of having cycles, although they occurrence will be directly related to the behaviour of the user. It is also possible to find cases of cycles among containers which involve more than two containers. We discourage users of creating this kind of relationships, as this means that containers are going to be loaded and unloaded together in memory. Therefore, the possibility of cycles requires the system to have a garbage collector of containers[3] in memory, as none of the containers in a cycle are going to reach an unreferenced state: each of them is referring the other ones. Without cycles, it would be possible to use solely the technique of reference counting in order to know when a container is unused (Cooper, 1997).

From an implementation point of view, the suitable moment for generating these dependencies would be the time when the container is saved to secondary memory. These dependencies would be resolved in the moment of its eventual loading (Cooper & Wise, 1995; García Perez-Schofield *et al.*, 2001b). This is easy, as the information emitted by the compiler, plus the information in the structures representing classes, is available for all objects and functions, in the former case with their tables of pointers. It is only necessary to check if the pointer points to the address space of the current container or to another one. Then, the system must find the public object being referenced from outside. This is carried out in an efficient way thanks to a mapping in all containers which has been designed in order to be able to find all *interface objects*.

Once this information has been gathered, the CN-Swizzling[4] table of the container is built, storing the information needed for each foreign pointer: the container_id of the *foreign* container and the name (i.e., the C++ identifier) of the *named_object* (i.e., the interface object) being referenced; also, the address of that object in the foreign container at the current time (this way it is possible to know if the object has changed its position: perhaps no fixing is needed after all). After that, there is a set of pointers to the places in which there is a reference to that object in the current container, represented as an offset from the beginning of the container (containers are always compacted before being saved).

The reason to use the name/identifier of the object instead of an OID comes from the need to guarantee that the system is going to support schema evolution. The role of the OID in orthogonal systems is a CN pair presented above. A drawback of using these CN pairs, of course, is that the size of identifiers can be arbitrarily long, although this space is only spent on secondary memory (i.e., disk).

Finally, containers will be eventually loaded from secondary memory. This implies two kinds of swizzling: among containers (C-N Swizzling) and inside containers (local swizzling).

[3] This garbage collector of open containers shouldn't be confused with the garbage collector of FGO's which is executed every time the container is closed.

[4] Container-Name Swizzling, as has been explained.

Swizzling between containers is done first. Some of the prerequisite containers will be in memory and some might be on secondary memory. The first step is to load the containers into memory in their unswizzled state. The second step is to detect, by exploring their CN-Swizzling tables, which pointers must be swizzled and then to swizzle all these pointers in all containers which have been involved in the loading of the initial container. The third step consist of adding a delta to all pointers in those containers which couldn't be loaded in their preferred location (*local swizzling*). Once these three steps are done, the containers are ready to be used. In the second step the system must take into account the fact that perhaps the third step could require modifying the pointers swizzled between containers, so they are swizzled to the correct value minus the delta between the new address of the container and the preferred one (0 if it doesn't need local swizzling).

The complexity in this loading phase comes from the possibility of having cycles in the relations of the containers. Because of these cycles, it is not possible to apply a single-pass algorithm when loading a container, doing C-N Swizzling, loading itself the related containers and so on, since it would fail in an infinite loop.

The whole phase involves the following tasks: loading container C, and then load the transitive closure of C, that is, load all the related containers or transfer them, from a theoretical point of view from the PS to main memory M. Perhaps some of the related containers are already in main memory.

Furthermore, defining recursively the transitive closure on C, it is possible to develop the following function of loading:

$$CT(C) = \{\forall C' \subset U \mid C \rightarrow C'\}$$
$$CT(C) = CT(C) \cup \{\{\forall C'' \subset U, \forall C' \subset CT(C)\} \mid C' \rightarrow C''\}$$

$$f : PSxUx\ldots xU \rightarrow MxMx\ldots xM$$
$$\forall C \subset PS, \forall C' \subset CT(C) \xrightarrow{f} \{\{PS = PS / C, M = M \cup C\}, \{\{\forall C'' \mid C'' \subset PS\}$$
$$\Rightarrow \{PS = PS / C', M = M \cup C'\}\}\}$$

2.4 The implementation in Barbados of this model

Barbados is an object oriented persistent system (Cooper, 1997): this means that there is not a file system, but a persistent object store. Programs, automatically and transparently to the user, save and restore all objects they use: this way programmers don't have to write code saving and restoring data which will be needed in later executions. This saving and restoring always implies a codification of the structures in memory to a flat sucession of bytes, and a later decodification to the appropriate structures in memory: programmers are freed of these tedious and error-prone tasks.

Other systems similar to Barbados are PJama (for example, Dimitriev & Atkinson, 1999), a Java-based persistent programming language, JSpin, (Kaplan *et al.,* 2000), a persistent programming system which allows multiple languages, built over an Object-Oriented Database Management System (OODBMS); PerDis (Shapiro *et al.*, 2000), which is a middleware (mainly a

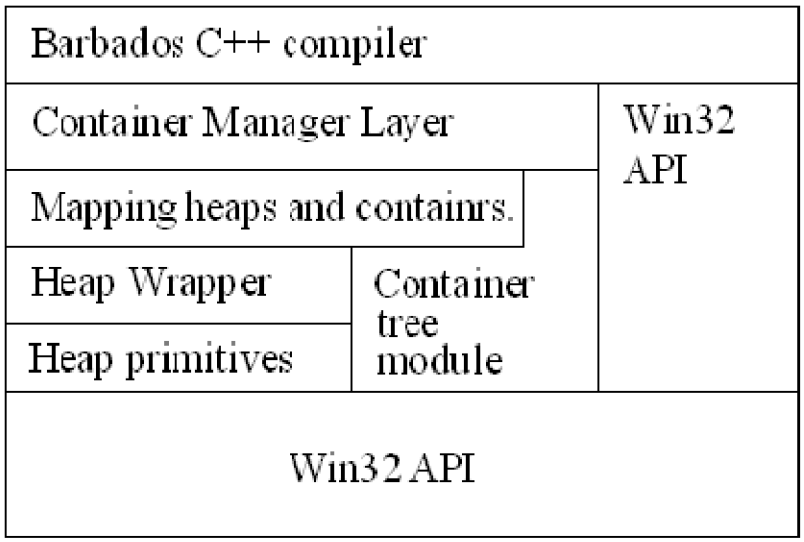

***Figure 4**. Architecture of Barbados*

daemon and a library) which can be used to benefit of persistence from any program; and Napier (Atkinson & Morrison 1989), a quite old persistent programming language based in Algol and later in PS-Algol. All these systems will be discussed and compared with Barbados in the *state of the art* chapter.

Barbados' unit of transfer between memory and the persistent store is the container (a Large Grained Object), which is a collection of FGO's (Fine Grained Objects). A FGO corresponds to any normal C++ object (class instances, primitive data ...). FGO's are distributed in non-overlapping containers.

One of the purposes of such a schema is to organize the persistent store in some way, preventing the programmer of converting the store in a soap of spaghetti pointers (Cooper & Wise, 1995). The application programmer is required to explicitly perform the operations of creating, opening and deleting containers, except of course when these operations are hidden inside other commands. This might seem contrary to the goal of Persistence (at least with Orthogonal Persistence, please see Atkinson & Morrison 1989), which is to minimise the effort of moving data between secondary memory and main memory, but because these operations happen at such a *coarse-grained* level we think it's nevertheless consistent with the goals of persistence (Cooper & Wise, 1996). The figure 4 shows the architecture of Barbados.

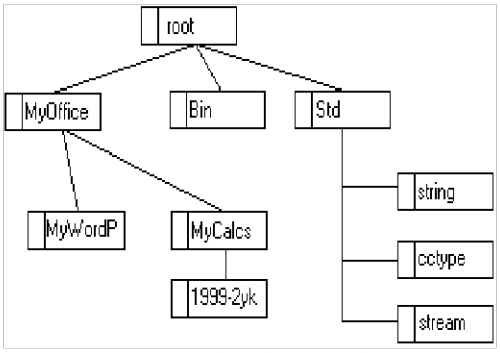

***Figure 5.** An example of a tree of containers*

A description of a container is just a set of FGO's. A special FGO of these inner objects is

called the ROOT, because *all* the remaining FGO's are reachable from it. The root always consists in an object of the *directory* metaclass. This metaclass (the term metaclass will be explained later) represents essentially a set of `namedobj`'s. A named_obj is a (*name, type, value*) triplet, or equivalently: (*name, any*) where 'any' is defined as a (*type, value*) pair. These objects have a special property: they are objects that have a "*public*" name by which they can be referenced. The directory itself (the root), each named_obj, and various FGO's reachable from these named_objs using plain C++ pointers/references are all stored inside the same container.

This means that directories can be nested: if a directory contains a named_obj of type directory, then that subdirectory will be part of the same container (an example can be seen in figure 5).

In other Orthogonal Persistent Systems, at the physical level, there is always some kind of clustering. Clustering (Sousa & Alves, 1994) consists on the grouping of near objects when saving the Persistent Store using the underlying platform. The definition for 'near' in this context changes from one system to another one: it can mean objects and its related classes, objects composing other objects … some kind of object relation. In Barbados, the clustering mechanism is based on containers (user-dependant) which in turn are based on reachability.

2.4.1 Glossary of terms

Follows a small glossary of the terms that will be used during this chapter.

- **Class** : A class in the C++ sense that the user (i.e. application programmer) creates.
- **Internal class**: A class that Barbados uses internally only and which is not visible to the user
- **Metaclass**: A class that spans both the internal world and the user world, i.e. a user class which is especially recognised by Barbados for the purpose of providing persistence.
- **Type**: Any possible C++ type.
- **FGO**: A Fine Grained Object: It corresponds with the normal object in C++, as structs objects and primitive types. Also, a function is considered as a FGO.
- **Root of container**: This is the FGO which is the root of an container. It is always an object of the *directory* metaclass.
- **Container**: A container is a collection of FGO's, with a *directory* as its root: all object stored in the container must be reachable from this root. It's a way of partitioning the Persistent Store: having a directory in the container's root, we allow to structure containers in directories.
- **Container_id**: it's a 4-byte integer identifying uniquely the container for Barbados. This is many times abbreviated in this chapter as *C_id.*
- **Container's associated file**: as Barbados runs over the Win32 platform, a container must be represented as a Win32 file. Each container, which is identified by a number called *Container_id*, has an associated file.
- **Preferred Base Address**: this is the memory address at which the container was loaded last time. If the container is loaded again on that location, then no swizzling will be needed.
- **/Common**: It's a special container that exists in the initital Persistent Store, before the user creates any other object. It is designed to offer or suggest a place where to put those objects

that are commonly and frequently used by the programmer.

- **/Common/STD**: It's a special container that exists in the initital Persistent Store, before the user creates any other object. This container has many subcontainers (subdirectories) as possible #`includes` are found in the C++ standard library specification.
- **Paths:** it's a UNIX-like path identifying a container at user's level. For example /STD/vector.
- **ROOT container of Persistent Store**: This is the container with `container_id` = 1, which is at the root of the directory tree and therefore the entire persistent store
- **Persistent Store**: It's a huge pool of containers in which all the Barbados' objects live. We get access to the objects in this pool navigating through directory-containers.
- **Persistence**: The capability of an object to survive to the process that created it. It must be able to be used by any other process.
- **Swizzling**: It means to convert the pointers of a given object, from its representation in memory (in which they are pointers, in the most systems), to its representation on secondary storage (for example, this representation could be the object identification of the object being pointed to).
- **Heap**: The heap is a special memory zone in where all data is stored for a given container. It is contained in a *Conim* (please see the Container Management Layer section), and it is its main component.
- **Conim**: It is the representation in memory of a container and is little more than a heap attached to a *container_id.*
- **Memory block or tile**: A memory block is a term used to designate a single contiguous set of bytes, usually small space allocated into a heap.
- **Directory**: a metaclass which is able to store named_objs.
- **Named_obj**: it is a metaclass which identifies an object through a public name. The object is a C++ object that must exist in the same container. It's a pair (name, any)
- **Any**: metaclass which is an infinite union of types.

2.4.2 Definitions

- **Orthogonal Persistence**: The persistence offered by a system is orthogonal if a) all types in the programming language can persist, b) there is no difference for the programmer to deal with persistent and transient data, and c) the identification of the persistent data is done by the system: this is achieved by transitivity, from a root object which is known to be persistent (this is a brief summary of the conclusions exposed in Atkinson & Morrison 1995).
- **Swizzling and swizzling used in Barbados**: when any persistent system stores an object, it has to take care of the pointers of that object, in order to be able to a) translate the pointers in memory to a recoverable codification on disk, and b) do the inverse translation to the correct locations the pointers must point, taking into account that the object is not possibly in the

same memory address as it were last time (Moss, 1992). Barbados does not do exactly a translation, as pointers on disk are still memory pointers, and the container is prepared for running, provided it's located in memory in the expected position the next time is loaded. Swizzling is only carried out when the container is to be stored on disk, if its location in memory has changed, and when the container is loaded, if it wasn't possible to load it in its expected address. This way, swizzling is not needed to be done again until the container is loaded in a different location, or the container changes. Moreover, swizzling is this way a matter of adding a given delta to all pointers in objects, avoiding the transformation step from an OID to a memory address. A graphic description of a container is given in figure 6.

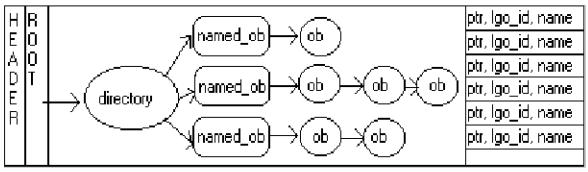

Figure 6. *Representation of a container.*

- **C-N Swizzling:** this kind of swizzling is only found in Barbados, and it solves the matter of pointers that point to outside the container, i.e., to another container. For each container, there exists a list of (*container_id, named_obj*), which helps locating the containers this container is related with, in order to load them in memory with the container which is doing the references. Secondly, the FGO's being referenced are located. Then, the pointers are arranged in memory to the correct locations (FGO's). This mechanism is totally transparent to the user.

2.4.3 Barbados Compiler

The compiler (Cooper, 1997) is the layer the user deals with. It has to support the C++ (Stroustrup 1991; Ellis & Stroustrup, 1991) language with the special commands and structures that made possible to use the special mechanisms of Barbados. We will use the term "Barbados C++" to reference the whole language supported by the system.

2.4.3.1 Metaclasses

Metaclasses are the intersection between the internal structures of Barbados and the programmer's world. They can be used by the programmer as any other C++ class. The possible metaclasses are:

- **any**: *any* is a very important metaclass, because is able to represent every possible type in Barbados C++ . It's the infinite union of types of the language supported by Barbados. It is important to distinguish *any* from *void* *, which is a pointer in C++ capable to point to every C++ type, although it not stores information about the type of what object is pointing to. More technically, this is a (type, value) pair where the type of 'value' is defined by 'type'.
- **directory**: it represents a directory. This class holds a list of references to a set of **named_objects**.
- **type**: this represents any type in Barbados C++. There is an operator called `typeof()`, which returns the type associated with any variable. This is not really a class, it is a '`unsigned char *`' sequence of bytes. This type defines types. Most important types are classes, but not all types are classes: e.g. "`int`", "`char *`", "`C *`", "`C[]`" are not classes.
- **named_obj**: A *named_obj* is just a mechanism to map a FGO with a name. It is a (name, any) pair, for example being useful to identify the components of a **directory**. This can be decomposed into a (*name, type, storage-class, value*) tuple. It represents an identifier in the C++ language, as opposed to the value which the identifier itself represents. For example, the compileable expression "`int i;`" will create not just an integer variable but also a `namedobj` object which stores the name '`i`', the type '`int`' and the storage-class '`static_storage`'.
- **classdef**: this is a true class-definition class. A '`classdef_node`' is a sequence of members, representing *classes*, *struct*'s and *unions* (which are all similar from an implementation point of view.

2.4.3.2 The Language to be supported by the compiler

We are dealing with C++ language, which introduces the problem of type safety. In C++, programmers still have the possibility (as in C) of using pointer arithmetic, as well as to do indiscriminate casts. These two issues make the implementation of a C++ persistent environment different from other type-safe languages. Memory protection is provided with containers and therefore, inside each container, these unsafe operations are allowed.

The Barbados C++ language consists of:

- The Standard C++ Language

- UNIX-style path names which identify FGO's, and container commands which are the following: `Opencontainer()`, `Createcontainer()`, `Deletecontainer()`, and `Closecontainer()`.
- Metaclasses

Also, the Barbados language offers libraries in order to wrap the calls to containers in some cases (for example when navigating through containers, a set of Unix-like directory commands), and GUI non-standard classes in order to manage the input/output of the application.

2.4.3.2.1 Description of the Language parts

The Barbados C++ type-system is built up of these constructs:

- **Fundamental types**: such as `int`'s, `double`'s, `float`'s ... etc
- **Classes and structures**: the nominal C++ structures, capable to contain methods and data members. A special structure is the so-called '`union`': all the members of an union have the same physical beginning. The unions introduce some problems when dealing with persistent data, as will be studied in a section below.
- **Functions**: this includes C++ non-member functions and member functions (methods of a class).

The container commands are very important since they are used to hold the UNIX-like directory commands:

- **CreateContainer**() – Creates a container of any type. Returns an any pointer to the root.
- **OpenContainer**() – Opens any existing container. Returns an any pointer to the root.
- **CloseContainer**() – Closes any existing container. It means that the user won't use it, at least for a time.
- **DeleteContainer**() – Deletes any container, definitively.

The metaclasses present in Barbados are:

- **type** – the operator `typeof()` returns a value of this type.
- **any** – a (type, obj) pair. Somehow similar to pointers `void *`, but storing the type of the object.
- **named_obj** – a (name, any) pair.
- **directory** – is a collection of *named_objs*.

The UNIX-like wrappers for directory commands are:

- **cd**() – allows user to "enter" (i.e., open a directory container) into a directory (involves a call to `OpenContainer()`), and exit the previous current directory. Also it must be mentioned that it exists a fine-grained version which changes to a plain directory instead of a directory container.
- **mkdir**() – creates a new directory container (involves a call to CreateContainer()). It is defined as a macro: `#define mkdir(s) directory container* s = Createcontainer(.)`

- **rmdir**() – deletes a directory container (involves a call to DeleteContainer())

2.4.3.3 How Barbados represents the C++ type system

The following are the main structures used in order to represent types in Barbados C++. For each one of the following representations, there will be a suitable *FindPtrs*() function which will return a list of the pointers to be swizzled on it. Swizzling has to do with persistence, which is one of the first objectives of Barbados.

The operation of the saver and the loader relies on the ability to find pointers in a container. When saving, we need to find all pointers in order to find all objects referenced and then do garbage collection. When loading, if we fail to load at the preferred address then we will need to add an offset to each pointer in order to relocate it.

2.4.3.3.1 Fundamental types and storage considerations

Actually, the items described in the first part of this section are just identifiers to use in the *named_obj* structure, which is the one that stores the information related to the type of the object (although it can point to a *classdef*). This structure is presented at the end of the present section. In the figure 7 all storage possibilities are shown.

```
typedef enum { unknown_storage,
               static_storage, perprocess_storage,
               perthread_storage, local_static,
               member_storage, inherit_storage,
               auto_storage, parameter_storage, const_storage,
               straight_fn, library_fn, member_fn,
               virtual_fn, oneinstr_fn,
               typedef_storage, macro_storage, keyword_storage
} storage_enum;
```

Figure 7. *Available storage possibilities in Barbados*

The storage classes and their interpretations are as explained below, in table 1. Many times explanations refer to the `classdef` class, thoroughly discussed at the end of this section, and presented in figure 8.

static_storage	The most common storage class. This refers to a variable (not function) which is defined in a directory and not in a class or function. *u.location* is a direct pointer to the data which is embedded in the same heap tile as the namedobj. This is to make memory management easier (they can be freed with one call to free).

perprocess_storage These variables are local to the container they belongs to. This has the same basic meaning as static, but the objects noted as transient_static will be reset to its default value after any time its container is closed.. This mechanism is needed because the well-known static variables are expected to maintain its value until the program finishes. But, in a persistent system, a static variable would never had an opportunity to be reinitialized, because it's value will persist. It must be noted that this behavior is not orthogonal, but it was added in order for the user to feel comfortable using Barbados C++

perthread_storage The same meaning with perprocess_storage, but relative to threads.

member_storage This is for data-fields of classes/structs/unions. *u.offset* gives the byte offset of the field.

inherit_storage This is very similar to member_storage in many ways. The main difference is actually a scoping issue: the fields of these fields have the same status as member_storage fields. It's used to implement inheritance of course.

auto_storage Used for auto variables i.e. local variables which are allocated on the stack. *u.offset* represents the byte offset of the data with respect to the frame register, and is always negative. On Intel chips, the BP register is conventionally used as the frame register and we have kept the tradition.

parameter_storage This is used for parameters to a function. It is very similar to auto_storage, the only real difference is that the offsets are always zero or positive.

const_storage This is used for constant values of size <= 4 bytes. *u.constval* represents the value. It is the storage class of enum constants (and nothing else at present). It should not be confused with the tp_const type modifier which is a different thing.

straight_fn This represents a function compiled by the user and not a class member function. The function's compiled code is in a separate heap tile, pointed to by *u.funcblock*. *u.funcblock* is of type 'funcblock' which is an internal type used to represent functions. The entry point of the function is actually this same address, but an extra header of information is stored at the end of the block – (you reach it by finding the size of the block, subtracting the size of the header and taking that address). This rather complicated format was chosen because it was becoming rather complicated to maintain a function entry-point which was different from the address used to identify the heap tile. The main complication arose from the fact that library functions e.g. printf() that came from Visual C++ couldn't conceivably have a funcblock header.

library_fn This is essentially the same as above but is used for functions provided by VC++. If we want to implement our own calling conventions, e.g. values passed through registers, then the different calling conventions will form *part* of the solution.

member_fn	This is essentially the same as above but is used for non-virtual member functions. Note that the type of the function does *not* specify the 'this' object as a parameter, but the compiler's intermediate code and compiled code act as if 'this' was the first (i.e. lowest in memory, last to be computed) parameter to the function.
virtual_fn	This is for virtual functions. *u.virtualfn_idx* is an integer which ranges from 0 to some value and represents the index into the classdef->VirtualFn array. (To find it, you need to subtract 4 from 'this' to get the type-ptr, then dereference and add the offset of 'VirtualFn' and *u.virtualfn_idx*, and then dereference, and pass the result into the program counter). ((voidvoid_fn*)(&classdef->VirtualFn))(*u.virtualfn_idx*) points to a funcblock with the same representation as the above functions.
oneinstr_fn	This represents instructions which are represented directly by intermediate-code (which usually means they are directly implemented by Intel instructions). Examples include integer addition, floating-point multiplication, trig functions, int-to-float conversions etc.
inline_fn	This represents an inline function. It is not yet implemented. *u.ipr* will point to an intermediate code representation of the function. This will enable all calls to the function to make full use of opportunities for optimisation, we're *not* just eliminating a CALL/RET instruction pair.
typedef_storage	This represents a typedef name which also includes class/struct/union tags and enum tags as well as named declared with 'typedef'. Note that the type is defined by the *type* field, and the *u* field is completely unused. In this sense, you can see that C++ is not much different from prototype languages which use ordinary variables as templates for creating other variables. This storage class has the property that it affects the parsing process at one of the earliest stages: i.e. if a compileable string starts with a typedef name then we compile it as a declaration, but if it starts with a namedobj of some other storage class then we parse it as an expression.
macro_storage	The Barbados compiler has the preprocessor built into the lexical analysis routines as a layer rather than as a separate pass. This means that macros form part of the same name-space as all other identifiers. A macro's expansion string is given by *u.macro* and is equal to the string the user gave it but slightly processed (macro parameters are replaced by the characters '\1', '\2', '\3' etc. and a byte at the front specifies the number of macro parameters).
keyword_storage	For convenience in the compiler we've implemented keywords as namedobj's. *u.keyword* returns an enum value that we then use for parsing.

Table 1. *Deep explanation of available storage classes.*

- Visibility for member data in C++ classes and structs is *private*, *public* and *protected.*

```
typedef enum { private_visibility,
               public_visibility, protected_visibility }
visibility_enum;
```

- Fundamental types

Types are represented through the type metaclass, which is represented using the following enumeration.

A type basically consists of a sequence of `char`'s. It is also called a '`type-string`'.

The first char identifies the associated data entity. All possible values are shown in table 2. This first char in a `type_type` is to be interpreted using the following enumeration set:

```
typedef enum { tp_error='!',
         // Datums:
         tp_void='v', tp_bool='b', tp_char='c', tp_enumerated='e',
         tp_short='2', tp_int='i', tp_long='l', tp_float='f', tp_double='g',
         tp_uint='u', tp_uchar='B', tp_ushort=':',
         // Building blocks:
         tp_class='{', tp_pointer='*', tp_reference='&', tp_array='[',
         tp_dynarray='A', tp_function='(', tp_terminated=')', tp_volatile='~',
         tp_const='K', tp_container='O',
} tp_enum;
```

Symbol	*Meaning*
v	void
b	bool
c	char
e	enumerated
2	short int
i	int
l	long int
f	float
g	double
u	unsigned int
B	unsigned char
:	unsigned short int
{	class
*	pointer
&	reference
[	array
A	dynamic array
(	function

Symbol	*Meaning*
`)`	`End of type string.`
`~`	`volatile`
`K`	`const`
`0`	`container`

***Table 2**. Symbols used in order to identify types.*

- **Fundamental datums:** If the type represents a single fundamental datum, i.e. something in the first section, then the *type_type* (internal representation of *type* metaclass) is only one character long. *Type*s are *not* null-terminated strings. They can't be, because they often have pointers embedded in them which will almost always contain a zero byte. Instead, it can be calculated in its length by calling '`LengthOfTypestr(type_type)`' which interprets the type-string according to the rules defined here.
- **Pointers and references and Container-id's:** If first byte (or any byte of the type) is a `tp_pointer` or `tp_reference` or `tp_container`, then the type represents a pointer or reference or container-pointer to the type which follows straight on.
- **Arrays:** An array *type_type* consists of at least 6 bytes: the *tp_array* byte, 4 bytes of the length, immediately following on; and then the rest of the type follows on from there.
- **Classes, structs and unions:** These 3 possibilities are all stored using the '`classdef_node`'. A type representing a class has 5 bytes: *tp_class* followed by the 4 bytes of the pointer to the former.
- **Functions:** A function is represented by `tp_function`, followed by the return type, followed by a single byte giving the number of parameters (therefore Barbados is limited to functions with at most 255 parameters), followed by this many `type_type`'s concatenated straight onto the end of the preceding parameter's type. Finally, purely as a check for corrupt data, type is ended with a '`tp_terminated`' byte at the end of the last parameter's `type_type`. `Type_type`'s are normally extremely short except for functions which can be considerably longer than other `type_types`.
- **Other tp's:** `tp_dynarray` represents a non-standard feature in Barbados which corresponds with (it was a substitute for STL sets and sequences until the new standard). `tp_volatile` and `tp_const` can be inserted onto the front of any type to have the usual meaning.

Note that the presence of numerous pointers and integers inside a `type_type` can lead to many non-aligned pointer dereferences. Our assumption is that they will be slower than aligned pointer accesses but not disastrously so, the alternative would be to use `memcpy()`'s or some other form of encoding of pointers.

- The figure 8 shows the *namedobj* class. Normally, the *namedobj* represents the identification of a type or datum (its name), while the type itself is represented by a *classdef* class. When the type being represented is simple enough, such as an '`int`' or a '`double`', then this type is

```
class namedobj {
            make_node *make;                        // Make information.
            namedobj  *next;                        // (For a linked list
                                                    // of obj's)
            union {
                        void* location;             // Use this for
                                                    // static_storage
                        int offset;                 // Use this for
                                                    // parameter/auto/member
                                                    // storage
                        char* macro;                // Use this for
                                                    // macro storage
                        int constval;               // Use this for
                                                    // const_storage (enums)
                        funcblock_type funcblock;   // Use this for all
                                                    // functions
                                                    // except virtuals
                        int keyword;                // Use this for
                                                    // keyword_storage
                        int virtualfn_idx;          // Use this for virtual_fn
                        struct {                    // Use this
                                                    // foroneinstr_fn's
                                    char k;
                                    char tp;
                        } ipr;
                        directory_type dir;         // (A shortcut for
                                                    // static_storage
                                                    // directories).
            } u;
            str name;                               // The name
            uchar storage;                          // The storage class.
            uchar visibility;                       // public/private.
            char overload_version;                  // For making overloaded
                                                    // names
                                                    // unique.
            unsigned char type[1];                  // The type
} *namedobj_type;
```

Figure 8. *The namedobj class.*

directly represented in the *type* field of the *namedobj*, in the way explained below. But, if the type or the datum needs a *classdef* to be defined, then the contents of the type field will be an '{' character (`tp_class`) and a pointer to the *classdef*.

- *name* is a C-style null-terminated string. This name is the name of the namedobj as it is visible by the user, from the Barbados' environment.
- *storage:* The *storage* field is used to enable us to interpret the *u* field. This is a 1-byte enum which is meant to be interpreted using the *storage enum* set (explained above). The *u* field is a union of different members, approximately one member per storage class.
- *next* is used to form a linked list of namedobj's. Linked lists of namedobj's are used both in class definitions and directories. In the first case, the linked list corresponds to the methods (member functions, in C++) of the class. In the second case, the list corresponds to the namedobjs in the directory.
- *make* is used to point to a '`make_node`' which is a purely internally-used class used to implement the fine-grained make algorithm.
- *visibility* is used for class members. It defines the member as *public*, *private* or *protected.*
- *overload_version* is used for overloaded functions to help us differentiate between different overloads of the same function name. It is used to link member-functions

with their source-code objects and to give users a way to identify specific methods e.g. for specifying the target of a 'delete' command.

- *type* is declared as having 1 char, but we are using the old trick of having data carry on past the end of the struct. The length of this field is actually variable, depending on the length of the `type_type`. For example, an '`int`' would be represented by a single character '`i`' (`tp_int`) in this field, while a function would take more characters. Actually, this field is not the only reason why we overflow the end of the namedobj: the other reason is that *static_storage* objects have their data memory embedded in the same heap tile at the end of the `type_type` field (and aligned up to a multiple of 4 bytes). For these two reasons, you can't create a `namedobj` using the normal **new** operator, nor can you inherit from `namedobj_node`. The same discussion applies here as to `classdef_node`'s.

If the user enters in the editor "`int number;`", then there will be solely one new *namedobj* in the system (i.e., in the current *container*). The '`name`' field will store the string '`number`', while the '`u.location`' field will store a pointer to the location of the place in which the 4 bytes for this `int` number are stored. The '`type`' field will content an '`i`' character (`tp_int`).

Note that not in all cases a *namedobj* is created for a given object. For example, the objects created with the operator `new` don't have a *namedobj*. Nevertheless, in all cases, there is a pointer in the header of the tile to the *classdef* of the type if the type of the object is complex. If it is a simple type, then the type string is stored inside the tile (and the header of the tile points to that string).

There will be a suitable `FindPtrs()` function for *namedobj*'s, as well as for any other metaclass. As explained previously, this function will return all existing pointers in the class, allowing us to *swizzle* them for loading/saving the container. The `FindPtrs`() function is expected to return a a) NULL if there is not any pointer to swizzle or b) a pointer to a vector of pointers to be swizzled, ended with a NULL one. For example, with fundamental types, there are not pointers present, so we return a NULL. A draft of this function will be completed in the following sections.

Actually, the `FindPtrs`() functions are implemented not to return a list of pointers, but a series of calls `FindPtrs`() functions, processing all the pointers "on the fly". But for clarity, we will consider it as a function returning a NULL-ended list of pointers (so all `FindPtrs`() functions listed here are indicative only).

```
void **FindPtrs(namedobj_node *x)
{
     void **dev = NULL;
     for (int n = 0; n<LengthOfTypeStr(x->type); ++n)
          switch (*(x->type++)) {
               case tp_bool:
               case tp_char:
               case tp_enumerated:
               case tp_short:
               case tp_int:
               case tp_long:
               case tp_float:
               case tp_double:
               case tp_uint:
               case tp_uchar:
               case tp_ushort:
                    break;
               case tp_pointer:
                    dev      = (void **) malloc(sizeof(void*)*2);
                    *dev     = &x->location;
```

```
                    *(dev+1) = NULL;
                    break;
               case tp_class:
                    dev = FindPtrs(((char *)x->type) + 1);
                    break;
                                                …
          }

     return dev;
}
```

2.4.3.4 Unions, classes, structs and namespaces.

Each class/structure is represented using a `classdef`, which has a collection of `namedobj`'s, one for each member. In addition, unless it is an 'unnamed' class, there will be a `namedobj` which has the storage_class *typedef_storage* which associates the class name with the *classdef*.

These two structures must be examined for each class, struct or union when saving the container.

The C++ feature of '*namespaces*' is not available in Barbados. The reasons are that (a) the same benefits are provided by the '*directory*' feature, and (b) use of such a feature would break Barbados's "fine-grained make" i.e. incremental compilation feature. Therefore programmers will be forced to use *directories* instead of *namespaces*.

2.4.3.4.1 Schema Evolution

When representing classes and structs, the schema evolution problem (as found in the PJama programming language, Dmitriev & Atkinson, 1999, and in the O_2 OODBMS, Banchillon *et al.*, 1992) must be taken into account. For a more detailed discussion about problems when dealing with schema evolution, please refer to the chapter titled "Design and Implementation of Schema Evolution in the Container-based Model".

2.4.3.4.2 Unions

In the concrete case of the unions, a little extra problem is present. If, for example, a union is composed by an `integer` and a `char *`, then, the matter of whether the pointer should be swizzled arises. If the pointer is swizzled, all works correctly, provided the programmer is using it as a pointer in order to point some object, but if he or she were using it as an integer, then he or she would find an unexpected value there.

One possible solution to the problem present in unions is to have an extra bit inside them, denoting whether the pointer must be swizzled or not. This field would be set when the union were assigned for the first time, so if the field set was a pointer, then it would be automatically set to '*swizzle*'. This allows the most common programmer's behaviors, explained in points 1 and 2, but not other possibilities such as point 3:

1. using the union in a limited variety of situations, in a very specific domain: one field is used in one situation, but never in the other ones. This is typically used in linked lists and other similar data structures, where the ability of the records of containing mutually-exclusive data is represented by an indicative field. The presented mechanism covers this situation.
2. doing conversions: the programmer sets the pointer and then uses the other field in order to take its integer value, for example, or vice versa. This behavior is also covered.

3. doing what we could define as “settings of union fields through strange casts”: for example, in a union with an integer and a pointer, the programmer could try to initialize the pointer through the integer field. This would only work every time the container is loaded in memory in the same address it was when the field was set (because local swizzling is not performed). The opposite case wouldn't work, because of the same problem resulting in destruction of information.

For the first thought, the solution presented above is satisfactory as it covers the two first cases, which we think are the most common. Of course, the programmer should be warned about this later case.

However, exploring other possibilities, we discovered we can’t rely on this mechanism in order to solve all possible situations. This is because in C and C++, we can obtain pointers of the adequate type for any variable or any struct field. For example:

```
union mix {
          char * x;
          int y;
};

mix t;
char **ptr;
int *x;

ptr = &t.x;
x = &t.y;
```

It would not be possible to detect this cases and fix all pointers.

We are therefore sure we cannot offer any definitive solution about this problem. Since unions are a live characteristic of the C++ language, we should give support to them (although C++ users should take advantage of the object oriented mechanisms of C++, instead of using C mechanisms as unions).

Finally, we have concluded users should be able to build their own `FindPtrs()` function for *unions* or even for *structs*, in order to let the user choose how pointers in his structure are swizzle. This certainly leads to a loss of orthogonality (Atkinson & Morrison, 1995), but it must be noted that C++ has its own mechanisms that, as we have already said, substitute unions. In sum, this is a partial solution provided only for the user who still wants to use unions.

2.4.3.4.3 Structures and FindPtrs() functions for classes, structs and unions

We use the following when representing `classes`, `unions` and `structs`:

```
typedef unsigned char* type_type;   // Points to byte-size ‘tp_enum’s.

struct classdef_node {      // Classes, structs & unions
#define HELLO_BABE  0xe110babe
 int4 Version;   // How many times this class/struct has been redefined.
```

```
 int4 RoundupSize; // = size + 4 and rounded up to next allowable tile size
 int4 signature; // Should equal HELLO_BABE.
                 // Used for checking that malloc blocks are not corrupt.
 int4 size;      // Number of bytes used by an instance of this class
 int4 def_len;   // Length of definition block
 int4 interface_stamp;
 namedobj_node *typedef_obj;// The main name of the structure
 namedobj_node *member;     // Each member of the structure
 void* VirtualFnTable[1];   // For classes
};
typedef *classdef_type;
```

If the user enters in the editor the definition of class 'a': "`class a { public:int x;};`", then that class will have in the system (i.e., in the current *container*) mainly a *namedobj* and a *classdef*. The *namedobj* will store the name of the class, '`a`', a NULL in its '`u.location`' field, while its '`type`' field will have a '{' character (`tp_class`) and a pointer to the *classdef*. That *classdef* will store a pointer to another *namedobj* (the one needed for its member '`x`') in its '`member`' field, and a pointer to the *namedobj* of the class '`a`' in its `classdef_type` field (the already explained one). The *namedobj* for member '`x`' will content a '0' in the '`u.offset`' field (as it's the first (and unique) member), an 'x' in the '`name`' field, and a NULL in the '`next`' field.

If then the user enters in the editor the definition "`a oba;`", creating an object of the class '`a`' declared above, then another *namedobj* is created. In this new *namedobj*, the '`name`' field will content an '`oba`' string. The '`type`' field will content a '{' (`tp_class`) character and a pointer to the *classdef* defined previously for class 'a'. The '`u.location`' field will content a pointer to the place in the heap in which the values for the `oba` object are stored. In this case, that place will be at least of 4 bytes size (only an '`int`' member is declared in class 'a'), and will content zeroes. Note that the fact of the '`u.location`' field being NULL or not is the way to distinguish between *namedobj*'s for classes and *namedobj*'s for objects (in terms of the C++ language).

The set of members is represented as a linked list hanging off '`member`'. All (non-static) data-members have the storage-class '`member_storage`' and the byte-offset is present in the *namedobj->u.offset* field.

The signature field is provided purely as a means for detecting corrupt data. It should always equal the value of the macro HELLO_BABE which is defined through a `#define` clause as the hexadecimal value `e110babe`.

The '`typedef_obj`' is a pointer to the *`namedobj`* which defined this class. E.g. if programmer types '`class C { … }`' then there will be a *`namedobj`* metaclass called '`C`' defined in the current directory, with storage-class '`typedef_storage`', and this is what the `classdef` will point to via the '`typedef_obj`' field. If the programmer doesn't define a class name, then it constructs a class name for itself by appending a quote '' mark to the end of the name of the first declarator declared by this class. Note that the C++ syntax guarantees that there's always either a class name or declarator in any single declaration (Ellis & Stroustrup, 1991).

So far, during compilation normally the system starts with the typedef object and works forwards to the *classdef*, but at other times pointers can be followed in the reverse direction, for example, while trying to print in ascii format the type of a given `namedobj`.

The '*VirtualFns*' field is a pointer to an array of virtual function pointers. This array is a dynamic array, whose size is given by a 4-byte integer occupying the first 4 bytes of the memory block. During the compilation of a declaration, it is a normal Barbados dynamic array, i.e. it is potentially reallocated with each new addition, but after a *classdef* is completely compiled it takes on a different format: the pointer itself becomes the size field (reinterpret it as an integer), and the functions follow straight on from the end of the `classdef`. This rather complicated scenario was designed to make virtual function calls as efficient as possible by minimising the number of dereferences required. (They are equally as efficient as in traditional C++ implementations).

While a `classdef` is being compiled, the `namedobj`'s are scattered about in memory. However, the final step in the compilation of a class definition is to consolidate all memory required by the classdef *into a single heap tile*. In other words, we flatten the linked list of members and the virtual function table and allocate them all in a single 'malloc' block. The reason for this rather complicated format is to make memory management as easy as possible. The length of this *malloc* block is given by the '`def_len`' field.

```
// The FindPtrs() function, which finds the pointers to be swizzled

void **FindPtrs(classdef_node *x)
{
            void **dev, **aux, **aux2;
            int items = 0;
            obj_node members = x->member;

            // Add the pointer to the obj_node of the info of
            // the struct to the list
            dev           = (void **) malloc(sizeof(void *));
            *dev          = &x->typedef_obj;
            ++items;

            // Add each member of the struct to the list
            while (members!=NULL)
            {
                        dev = realloc(sizeof(void *)*(++items));
                        *(dev+(items-1))= member;

                        if ((aux=aux2=FindPtrs(member))!=NULL)
                        {
                                    do
                                    {
                                      dev = realloc(
                                              sizeof(void *)*(++items)
                                      );
                                      *(dev+(items-1))= *aux2;

                                      ++aux2;
                                    } while(*aux2);

                                    delete aux;

                        }
                        ++members;
            }

            dev = realloc(sizeof(void *)*(++items));
            *(dev+(items-1))= NULL;

            return dev;
}
```

Finding pointers is a highly recursive activity. For example, let's suppose we are trying to

find pointers in the object "foo" as declared below:

```
class C { public: int x,y; char* s; };
class B { public: C c; C d; } foo[4][3];
```

The function to find pointers in an object given a pointer to the start of the object and type_type is:

```
FindPtrs(void* v, type_type type, PointerProcessor_fn);
```

This is closely tied with the following function which finds pointers in an object given the classdef object:

```
FindPtrs(void* v, classdef_type type, PointerProcessor_fn);
```

So we first call `FindPtrs(&foo, typeof(foo), fn);` which calls `FindPtrs(&foo[i], typeof(B[3])` four times `(for i=0 to 3)`. Each of these calls will call `FindPtrs(&foo[i][j], typeof(B), fn)` three times `(for j=0 to 2)`. Each of these calls will call `FindPtrs(&foo[i][j], B-classdef, fn)`. Each of these will call `FindPtrs(&foo[i][j].c, typeof(C), fn)` and `FindPtrs(&foo[i][j].d, typeof(C), fn)`. Each of these will call `FindPtrs(cp, C-class, fn)`. Each of these will call `FindPtrs(?, typeof(?), fn)` for each member of `class C`. The integer members will return immediately, whereas the member '*s*' will cause `FindPtrs(?, typeof(char*), fn)` to be called. This function returns immediately and thereby the recursion bottoms out.

2.4.3.5 Functions (compiled code)

These are the nominal functions and methods (member functions) that can be found in C++. In Barbados, the code emitted from the C++ source language is primitive for the Intel[5] processor. There is not an intermediate language running on Barbados interpreter: the code is directly executed by the processor (Cooper, 1997). This leads to efficient execution times, but also supposes a loss in flexibility. One of the problems present here are how to locate pointers into executable code. The only way to do it is to store the addresses of pointers into a table that can be consulted in order to find what pointers to swizzle. So we need to prepend a pointer table in order to be able to locate all the pointers into compiled code (these information will be needed by the system during any swizzling process). This pointer table will be a structure holding the pointers into compiled code, just a NULL-ended, simple vector. Also, we need to distinguish the case of when an object into this container is referenced and when a "*foreign*" object, into another container is referenced. Fortunately, this is transparent for the user, except for the fact that not any address into another container can be referenced, but only addresses in which named objects (i.e., *interface objects*) are found. This guarantees that we present a type-safe model, while we allow one object into a container to reference foreign objects into another container at the same time. Of course, we need another *type-safe* checkpoint which will be presented later: the one which is done when saving a container to disk, consisting in a reachability verification.

However, a complication arises here. In compiled code (for *Intel Pentium* processors), the most common form of the CALL instruction involves a relative jump, i.e. the CALL has an operand which specifies how many bytes to *add* (or subtract) from the program counter register. These self-relative pointers are not found anywhere else in the Barbados type-system. In fact, self-relative pointers are not a normal part of the C++ language. Self-relative pointers must be processed using

[5] ® Intel Corp.

special functions.

The JMP (jump) instructions and conditional jumps all use relative jumps, however JMP instructions always jump to a point within the same function, and therefore the same tile. By comparison, CALL instructions almost always jump to other functions and therefore other tiles. This means that the JMP instructions do *not* need to be processed at all.

However, the CALL instructions *do* need to be processed. The *Saver* needs to map tiles in one address space which are arranged one way to tiles in another address space which will often have a completely different arrangement. The *Loader* can safely ignore CALL instructions into the same container, because here the self-relative nature of the pointers actually helps us avoid work; however pointers into other containers (e.g. into the `/Common/STD` container) will need to be taken into consideration.

```
struct pointer_table {
            void        ** ptr;        // This is NULL-ended list
};

typedef struct funcblock_node {       // The type for functions
            pointer_table pointers;
            void *entrypoint;         // Pointer to the entry point of the function
} *funcblock_type;

// The FindPtrs() function, upgraded, in order to find pointers for functions
void **FindPtrsInFuncBlocks(funcblock_type x)
{
            …
            // Run over the list of pointers
            …
}
```

2.4.3.6 Metaclasses

When dealing with `metaclasses`, only the pointers involved within them must be taken into consideration (it is not necessary to follow pointers). For example, *any* is a tuple which has a type (*type*) and a value. Nothing else is needed, apart from taking into account that pointers to type's, pointers to names and pointers to objects that must be swizzled, too. So, in practice, they are treated not as special cases, but as ordinary applications of the C++ type system.

Representations of metaclasses (*type, directory, any, named_obj*) follow:

```
typedef unsigned char * type_type;              // type metaclass

struct direc_node {                             // a container with a collection of
                                                // named_objs.
 struct obj_node        *root;                  // The root object of the
                                                // directory-container
 struct direc_node      *parent;                // directory metaclass of
                                                // the parent directory
 container_id           parent_container;       // id of the parent
                                                // directory-container
 Conim                  *conim;                 // The container itself
};
typedef direc_node *directory_type;


struct any {                                    // the pair (type, object)
 type_type type;                                // it's similar to the void *, but
                                                // maintining the type
```

```
                                                    // information
    void * value;                                   // i.e., infinite union of types
                                                    // into the system
};
```

Creation of their respective `FindPtrs()` overloaded function is very easy, because the pointers that are to be swizzled are fixed. For example, here is shown the `FindPtrs()` for a `directory_type`:

```
void **FindPtrs(directory_type x)
{
            void **dev;

            dev = realloc(sizeof(void *)*4);

            // pointer to the root
            *dev = &(x->root);

            // pointer to the parent directory
            *(dev+1) = &(x->parent);

            // pointer to the container in memory
            *(dev+2) = &(x->Conim);

            // Trailing NULL
            *(dev+(items-1))= NULL;

            return dev;
}
```

2.4.3.7 The Standard Library

The C++ Standard Library, as has been seen, must exist in the Barbados Persistent Store, in order to allow programmers to use the standard resources of C and C++ (Stroustrup, 1991) for their programs.

The Barbados C++ standard library has four main parts (the C library has its modules named with a lowercase 'c' as prefix (Allison, 1998)):

1. built-in wrapper commands designed to deal with containers.
2. the standard C library (`cstring`, `cctype`, `cstdio`, `cstdlib`, `cmath`, `ccomplex`, …).
3. the proper standard C++ library (`fstream`, `iostream`, `complex`, …)
4. the STL library (`string`, `map`, `vector`, `algorithm` …).

Therefore, the standard Barbados library will consist of a set of containers under the /STD directory container, each one called as the part of the standard library that it is representing (i.e., the same name as the `#include` in normal C++ implementations). The standard library is currently a container not linked to the hierarchy of directories. A reference can be obtained to it through a call to the Barbados API function `StdLib()`.

2.4.3.8 Using the standard library

The standard library is used in ANSI C++, by the inclusion of a header file, which includes

the prototypes for its functions and classes. Also, a 'using' clause must be used in order to get access to the *std* namespace. It also includes a linkage with the standard library, but this is done by default.

In Barbados, this is resolved by referring to the appropriate container through a pointer, or by accessing a concrete function or type, using paths. The using clause will not be useful in Barbados, but it will be compiled without effect (except when used for classes (Elis & Stroustrup, 1991)). An example is shown right below:

```
...
directory &io = /common/std/cstdio;  // Now the contents of cstdio can be referred
io/printf("Hello, World!");          // Also possible.
...
if (/common/std/cchtype/isalpha(ch)) // Another way to achieve the same
                                     // functionality.
{
            printf("Hello, again !");
}
...
```

2.4.3.9 Summary

The Barbados Persistent System is a C++ development environment plus some extra additions in order to manage an underlying Persistent Store. The representation of the C++ types has been shown. The normal C++ objects are called FGO's and are stored into containers, which the programmer must create, open and close, distributing the objects among them.

When saving and loading containers to/from disk, we will need to do swizzling (as will be explained later), and therefore we need to be able to locate the pointers into every data structure.

Programmer's use of the concept of containers is hidden by other common operations, making him believe that he or she is navigating inside a Persistent Store divided into directories. This way, although this container-based (Cooper & Wise, 1996) approach is considered as a loss of orthogonality (Atkinson & Morrison, 1995), the way it is hidden under this directory abstraction partially solves the problem.

2.4.4 Container Management Layer

This layer is concerned with containers at a low level, although containers are mainly composed by *heaps*[6], and *heaps* are managed by the underlying level. It manages the containers in memory, and supports directly the container commands, `OpenContainer`(), `CloseContainer`(), `CreateContainer`(), and `DeleteContainer`(). When the Barbados' programmer executes an `OpenContainer`() command, for example, this layer must be capable to find the container by calls to the *container-tree-id* layer and load it in memory by calls to the underlying layer *Conim*. The containers that the Conim manager deals with, are represented by objects of the `Conim` class. Objects of the `Conim` class (Containers In Memory) are little more than heaps[6]. This layer also deals with relations among containers.

Containers are central in the model of Barbados, as well as in the model of Grasshopper (Dearle *et al.*, 1993), although there are a lot of differences between the two systems. For example, Grashopper is an OOOS, without any explicit support for fine grained objects (suitable programs such as compilers or developing environments must be created on its top before starting working with Grasshopper), while Barbados is a complete persistent C++ programming environment. Another example of differences between Barbados and Grasshopper is that, in Grasshopper, containers can map other containers entirely in their space address, in contrast to the limited communication which is allowed in Barbados. Containers role in Barbados is twofold: on one hand, they act as a clustering mechanism (Sousa & Alves, 1994), and on the other hand, they organise the PS in a hierarchy of directories (García Perez-Schofield *et al.*, 2001b).

Another system with a similar model is PerDis (Shapiro *et al.*, 2000), which uses clusters, so therefore the user must choose in which cluster he or she wants to store his or her objects. The PerDis system is a middleware, so all functionality is provided through libraries which can be linked to any program.

2.4.4.1 Brief description of the Conim class

A Conim represents any container loaded in *memory* at any time, as shown in figure 9.

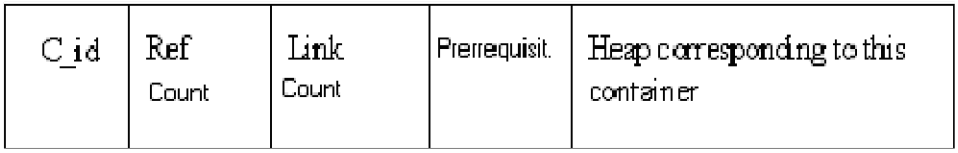

Figure 9. *Diagram showing an aproximate vision of a container in memory.*

Basically, a container_id (c_id), a reference counter and a heap object compose the Conim.

Also, we need to store pointers to the containers that this container depends on (the prerequisites). This all will be explained in detail in the following sections.

2.4.4.2 Interface of the Container-Management Layer

The figure 10 provides a description of how this layer works, in correspondence with other layers.

[6] This is not the usual concept of heap that is present in ANSI/ISO C++. Normally, the heap stores dynamic data, i.e., the kind of data which is stored through **new** and destroyed with **delete**. In this case, we store all data in the heap, and each container has a heap, so we could say that in some way, the heap is the memory available for the container. The terminology of heaps comes from PS-Algol, which is explained in (Atkinson *et al.*, 1982).

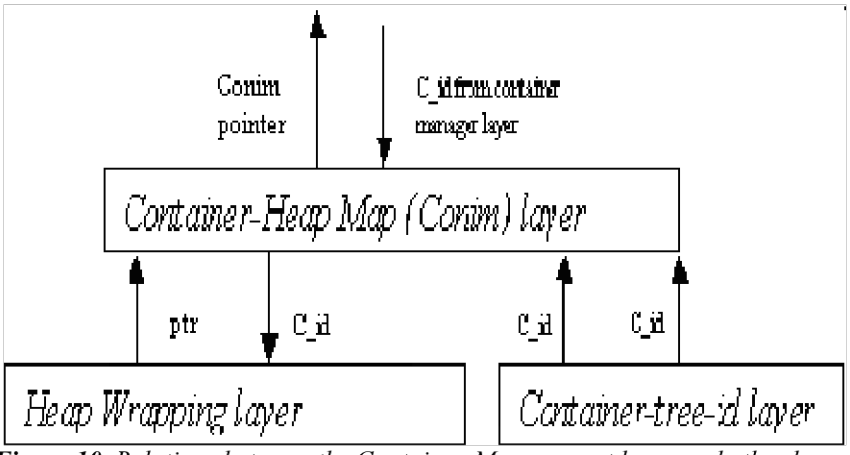

Figure 10. *Relations between the Container-Management layer and other layers*

Each node of the map contains a Conim class, and a pointer to the next node (or NULL if the next one does not exist).

A simplified version of the `Conim` class (the interface of the CONtainer IN Memory Management Layer) is shown below. This class implements concrete operations for each container, as well as layer operations for all containers, as static member functions. For example, the *saver* and the *loader*, central to the persistence part of Barbados, are inside the Conim class.

The important parts of the interface are:

- **Link, Unlink and LinksTo:** Store and retrieve relations among the other containers and this one.
- **ContainersNeedingOfLocalSwizzling:** Holds a list of containers which haven't been swizzled yet in the last load. This happens within the operation of the *loader*. Once the loader is finished, this list is always empty.
- **Prereqinfo:** Holds the complete list of dependencies with other containers. It's the direct load of the CNSwizzling table. Once the loader finishes its operation, this list is always empty (Its information now can be accessed through the Link & Unlink member functions).
- **directory():** It returns the associated root object of this container, which always a directory.
- **getPreferredAddress():** It returns the address in which the container was loaded for the last time, i.e., currently, the starting address for the container in memory. The next time the container is loaded in memory, the system will try to locate it in this address.
- **findNamedObjByPointer():** This is used in the *save* phase. When a pointer is detected to be pointing to another container, then this member function is called in that container in order to find the corresponding `namedobj`.
- **findNamedObjByName():** This is used in the *load* phase. The name of the foreign FGO is retrieved from the CN Swizzling table, along with the c_id of the container in which resides. The container corresponding with that c_id is then searched for an interface FGO of that name.

- **ptr_to_Conim():** determines whether a pointer pertains to this container or not. All open heap segments for these containers must be searched.

A simplifed version of the `conim` class follows:

```
class Conim : public Heap {
private:
  void Link(container_id cid);     // Links a container to another one.
  void UnLink(container_id cid);   // Unlinks a container from another one.
public:
  container_id cid;                // The conim's cid.  0=no conim.
  static
  Conim ** ContainersNeedingOfLocalSwizzling;
                                   // A list of containers which have been loaded with
                                   // the last OpenContainer() call and need to be
                                   // swizzled.
  static
  CNInfoList prereqinfo;           // A list of entries for CN Swizzling info.
                                   // that point to outside this container
                                   // and therefore, will be needed to be C-N Swizzled
                                   // used only while saving

  directory_type directory() const;           // The root object of this container.
  void* getPreferredAddress(void) const;      // The starting address of the
                                              // first segment

  namedobj_type findNamedObjByPointer(void * ptr);
               // Finds a namedobj by its u.location pointer
               // If the namedobj is a class member, then the object owner is
               // filled
  namedobj_type findNamedObjByName(str);
               // Finds a namedobj by its name or uname
  void Protect(bool can_write);
                    // Make the whole heap unwriteable (or writeable again).
  void Close();
                    // Decrement this conim's reference count and perhaps
                    // close it.
  bool LinksTo(Conim *B) const;
                    // Does this Conim have B as a prerequisite?
  bool IsWriteable(void) const;

  bool isInShutdownState();
                   // Is this container being closed ?
  /* Interface functions */
           static Conim* CreateContainer(container_id parent);
                    // Create a new container and conim.

           static Conim* OpenContainer(container_id cid, bool write_permission);
                    // Opens a pre-existing container into memory

           Conim* OpenPrerequisiteContainer(container_id pid);
                    // Opens a pre-requisite container in memory

           static void processForeignPointers();
                    // Finishes all pending CN Swizzling processes.

           static Conim* FindConim(container_id cid);
                    // Do we have this Conim with that c_id already in memory?

           static void RealCloseAll();
                    // Actually close all conims.

           static Conim* Ptr_to_conim(void *mem);
                    // Map this pointer to a Conim.
```

```
        static graph_node* LinkGraph(void);
                    // For diagnostic purposes, output the links as a graph.
}
```

2.4.4.3 Saver/Loader Algorithms

These algorithms are the main algorithms dealing with persistence in Barbados. They have to write the heap of the container on disk and also be able to restore it to memory. These algorithms are in the *Container-Management* Layer as heaps are closely related to containers. A container is basically a heap with a preceding header and a table of C-N Swizzling entries at the very end. The `Conim` class takes the responsibility of saving the heap with the appropriate information, and of loading it in memory, leaving it in an executable state.

The steps in order to load and save a container are the following:

- Saving
    1. Garbage Collection
    2. Compaction
    3. Find the pointers among different containers: C-N Swizzling. Save the table at the end of the container.
- Loading
    1. Load the header of the container; check if the preferred address is available
    2. If it was, then load it in that memory address and continue. In other case, locate it in a free place in memory and swizzle all the pointers in the heap.
    3. Assure that the containers referred by the C-N Swizzle table are in memory, and swizzle all the pointers in the C-N swizzling table.

2.4.4.4 Implementation details of this Layer

There is a static map that holds all the containers in memory. This map relates container_id's (also known as c_id's) and pointers to containers.

Each container has a *preferred_address* or *base_address*. This is the memory address in which the container was loaded the last time (also the memory address provided by the operating system for the first time). In order to prevent corruption problems with the *container_id_tree* table, the preferred address is only stored in the header of the container's associated file. The *preferred_address* is expected to change with the pass of the time for a given container, and once the container is stable (no modifications), is expected to settle with a fixed value and have almost no modifications. The preferred address for a given container changes when the container is going to be loaded in memory (in *write* mode), and its *preferred_address* address is not available. Then the container is moved to a new, available address, and its *preferred_address* updated to this new location.

Related to the *preferred_address* we find out two main problems at this level: firstly a), we need to accommodate in memory a large number of containers, and secondly b), we will need to

arrange the pointers into the data structures in those containers in order to work with them when they are finally loaded in memory (swizzling).

When placing the container in a physical position in virtual memory, we basically adopted the following mechanism: we let `VirtualAlloc()`[7] (note that a similar mechanism to `VirtualAlloc()` (Microsoft, 1998a) can be found in other platforms) to find the space for us, trying firstly the last address at which the container was loaded in memory (the *base_address* in the container's header) for the last time. If this last address is free, then swizzling can be avoided; if not, then it cannot be avoided. In this later case, the `VirtualAlloc()` will find a place, and this will become the new preferred address of the container (provided it is open in write mode, in the other case, it is moved in memory, although this change will not be reflected on disk). Our assumptions are that containers will trend to arrange themselves in memory, and that in a the base address will become fixed after a few loads. So, if one of them clashes, then it will be changed to another address, and probably next time the same set of containers are visited it will not clash. With this scheme, we try to provide an algorithm assuring as less changes in the `base_address` as possible.

The second problem (or b)) is directly related to swizzling: this consists of converting pointers from memory addresses, to disk identifiers (when the container is flattened and stored in disk), and vice versa (when container is loaded in memory). Actually, Barbados does not use in-disk identifiers for pointers (Moss, 1992); it always uses memory pointers, in the hope the container will be always loaded in memory at the same address, and therefore swizzling will be possible to be avoided. If the container can't be loaded in the same place it was the last time it was loaded, then swizzling is necessary, and this is done adding to all pointers in the container the differential (as it is done in DLL's (Microsoft Press, 1996) between last address it was loaded and the new address.

Again, the final objective in this mechanism is to avoid swizzling as much as possible, increasing the overall performance of the system, and converting swizzling to a mechanism as simple as possible.

2.4.4.5 CloseContainer()

When a container is closed, then it must be checked out whether it exists in the set of containers in memory. If it does not exist, an error condition must be reported. Once its Conim object is located, then its `refcount` is decreased by one. If the final reference count is 0, then it will be saved and unloaded from memory. Therefore, the `PhysicallySave`() method of the Conim object is always called in this method.

```
void Conim::Close()
/* Decrease this Conim's reference count. */
{
            if (reference_count <= 0) {
                b_errno = E_NOTOPEN;
                return;
            }
            reference_count--;
            if (reference_count + link_count == 0)
                delete this;
            else if (reference_count == 0)
              {
                if (writeable==READWRITE)
                {
                        ForceSave();            // Save container (PhysicallySave())
```

[7] This Windows API function returns a portion of *virtual memory* for the use of a program.

```
                    protectionUpgrade();    // Because now it's only linked
                }
                LinkGCRequired = yes;
            }

        if (LinkGCRequired)
            LinkGC();
}
```

2.4.4.5.1 CN Swizzling table structure

The CN Swizzling table only exists on disk, once the container has been saved, at the very end of the container. The CN Swizzling table can be located easily: in the header, the `lin_start` field gives the point in the file in which the table starts. The structure of the CN Swizzling table is shown in the table 3 and the figure 11.

Structure of an entry of the CN_Swizzling table

*ptr	The foreign address. With this address, the system can verify if the foreign container is the address it was when this one was loaded. This way, if all involved containers are in the same positions, CN-Swizzling can be avoided.
container_id	The reference of the foreign container.
namedobj's name	The name (as a sequence of characters, the length is given before) of the foreign interface object.
reducedclassdef	The reduced *classdef* of the type of the object. This is used when the foreign FGO is a *classdef*, and the object is located in this container (*TypeClass* relation). This way, both types can be compared and trigger the schema evolution mechanisms can be triggered if needed.
ptr, ptr, ptr, ptr ….	A list of relative pointers to places in this container in which this foreign object is being referenced. This way, we don't need to repeat all this information for each entry.

Table 3. *The structure of the CN Swizzling table (repeating the shown one, until a FFFFFFFF address is found), appended to all containers when saving.*

Foreign-Addrs	Container	Length-name	Rel/Abs/TypClass	Object's name
*ptr	container_id	1x	R/A/C	name

Length-RC	Rclassdef	Length-list	List-of-places
y	*i …	z	ptr,ptr,ptr

Figure 11. Detailed structure of the an entry of the C-N Swizzling for an int.

The purpose of the CN-Swizzling table is to identify the places in the container in which a pointer to a foreign container is found.

Name of relation	***Explanation***
A (*absolute*)	The pointer identifies an absolute address in which the foreign object resides.
R (*relative*)	This pointer must be added to the Program Counter in order to find the location of the foreign object.
C (*TypeClass*)	The object is in this container, while its type definition is in another one. *Classdef*'s are also FGO's which can be referenced legally.

Table 4. *Types of relations available between FGO's of two containers.*

`*ptr` is the address which was valid when the container was in memory, just before saving. The list of pointers (*ptr*, *ptr*, … at least, one pointer), is the list of places within the container being saved in which the `*ptr` address is referenced. `Container_id` and `namedobj`'s name is a meaningful pair of information. They uniquely identify the `namedobj` which is being referenced outside the container, as only `namedobj`'s are legal as CN-Swizzling relations. The name is a string which has a special character as its first position: this special character identifies whether the pointer is an *absolute*, *relative* or *TypeClass* one (as explained in table 4).

ReducedClassdef's are strings which describe classes. They are textual conversions of the type information directly used by Barbados. This information is going to be used when the foreign container is not found, and therefore the class must be rebuilt, and when the types don't match, and schema evolution must be performed. This field is related to schema evolution.

2.4.4.5.2 Saving

Compaction is done before saving, as well as Garbage Collection. This is achieved through copying the heap being saved in another one, in which the tiles to be saved are stored sequentially. Finally, the pointers to functions detected while compaction to point outside the heap being saved, are stored in an appended table.

```
void Conim::PhysicallySave()
…
        /* Get the filename in advance: */
        if (CidToPath(cid, path, sizeof(path))) {
          // Create (if possible) the auxiliary file
          strcpy(auxfilename, path);
          strcat(auxfilename, "$$SAV");
          FileHandle   = CreateFile(auxfilename, GENERIC_WRITE,
                                    0 /* No sharing */,
                                    NULL /* No process inheritance */,
                                    CREATE_ALWAYS,
                                    FILE_ATTRIBUTE_NORMAL,
                                    NULL)
           ;

           // Test if it is correctly created
           if (FileHandle)
```

```
{
 /* Give each tile a saver_node */
 int NumTiles = 0;
 SortSegments();
 Assert();
 for (each_tile)
           NumTiles++;
 size = NumTiles * sizeof(struct saver_node) + 12;
 Savers = (saver_type) ::malloc(anon_heap, size);
 S_idx = 0;
 for (each_tile) {
     Savers[S_idx].offset = (uint)-1;
     Savers[S_idx++].tile = tile;
 }

 /* Recurse on the root tile. Garbage Collection */
 HeaderSize = Heap::RoundUp(sizeof(struct ContainerHeader)) + 4;
 offset = 0;
 Recurse((void**)&header); // FindPtrs-only phase

 /* Create the image: */
 Image = (char*) ::malloc(anon_heap, offset);
 *(uint*)Image = HeaderSize | 2;
 header->lin_start = 0;

 /*
    Here we are deciding wether we are going to put the saved
    container in its current location or not. If it has more than one
    segment, then surely it is going to clash with other container
    in the future, so better we save it in another position
    in memory.
 */
 if (getNumOfAllockedSegments() > 1)
      Base    = Image;
 else Base    = (char *) header->base_address;

 // Create a copy of the current container
 putDelta(Image - Base);
 for (saver=Savers; saver < &Savers[S_idx]; saver++)
  if (saver->offset != (uint)-1) {
     current_saver = saver;
     memcpy(Image + saver->offset, (char*)saver->tile,
```

```
                                        AllocdTileToSize((char*)saver->tile));
        // FindPtrs and swizzling phase
        FindPtrsInTile(Image + saver->offset, WritePtr);
    }

    /* Save the image to disk. */

    // Prepare the size of the Image
    ((ContainerHeader*) (Image+4))->lin_start    = offset;
                // Size without the CNSwizz table
    SetFilePointer(FileHandle, 0, 0, FILE_BEGIN);

    /* Write the Image */
    WriteFile(FileHandle, Image, offset, &numwritten, NULL);
…
```

The file name for the container is retrieved from the *container_id_tree* layer, and then the savers are created for each tile in the heap. `FindPtrsIntile`() is a function used to follow pointers in general: this functionality can be used for garbage collection or for applying a delta to all the image (swizzling, which happens at load time). In the saver, all pointers in tiles are followed, doing garbage collection of the unreachable tiles.

As previously explained, in order to save the container to disk, an image of it is prepared in order to copy within it all objects in the container (performing garbage collection and achieving compaction at the same time). Each object must be within a heap tile, and each tile has a corresponding saver. When any pointer is wrote (all the objects are searched for pointers), it is detected whether this is a foreign pointer or not (understanding that a foreign pointer is a pointer to another container and therefore the C-N Swizzling mechanism will be needed).

Within this approach, when a pointer is found, its position within the image is stored. In the figure 3 (page 22), an example of a pointer which means a CN Swizzling relation is shown. We receive a pointer to a pointer (ptr), because we will probably need to modify it (due to normal swizzling). In this particular case, the pointer is the 'value' field of a `namedobj`, i.e., a metaclass which represents a C++ object. ptr2 represents the offset of the ptr to the image (this works out since the image is compacted). In this example, *ptr is the pointer to the foreign object, and **ptr the location of the data in the other container. Please note that perhaps we could receive a pointer to a foreign container that is not legal, maybe because the pointer is corrupted or perhaps because the pointer refers to an object that it is not one of the interface FGO's into that container. This is easily detected since we need to find the corresponding `namedobj` in the foreign container which is the owner of the data being pointed to (given that we obviously need to find the name of the object in order to identify it).

Follows a simplified algorithm of this part of the process. All pointers are swizzled to the image container which is going to be the container on disk. Also, pointers are check through `WritePtr()` in order to verify whether they point inside this container or into another one.

```
void WritePtr(void** ptr)
```

```
{           saver_type saver;

            saver = PointerToSaver(*ptr);
            if (saver != NULL)      // If we don't have a saver, probably the pointer
                                    // points outside.
           {
              // From what heap the ptr is pointing to ?
              Conim *beingwritten = Conim::Ptr_to_conim(ptr),
                    *foreign;

              // Get the foreign heap (if it exists)
              if (*ptr!=NULL
                 &&ptr!=NULL)
                    foreign      = Conim::Ptr_to_conim(**((void ***) ptr));
              else foreign = NULL;

               // Is the pointer pointing to any location outside
               //  this container ?
              if (foreign     !=NULL
               && beingwritten!=foreign)
              {
                namedobj_node *obj;

                // We must add an entry in the C-N Swizzling list
                // with this pointer
                if (obj=(foreign->findNamedObjByPointer(**((void ***) ptr))))
                {
                      void **ptr2 = (void **) (((char*) ptr)-((int) Image));

                      // It was found, it is a legal CNSwizz node
                      CNInfo entry (foreign->cid, obj, … // fulfil the CNInfo entry
                      prereqinfo.add(entry);
                }
              }

              // Swizzling the pointer in this Container
              *ptr = Base + saver->offset + ((char*)*ptr - (char*)saver->tile);
               }
}
```

In the function above, it can be seen how the pointers are determined to be from what container using the `Ptr_to_conim()` function, as well as we use the `findNamedObjByPointer`() function in order to find the object which its data is being pointed to.

Follows the algorithm for searching *namedobj*'s. It is actually implemented using a mapping, not a sequential search, but this way is more readable:

```
//------------------------------------- findNamedObjByPointer()
namedobj_node *Conim::findNamedObjByPointer(void * ptr)
{
        namedobj_node * toret = NULL;

        // Get the first named_obj
        namedobj_node *actual;

        actual = header->directory.root;

        // Iterate until finding the namedobj_node
        while(actual!=NULL)
        {
                // Is this the namedobj_node we are looking for ?
                if (actual->u.location==ptr)
                {
                        toret = actual;
                        break;
                }
                // Search the next namedobj_node
                actual = actual->next;
        }

        return toret;
}
```

Once we've identified and stored references for all the foreign objects, we need to build a C-N Swizzling table at the end of the container that is being stored. A simplfied version of this algorithm of this stage follows (it continues the `PhysicallySave()` algorithm shown above):

```
…
…
      /* Save the image to disk. */
      assert(FileHandle);

      // Prepare the size of the Image
      ((ContainerHeader*) (Image+4))->lin_start    = offset;
                           // Size without the CNSwizz table
      SetFilePointer(FileHandle, 0, 0, FILE_BEGIN);

      /* Write the Image */
```

```
 WriteFile(FileHandle, Image, offset, &numwritten, NULL);
 if (numwritten < offset)
        b_errno = E_IOFAIL;
 else {
        /*
           Now I put here the dictionary of the C-N Swizzling entries.
           Structure is as explained in the CNInfo class declaration.
           Finally, a TABLE_FINISH_MARK value is appended.
        */
        prereqinfo.deleteRedundantRClassdefs(); // only one RC per class
        for (int i = 0; i < prereqinfo.getNumberOfEntries(); ++i)
        {
            prereqinfo[i]->saveCNInfo(FileHandle);
            Ptr_to_conim(prereqinfo[i]->getNamedobj())->cleanNOHash();
        }

        // Write End of the CN-Swizzling table
        WriteFile(FileHandle, (const void*) &TABLE_FINISH_MARK,
                  sizeof(void*), &numwritten, NULL);
    }

    // We're done !
    // Close the original container's file
    if (FileHandle!=NULL)
    {
      CloseHandle(this->FileHandle);
      this->FileHandle = NULL;
    }
    // Close the auxiliar file
    CloseHandle(FileHandle);
    // Actualise the container
    DeleteFile(path);
    MoveFile(auxfilename, path);
    // Free Memory
    ::free(anon_heap, Savers);
    ::free(anon_heap, Image);
 } else b_errno = E_IOFAIL;
}

// Free allocated resources and finish
prereqinfo.clear();
beingwritten        = NULL;
```

```
        return;
}
```

The algorithm shows how the table is appended at the end of the image being saved. At the header, there are two very important fields: `lin_start` and `base_address`. The first one is the offset from the beginning of the container's file in order to find there the C-N Swizzling table, while the second one is the address the container can be loaded at without doing any swizzling.

The C-N Swizzling table has the structure shown in figure 11 (page 20). The *Rel Pointer* mark has to do with self-relative pointers, which are present in some assembly instructions like `Call xxxx`, in which `xxx` is a self-relative delta between its position in memory and the memory address to be called. *Absolute* pointers are normal pointers to objects. *TypeClass* pointers are pointers to types.

The finish of the table is signalled by an hex value of FFFFFFFF in the 'ptr' field.

2.4.4.6 OpenContainer()

This algorithm has the responsibility of creating the associated Conim object for any container and insert it into the map of Conim's.

The algorithms used here are: `OpenContainer`(), `OpenPrerrequisiteContainer`(), and `processForeignPointerList`(). `OpenContainer`() is called whenever a single container is required to be loaded. The containers related by C-N Swizzling (the prerequisite containers) are loaded using `OpenPrerrequisiteContainer`(). Also, `processForeignPointerList`() is only called at the end, when all the related containers are in memory (although perhaps not swizzled locally: `processForeignPointerList`() must deal with that possibility). This is done this way in order to prevent cycles in the relations among containers. If a cycle were present between **A** and **B** (i.e., each of the two are referencing the other one), and the algorithm were a simple load-explore-the-CN-table-then-fix, then the algorithm would enter into an infinite loop.

When a container is requested, as an identifier like `/Common/STD/string`, then the *Container-Management* layer must check if the container is already in memory. This involves calling `PathToPid()` (in the *container_id_tree layer*) in order to get the container_id, and to check the set of `Conim`'s. If it is in memory, then it adds one to its reference count, If it is not, then it must be loaded into memory, which involves a call to `PhysicallyLoad`(). Note that the `Opencontainer(container_id)` function should call successively `OpenPrerrequisiteContainer(container_id)` in order to load into memory all the prerequisites of that container.

```
//------------------------------------------------------------- OpenContainer()
Conim* Conim::OpenContainer(container_id cid, bool write_permission)
/* Open a pre-existing container into memory */
{
        Conim* conim;

        // Perhaps it's already in memory
        conim = FindConim(cid);
        if (conim == NULL) {
```

```
    // Prepare the information entries
    prereqinfo.clear();
    ContainersNeedingOfLocalSwizzle           = NULL;

    // load it in memory and do basic swizzling
    conim = PhysicallyLoad(cid, write_permission);

    // is all right ?
    if (conim == NULL)
       return NULL;

    // fix all pending CN - Swizzling entries
    processForeignPointerList();

    // Now please swizzle all containers we have loaded during this process
    {
        int i;
        Conim *pqconim;
        for (each_aeli(pqconim, ContainersNeedingOfLocalSwizzle))
        {
                /* Swizzle it: */
                pqconim->localSwizzle();

                /* Is it healthy or sick ? */
                pqconim->Assert();

                // Convert it to READONLY if it is a prerequisite
                if (pqconim != conim)
                        pqconim->SetPagePermissions(READONLY);
        }
    }

    // Cleaning structures for CN-Swizzling which will no longer be used
    prereqinfo.clear();
    Array_Free(ContainersNeedingOfLocalSwizzle);
    ContainersNeedingOfLocalSwizzle = NULL;
    }
else {
        // Handle this request
        if (write_permission==READONLY)
                conim->protectionUpgrade();
        else    conim->protectionDowngrade();
```

```
        }

        // Anyway, increment the reference count for this container
        conim->reference_count++;

        return conim;
}
```

In the above algorithm it can be seen how the different algorithms are related. The call to `PhysicallyLoad()` can result in calls to `OpenPrerequisiteContainer`(), which calls `PhysicallyLoad()` too. Only at the end, the entries collected through all the related containers are managed in `processForeignPointerList`(). All these algorithms are deeply studied below.

The following algorithm is the one in charge of loading the related containers in memory, It is called from `OpenContainer()` each time a related container is detected in the CN Swizzling table.

```
//------------------------------------------------ OpenPrerequisiteContainer()
Conim *Conim::OpenPrerequisiteContainer(container_id cid)
{
        Conim *toret;

        if ((toret=FindConim(cid)) == NULL)
        {
                toret = PhysicallyLoad(cid, READONLY);
                Array_InsP(ContainersNeedingOfLocalSwizzle, toret);
        }

        return toret;
       }
```

2.4.4.6.1 Loading

Before loading the whole container, its header is inspected in order to know when it was placed (its preferred address) the last time it was in memory. Perhaps this location is not free, so we will have to place it in a free place and swizzle all the pointers in the heap in order to left it ready to work. Of course, the first possibility is the preferred one.

Whenever a container is placed in memory, its preferred address must be tested for availability: the preferred address for the new container can be occupied by another container. This happens whenever the '*preferred address*' for two containers are the same and the system attempts to load them in memory (i.e., the first one has been loaded and remains active and the second one is then tried to be loaded).

```
Conim* Conim::PhysicallyLoad(container_id cid, bool writeable)
{
            …
```

```
      // Open the file:
      if (not CidToPath(cid, path, sizeof(path)))
          return NULL;       // CidToPath has already set b_errno
      file = CreateFile(path,
                      GENERIC_READ,
                      FILE_SHARE_READ,
                      NULL,
                      OPEN_EXISTING,
                      FILE_ATTRIBUTE_NORMAL,
                      NULL);
      if (file == NULL or file == INVALID_HANDLE_VALUE) {
          result = GetLastError();
          if (result == ERROR_FILE_NOT_FOUND or result == ERROR_PATH_NOT_FOUND)
                  b_errno = E_NO_SUCH_CONTAINER;
          else if (result == ERROR_SHARING_VIOLATION
                  or result == ERROR_LOCK_VIOLATION)
                  b_errno = E_LOCK;
          else b_errno = E_IOFAIL;
          return NULL;
      }

// Read the header of the container to get info about it
{
   unsigned long int bytesread; // needeed for the ReadFile() function

   ContainerSize = GetFileSize(file, &read_size);
   SetFilePointer(file, 4, NULL, FILE_BEGIN);       // pass the tile header
   if (!ReadFile(file,&header, sizeof(ContainerHeader), &bytesread, NULL))
   {
          CloseHandle(file);
          b_errno = E_IOFAIL;
          return NULL;
   }
   SetFilePointer(file, 0, NULL, FILE_BEGIN);
}

      // Try to locate enough space in its preferred address
      mem = (char*)VirtualAlloc(header.base_address,
                           header.lin_start,
                           MEM_RESERVE|MEM_COMMIT, PAGE_READWRITE);

// If the preferred address didn't work, then
// we should try any other one
if (mem == NULL)
{
             mem = (char*)VirtualAlloc(NULL,
                           header.lin_start,
                           MEM_RESERVE|MEM_COMMIT, PAGE_READWRITE);

          // Perhaps it still didn't work, so let's exit.
              if (mem == NULL) {
                      b_errno = E_NOMEM;
                      CloseHandle(file);
                      return NULL;
              }

        // We want to be sure that pages are only marked as READWRITE
        // if really needed, so we first check the returned address.
        InitialProtection = (mem == header.base_address) ?
                                                        READONLY: READWRITE;
}

// Now we can copy the container in memory
      result = ReadFile(file, mem, header.lin_start, &read_size, NULL);
      if (not result) {
          int result = GetLastError();
          b_errno = E_IOFAIL;
```

```
                CloseHandle(file);
                VirtualFree(mem, 0, MEM_RELEASE);
                return NULL;
            }

        // Copy the CN Swizzling table in memory
        cntable = (char *) ::malloc(anon_heap, ContainerSize - header.lin_start);

        if (cntable != NULL)
                    result = ReadFile(file, cntable, ContainerSize - header.lin_start,
                                                                &read_size, NULL);
            }
…
}
```

The following algorithm is the next part of the `physicallyload()` algorithm, which is used used in order to open the related containers of the container being loaded.

```
…
        // Copy the CN Swizzling table in memory
        cntable = (char *) ::malloc(anon_heap, ContainerSize - header.lin_start);
…
        // Prepare the Conim object
        Base   = Image = mem;
        offset = header.lin_start;
        ((ContainerHeader*)(mem+4))->base_address = mem;

        /* Create the Conim object: */
        conim = new Conim(mem, header.lin_start, cid, InitialProtection, file, NULL);

        /* Set delta if needed */
        conim->putDelta(mem - ((char*)header.base_address));
…
        if ((writeable        == READONLY)
         && (InitialProtection == READONLY))
                  conim->SetPagePermissions(READONLY);
          else conim->writeable = READWRITE;

        // CN Entries are loaded in memory too
…
        {
            CNInfo         *reg;
            Conim          *foreign;
            char           *wr;
            void           *ptr;
            namedobj       *obj;

            // Get ready
            Array_InsP(ContainersNeedingOfLocalSwizzle, conim);
            wr  = cntable;
            obj = *((namedobj **) wr);

            // Reading entries
            while(obj!=TABLE_FINISH_MARK
              and wr < (cntable + (ContainerSize-header.lin_start)))
            {
                  // Read entry
                  reg = CNInfo::loadCNInfo(wr);
                  reg->rebasePtrs(((int) conim->header->base_address));
                  Conim::prereqinfo.addCNInfo(reg);
```

```
                // Link container
                foreign = conim->OpenPrerequisiteContainer(reg->getForeignContainer());
                if (foreign == NULL)
                        reg->markasInvalidContainer();
                else    conim->Link(foreign);

                // Next entry
                obj = *((namedobj **) wr);
          }
        }

        // We're done !
        // Free container table
        ::free(anon_heap, cntable);
        return conim;
}
```

The CN Swizzling table is mapped in memory, and then it is processed, storing the information of each C-N Swizzling entry in a vector of structures. At this stage, only the pointers are swizzled in order to let them in the same state they were when loaded. It should be noted that the fixing is not done at this stage.

Now, the `processForeignPointers`() algorithm is shown. It fixes all pointers, taking into consideration whether they are *absolute*, *relative* or T*ypeClass*:

```
//------------------------------------------------- processForeignPointerList()
void Conim::processForeignPointerList()
…
{
        CNInfo        *rec;
        int i;
        Conim         *foreign;
        namedobj_type where;

        for(int n=0; n<prereqinfo.getNumberOfEntries(); ++n)
        {
              // Process this entry
              rec = prereqinfo[n];

              // Get the foreign container
               if ((foreign=FindConim(rec->getForeignContainer())) != NULL)
              {
                      str tmp = rec->getName();
                      where   = foreign->findNamedObjByName(tmp);
                                 if (where != NULL)
                        {
                                if (!rec->isCorrupted())
                                        // Resolve the pointer
                                        ResolvePtr(*rec, where);
                                else ErrorWarning("Skipped corrupted "
                                                  "C-N Swizzling reference");
                        }
                        else {
                                char buf[PATHLENGTH];
                                strcpy(buf, "Skipped reference to a
                                             "non-existing public object: ");
                                strcat(buf, rec->getName());
                                ErrorWarning(buf);
```

```
                        rec->putPtrsNULL();
                        assert(false);
                }
        }
        else
        {
                rec->putPtrsNULL();
                ErrorWarning("Skipped reference to an unexisting container");
                assert(false);
        }

    }
}
```

Each entry in the vector of CN fixes (the `prereqinfo` static member of `Conim`) needed is managed, and for each pointer, its `namedobj` in its corresponding container is looked for. If it is not found, an error is shown, and if it is found, the pointer is fixed to point to the new location of the `namedobj`.

2.4.4.7 CreateContainer()

The function `NewContainerId`() of the *Container_id_tree* module is called within this function, so a new `container_id` and a new associated container file is created, under the appropriate directories, with a size of zero. The next step is to create an appropriate header for the new container, with an appropriate –empty- root, and then store it in the associated file. Then the `OpenContainer`() function can be called, and the container is located normally in memory.

```
Conim* Conim::CreateContainer(container_id parent)
/* Create a new container and conim.  The new container is           */
/* created with a reference count of 1, ie. you need a matching      */
/* 'CloseContainter()' call to close it.  Return the conim,          */
/* whose fields include the directory and the container_id.          */
{           container_id cid;
            char path[PATHLENGTH];
            Conim *conim;

            /*** Allocate a new container_id: ***/
            cid = NewContainerId(parent);
            if (cid == 0)
                        return NULL;

            /*** Allocate the memory for it: ***/
            conim = new Conim(4096, cid);
            conim->cid = cid;
            if (ContainerHeaderStruct == NULL) {
                static classdef_node tmp_classdef;
                ContainerHeaderStruct = &tmp_classdef;
                ContainerHeaderStruct->RoundupSize =
                    Heap::RoundUp(sizeof(ContainerHeader)) + 4;
```

```
        ContainerHeaderStruct->signature = HELLO_BABE;
    }

    /* Prepare the header of the container */
    conim->header = (ContainerHeader*)conim->New(ContainerHeaderStruct);
    clearS(*conim->header);
    conim->header->dir.parent_cid = parent;
    conim->header->lin_start      = 0;
    conim->reference_count        = 1;
    conim->header->base_address           = conim->getFirstMemorySegment();


    /*** Create an empty file and keep it open: ***/
    if (not CidToPath(cid, path, sizeof(path))) {
        delete conim;
        return NULL;       // CidToPath has already set b_errno
    }

    conim->FileHandle = CreateFile(path,
                    GENERIC_READ|GENERIC_WRITE,
                    FILE_SHARE_READ,
                    NULL,
                    CREATE_ALWAYS,
                    FILE_ATTRIBUTE_NORMAL,
                    NULL);
    if (conim->FileHandle == INVALID_HANDLE_VALUE) {
        conim->FileHandle = NULL;
        int result = GetLastError();
        if (result == ERROR_PATH_NOT_FOUND)
                b_errno = E_CORRUPT;
        else if (result == ERROR_SHARING_VIOLATION
                or result == ERROR_LOCK_VIOLATION)
                b_errno = E_LOCK;
        else b_errno = E_IOFAIL;
        delete conim;
        return NULL;
    }

{
 unsigned long written;
 /* Write the header to the associated file */
 WriteFile(conim->FileHandle, conim->header, sizeof(ContainerHeader),
```

```
                                        &written, NULL);
        }

        /* Go to the right position */
        SetFilePointer(conim->FileHandle,0, 0, FILE_BEGIN);


            return conim;
}
```

2.4.4.8 DeleteContainer()

The function `DeleteContainerId`() of the *Container_id_tree* module is called within this function, so its `container_id` becomes available and the container associated file is deleted. The container cannot be in memory, i.e., cannot be in the map structure of containers in memory. This would mean it would be opened, and therefore it would be a non-sense to delete it. So, if the container is in memory, it must be closed and then deleted.

```
bool DeleteContainer(container_id cid)
/* Delete this container. The container must not be open. */
/* We delete the file from disk and the id from the tree-table. */
{           char path[PATHLENGTH];

            if (FindConim(cid))
                return no;
            if (not DeleteContainerId(cid))
                return no;
            CidToPath(cid, path, sizeof(path));
            DeleteFile(path);
            return yes;
}
```

2.4.4.9 Relations among containers

Containers can relate to each other, in two main ways: by ownership relation and by link relation (the nature of this two kind of relations has been already discussed). The former one has to do with the natural relations among directories: a father directory and its children.

The later one is of a completely different nature: two or more containers are related because one of them *uses* objects of another one. The purpose of this is to have a mechanism similar to libraries: for example, the /*Common* container holds objects that are of general interest (although of course this mechanism is not only limited to the /Common container), that can be used by any other container. This has been discussed as C-N Swizzling.

Of course this data sharing capability is limited and restricted: in other way, it would be a non-sense to divide the Persistent Store in Containers (because in the end, we could have all containers related among them). Only the FGO's that are in the main directory of a a container

(which could be called the *public* objects or the *interface* objects) can be referenced from within another one.

This prevents the Persistent Store of becoming corrupted. Any possible error is isolated in one container, and although a container can have references to others, the nature of these references, and the nature of containers don't allow programmers to corrupt more than the container they are working with. This is assured by the way the containers which are related are opened. It should be noted that containers can be opened in read or read/write mode. All the containers opened through the C-N Swizzling mechanism are in memory in read-only mode.

The following functions can not be directly called from within a Barbados program, or at least they don't have a direct mapping with actions the user can do. They are wrapped in the operations involved when, for example, another container is referenced within another container and any object of the later one is used. This happens because this linking is transparent, wrapped by the C-N Swizzling mechanism.

2.4.4.9.1 LinkContainer()

This function has a direct relation with C-N swizzling. Basically, it's used when an FGO in a container points to an FGO (i.e., an object) stored in another container. The C-N Swizzling table of the "source" container therefore stores a new entry with a (*container_id, name*) pair, and the link count of the `Conim` associated to the container where the object needed is located is increased in 1. This will avoid the accidental closing of a container that is actually being used. The call to `Link()` is done immediately after to the call to `OpenPrerequisiteContainer`().

```
void Conim::Link(Conim *B)
/* Create a link from A to B, saying that A depends on B. */
{
        if (  B != NULL
          && !Array_HasP(this->prerequisites, B))
        {
                    B->link_count++;
                    Array_Add(this->prerequisites, B);
        }
}
```

2.4.4.9.2 UnLinkContainer()

Actually, this call does exactly the inverse action of the above one, decreasing the reference count of the other container. Perhaps we can remove from memory some other Containers that were linked to this one, so we launch the Container Garbage Collector in order to deal with them.

```
void Unlink(Conim *B)
/* Remove the link from A to B. */
{
            if  (B != NULL
              && Array_HasP(prerequisites, B))
            {
                        Array_DelP(prerequisites, B);
                        if ( --B->link_count <= 0
                         and B->reference_count == 0
                         and not(B->isInShutdownState()))
                                    B->RealClose();
                        else { LinkGCRequired = yes; LinkGC(); }
            }
}
```

2.4.4.9.3 LinkGC()

This algorithm copes with cycle references among containers. When the `link_count` and `reference_count` are not 0, then this means that the container cannot be unload from memory and its space be used for another container. The only way to have these fields reaching 0 (therefore giving the possibility to substitute them), is by the `LinkGC()` algorithm, or by the `CloseContainer()` one in the `Conim` class (commented above).

Definition. "A container x has a cycle in references with another container y, iif the set of links to prerrequesites containers for x contains the container y and the set of links to prerrequesites containers for y contains the container x".

This algorithm is only executed when the `UnLink()` algorithm is called, because a container has been closed. It has to iterate through all the containers in memory (`Conim` class, the containers present in the *hash* structure), trying to find al reachable containers: the remaining ones are orphans. Though the algorithm is very simple, its complexity is of *O(n)* being 'n' the number of containers in memory.

If we have a cycle in the link relation of containers, then we have a set of containers that reference each other. Such a set acts in some ways like a single container e.g. they enter and leave virtual memory together, but in the remaining ways they are different. Each container stores the counting reference (i.e., the number of containers that reference it, and the list of containers that are linked (depend on) this container. The garbage collection of containers is of value to remove cycles of containers that are not needed by a container but are still in memory, though its place is needed, because they depend on each other.

```
void Conim::LinkGC()                // A static function
/* Garbage collection of linked conim's. */
/* To do the garbage collection, we regenerate all the 'link_counts'  */
/* based on what containers have OpenContainer()-style reference-counts,*/
/* and purge any that are now orphans. */
{       Conim *conim, **Orphans;
            ConimIterator iter;
            unsigned int i;

            /* Reset the link_counts: */
            for (each_conim) {
                conim->link_count = 0;
                conim->flags &= ~REACHABLE;
            }

            /* Regenerate the link counts: */
            for (each_conim)
                if (conim->reference_count)
                        conim->FindReachable();


            /* Purge all the new orphans: */
            Orphans = NULL;
            for (each_conim) {
                        if (not (conim->flags & REACHABLE))
                        Array_Add(Orphans, conim);
            }
            for (each_aeli(conim, Orphans)) {
                        Array_Free(conim->prerequisites);
                        // These links have not been counted in the 'link_count's,
                        // so we want to avoid the normal process of decrementing
                        // a container's prerequisites' link_counts on destruction.
                        delete conim;
```

```
            }
            LinkGCRequired = no;
}
```

2.4.4.10 Summary

The container commands for the programmer have direct correspondence with the interface functions in this layer. This layer must be then able to create, delete, open and close containers in memory: these are the needs for each individual *container*. So it also manages auxiliar needs such as Garbage Collection of Containers, C-N Swizzling, Linking … etc.

The layer must organise the containers in memory in some way: this is done through a simple map on the preferred address stored in the *container_id_tree* layer for every existing container.

Containers in memory are represented by instances of a class called *Conim*, which is the basic piece of this *Container-Management* layer, and they act as the clustering masterpiece. Also, they provide a directory-based abstraction, allowing the user to specify (indirectly) the most appropriate cluster strategy in each moment. This is more convenient as other automatic clustering techniques are quite expensive in execution time (Darmon *et al.*, 2000; Tsangaris *et al.*, 1992).

The format on disk consists on the first tile always being the container header tile, and having a C-N Swizzling table at the very end.

The next tile to the header is the root tile; it contains the root object of the container: an instance of the directory metaclass, from which we can reach all remaining objects in the container.

The representation of a container on disk is designed to allow the container to be directly mapped into memory with no pointer swizzling or other modifications whatsoever, as often as possible.

`LinkGC`() is designed in order to find containers that reference each other. This becomes a cycle, causing the `reference_counter` never reaching 0, and therefore preventing `CloseContainer`() to remove it from memory.

Swizzling consists of arranging pointers in the container's heap when the container on disk cannot be located into the address that its preferred address signals as its last location. All pointers in the container must be swizzled with a *delta* (preferred - actual) of address.

Containers can reference each other because of the needs of the programmer of using information that exists in "*remote*" containers, a different container of the one he is working at. This causes the container to be linked with another one. When this happens, this layer must take these cases into account, incrementing references, opening the prerequisite containers in read-only mode and so on. This supposes to have two different kinds of swizzling: firstly, local swizzling, or swizzling inside the container, which is achieved with the use of a *delta* added to all pointers (Moss, 1992), instead of using hardware-based techniques (Narasayva & Ng, 1996). And secondly, C-N Swizzling, a novel kind of swizzling which happens among containers.

The swizzling techniques used here do not make use of OID's instead of pointers as in other systems (for example Branath et al., 1998), excepting the C-N Swizzling mechanism, which is a combination of the *container_id* and the name of the object, which represents uniquely any object (although this is not *exactly* a "traditional" OID).

2.4.5 Heap Primitives

The heap primitives are just the normal primitives of `malloc()` and `free()`. There is not a different Heap for C and C++ operators, simply the 'new' operator wraps the behavior of the 'malloc()' function.

```
char *astring = new char[40];
```

is equivalent, from the user's point of view, to:

```
char* astring = (char *) malloc(sizeof(char)*40);
```

… or vice versa. Actually, these two examples are not exactly equivalent, as the first one will make the compiler create a tile with type (`char` type), while the second one will suppose an untyped tile. Tiles with type are central to the model of persistence of Barbados.

The normal behavior of the *new* operator is to also call the constructor of the type after allocating the space. The 'new' operator is in fact more useful than the 'malloc()' function, from the Barbados point of view: it informs of the type the user wants to use in this new allocated space. Anyway, the user is able to do the following, breaking the type model and compromising persistence:

```
class a { public: int x;};
a * oba = (a *) malloc(sizeof(a));
```

Instead of,

```
a * oba = new a;
```

Really, this is not very common when programming applications in C/C++ (although it must be allowed). For Barbados, some of these mechanisms can be investigated in order to know what will be the use of this space. But many memory-management mechanisms that the user is able to design will not work with Barbados' persistence mechanisms, as we can't cover all cases without restricting the user's programming possibilities.

2.4.5.1 Implementation

Every memory block in the heap is expected to point to the type that is being stored. This approach has been taken because it's difficult to infer the type of the block when saving the container (that is, when the persistence needs to be guaranted). In fact, Barbados needs to be type-safe when saving, so the C++ non-type-safe features only can be done among savings of the container.

So, the heap primitives are: `malloc()`, `free()`, and `realloc()`, which manage a heap space in every container of the current process. The heap space is obtained through a call to the `VirtualAlloc()` Win API function (Microsoft Press, 1996).

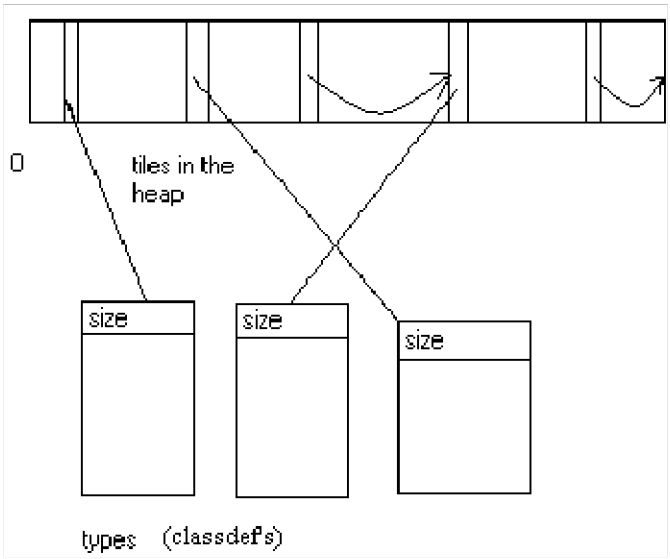

Figure 12. *An example of a sequence of tiles in a heap. Each tile corresponds to an individual object (in the general sense). Simple data entities (such as* int*) store their size in the header of the tile. The rest point to their* classdefs*, and their size is stored there.*

An example is shown in figure 12. Tiles are stored in segments, and a set of segments compose a heap. Segments are managed transparently by the system (concretely, by this layer). The ideal situation would be the segments to be stored continuously, but this is not possible many times because the variations in size of the container must be supported dynamically. Anyway, at the saving stage, all segments are stored one after the other one, in a single block.

2.4.5.2 Heap Class

As its name suggests, this class only wraps the heap primitives and basic functionality, such as segment management. This is needed in order to encapsulate the heap in the `Conim` class (as shown in figure 13).

The heap will be a file mapped in memory, with room enough to store the data that the container is managing.

Follows a simplified version of the Heap class:

```
class Heap {                                        // This represents a heap.

/* HeapSegments: interfacing with virtual memory */
  private:
struct HeapSegment {    // A contiguous set of pages
char* mem;      // The start of the segment
uint numbytes;              // How many bytes in the segment? Must be a multiple of
                  // 4096.
};

HeapSegment *allocked_segments;     // Corresponds to calls to VirtualAlloc
HeapSegment *coalesced_segments;    // Corresponds to maximal unbroken segments
static PageHashNode ** PageHash;    // A hash-table mapping pages to heaps.
static uint NumPages;               // How many elements in PageHash?
```

```
void PageMapAddSegment(void* mem, uint size);
            // Add this segment to the PageHash.
void PageMapDelSegment(void* mem, uint size);
            // Remove this segment from the PageHash.

  public:
bool NewSegment(uint numbytes);    // Create a new HeapSegment for this heap.
void FreeAllSegments();            // Return memory to the v memory manager

/* tile-level basic handling of the heap */
  private:
tile_type freechain[HEAP_NUMPOWERS];
void FreeTile(char* p, uint size);
void MergeTiles();
void* RawMalloc(uint size, uint h, uint header);
  public:
/* The normal malloc functions. */
void* malloc(uint size);
void* realloc(void* p, uint size);
void* calloc(uint size, uint n);
str strdup(str s);
void free(void* p);
void* New(classdef_type structure);
void* New(type_type typestr);
uint memused(void);    // How many bytes overall have we used?
/* Checking for corruption in the heap. */
bool Assert(void);
/* Querying the type information in tiles.
   These functions apply for pointers into tiles . */
static bool IsTyped(void* ptr) { return TileIsTyped((char*)ptr - 4); }
static bool IsClass(void* ptr) { return IsClass((char*)ptr - 4); }
static type_type Typestr(void* ptr, char dest[5]);
}
```

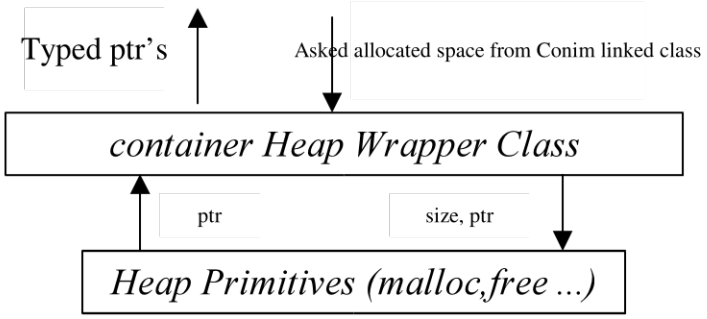

Figure 13. *Relations among the involved layers*

Every tile into the heap has in its header a pointer to the type_node allocated into it, or to the `classdef` of the type of the data entity stored in that block. The first type of tiles, which are storing non-class objects, have a place inside itself to place the type, and the pointer in the header points there.

With this mechanism objects are related directly with its type allowing us to identify each C++ data with its type, accessing memory in a very efficient way.

2.4.5.3 Heap Primitives: detailed implementation

This heap layer is the module which manages fine-grained memory allocation: i.e. the `malloc()/realloc()/free()/new/delete` family.

The system can infer which heap (container) any pointer (user or system ones), is pointing to

because of the space address. This way, the primitives are related exactly to one of the heaps that are open, while in the other way the primitives should reference the heap that is being dealt (even this is admitted at high level, at the user's functions, although it is not mandatory).

Note that there always will be a special heap, related to the kernel: The purpose of this heap is not to be persistent, because the kernel structures and temporals are not needed to persist.

In Barbados an optimised algorithm is used in order to manage memory for processes. It has been specially tuned to work fine under virtual memory swapping conditions, which happens usually in persistent systems. It is perhaps counter-intuitive that it would use less memory, because in fact it tries to optimise speed over memory usage and thereby it tries less hard than traditional algorithms to use memory efficiently. In fact, it rounds all heap tiles up to a multiple of 2 or 50% more than a multiple of 2, with the potential that the unfilled portion at the end is wasted. Another property of this algorithm is that it doesn't try to merge free blocks together upon *free* operations: instead it waits until all free-chains are exhausted, and then just prior to asking for more pages from the OS, it does a forward-match through all free tiles looking for adjacent free tiles that can be merged. Only if this fails to return a free tile of the required size, (or if we have done this time-consuming algorithm too recently), does it get more pages from the OS.

This discussion is important since as a consequence of this algorithm, the header of each tile need only store the length of the tile. No other information is needed (traditionally, the header stores a forward length and a backwards length). This means the header is only 4 bytes in size (this emphasise on low memory usage is about improving the performance of on-chip caches more-so than reducing the need for swapping in main memory).

Furthermore, the heap is customised to the C++ type-system. We don't store the size of the tile in the header, instead we store a pointer to the class-definition of which this object is a member. The size of the tile is then the very first 4-byte word in this class-definition object. It means that to find the length of a tile requires an extra dereference, but this is not important given the benefits this representation gives us.

The heap must also cope with untyped tiles, e.g. what we get if we call '`malloc(57)`', and tiles which have types but are not class instances, e.g. what we get if we call '`new C [57]`'. Untyped tiles have their length given by having bit 1 of the header word set, (bit 0 being on is used to denote free tiles), so in this case the system subtracts 2 from the value to get the true size. Tiles which are typed but are not class instances are subject to a more complex system: the header points into the block to a length word, and the type follows the length word as a type_type.

Also, we will need to find pointers at tile level. This is because heap tiles generally contain pointers in the header/trailer and we need to process these pointers too. The following function is defined in the file mem_heap.cpp:

```
FindPtrsInTile(tile_type tile, PointerProcessor_fn);
```

Note that 'tile' is a pointer to the start of the tile, i.e. 4 bytes before the address which is passed to users.

Note also that this function needs to process each of the tile formats as described above: it needs to ignore free tiles and untyped tiles; and it needs to process class instances and non-class-instance typed tiles. Non-class-instance typed tiles can have additional pointers in the type_type section which is at the end of the tile.

2.4.5.4 Mapping the data structures from/to disk

This is the most important part of the design of the persistence mechanism for Barbados; the `Conim` class (the mapping between containers and Heaps, the containers in memory, figure 14) is the one which manages the loading and saving of the containers. And this loading/saving (transparent for the user) is which provides persistence to Barbados.

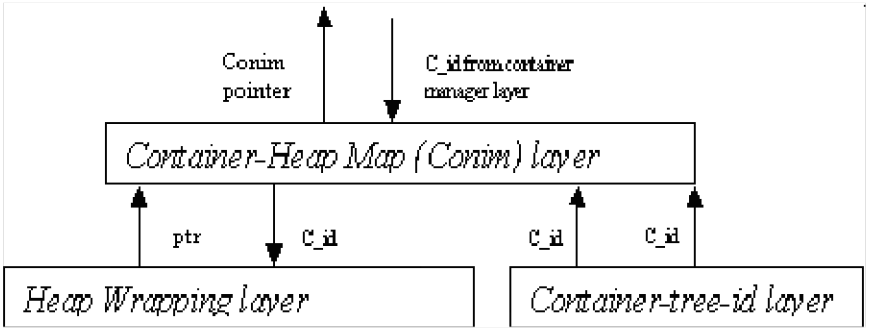

Figure 14. *Relations among the related layers*

The information stored in a container is a heap. The heap has the normal usage as in other compilers (i.e., to allocate memory for FGOs), but in order to acquire persistence we put all the remaining data (globals, for example) there. Therefore, when the container is closed and the mechanisms of persistence are to be used, we have got all information gathered into a single place.

The main matter is how to map the contents of a heap into disk. A container on disk is in the format of a Barbados heap (in fact a single continuous heap segment, as compared with containers in memory which can consist of multiple segments).

Following the format chosen, the first tile is always the container header tile. Although it is a header object, it conforms to the above description of a heap tile i.e. it has its own 4-byte header in the usual format of a tile header.

This object contains an '`any_node`' which we can use to get the location of the root object (which is always a `directory`).

The representation of a container on disk is designed to allow the container to be directly mapped into memory with no pointer swizzling or other modifications whatsoever, as often as possible.

The following problems must be addressed:

a) Pointers between nodes. A structure node for example has a pointer to its type, and the types of all its members. Perhaps some of these members are also pointers ... etc. These pointers must be swizzled every time the container is loaded into memory, in order to assure its correctness when used, if we do not use any special mechanism. This problem is solved with the `FindPtrs`() functions, seen at the compiler layer.

b) Function code. This kind of data will be stored as compiled code: no intermediate code is used. It is necessary to put after the code a table containing the positions in the compiled code of the pointers used in function. This is because the Pentium instructions do not contain sufficient information to determine which "immediate

operands" are pointers and which are not, so the compiler must store this information somewhere. An optimisation is made for CALL instructions however, whereby the destination operand is identified as a (self-relative) pointer by virtue of the instruction op-code alone, which substantially reduces the size of this table.

c) Pointers between Linked containers in function code. The pointers into compiled code need to be stored as explained above, denoting when the pointer points to the same container (the most common case, the same Heap) or to another container (linked, via C-N swizzling).

d) Garbage collection must be done before writing the entire container to disk. It will be done by reachability, following recursively the pointers from the root object and the remaining objects of the container. Orphaned objects will be discarded as garbage. A special optimization must be used in order to detect linked lists, because of the danger of overflowing the stack while running through it.

e) Pointers can point to a field into an object, and not to the head of the object (returned by operator `new`). This detail must be taken into account in order to implement garbage collection.

Some of the problems related in the list above have been solved in previous sections. The solution to the remaining ones can be found in the next sections.

2.4.5.5 Garbage Collector and Compaction Algorithm

These processes happen both at saving time, inside the Heap (it must not be confused with garbage collection in the *Container Management* layer).

The garbage collector algorithm has the objective of determining which objects in memory (i.e., in the current heap) must be made persistent and which not. This distinction is made by the reachability of that objects from the root of the container. If an object exists but is not reachable, then this object was the result of a temporal processing and does not need to be persistent. For example, in a directory container, the root is the directory metaclass. The root points to `namedobj`'s, i.e. these `namedobj`'s are reachable from the root. Each `namedobj` can have pointers to another objects into the container. Persistence is guaranteed to these objects by applying transitivity from the root of the container.

Please note that this is an object-grained garbage collection, while there is another one based on containers, which was explained in a previous section. They are completely orthogonal.

Once Garbage Collection is done, a new heap is created, in which the tiles are copied sequentially. This way, it is guaranteed that the heap is compacted.

The algorithm in charge of doing these tasks is embebbed in the `PhysicallySave()` one. It firstly creates a new, empty heap, in which all the tiles are copied sequentally. As can be supposed, it requires twice as much memory as the original heap it is saving.

```
void Conim::PhysicallySave(void)
{
            …
/* Give each tile a saver_node */
```

```
                int NumTiles = 0;
                SortSegments();
                Assert();
                for (each_tile)
                          NumTiles++;
                size = NumTiles * sizeof(struct saver_node) + 12;
                Savers = (saver_type) ::malloc(anon_heap, size);
                S_idx = 0;
                for (each_tile) {
                    Savers[S_idx].offset = (uint)-1;
                    Savers[S_idx++].tile = tile;
                }

                /* Recurse on the root tile. Garbage collection */
                HeaderSize = Heap::RoundUp(sizeof(struct ContainerHeader)) + 4;
                offset = 0;
                Recurse((void**)&header); // FindPtrs-only phase
                …
}
```

2.4.5.6 Summary

The main part of a container in Barbados is its heap. The heap is used in order to store there all container's data: globals, constants, functions, classes/structures/unions ... etc, so it does not correspond to the usual meaning of a "classic" heap, in which only objects for which the user requested dynamic memory are stored. The benefit here is that we have all data needed for an individual container into the same location.

A heap must be managed with the classic heap functions: `malloc`(), `free`()... and so on. We use a heap architecture which is composed by pieces called tiles. Tiles have a way to express if they are free or allocated (and in this later case, with what type). At its beginning, the heap begins with a single huge free tile which holds all the space.

Wrapping this functionality, we have the heap class, which helps to map *Conims* with its heaps.

2.4.6 Container_id_tree module

The purpose of this layer is to assign and disassign *container_id*'s, and also to take care of each container file associated to each container and *container_id*. Also, given containers have parental relations, this module also accommodates the files in the host plataform which will hold each container, mapping them with containers through the container_id. These files, obviously conform the Persistent Store at a physical level, and this module is which presents the abstraction of the Persistent Store to the rest of Barbados.

About the Persistent Store, as has been said, is divided in containers, which is a kind of clustering (Sousa & Alves, 1999), but handled by the user through a directory abstraction.

The main structure at this layer is just a vector[8] of structures containing a pointer and a int value. The container_id is just the index in this structure, and the `int` of the structure contains the container_id (the index) of the parent container, or 0 if this entry is not assigned. The ROOT container has a fixed container_id (index) of 1, so the 0 entry of this table is not used, being the value of a bad/erroneous container_id.

2.4.6.1 The structure of the container_id table

This is a draft of the structure for the *container_id_tree* table.

```
struct container_id_t {
            int       parent;

};

container_id_t containeridtable[];
```

The parent field stores the *container_id*, which is the index of the parent container. This way, by a simple access, the parent of any container can be found, although it is more difficult to find all subcontainers of a given *container_id*. Anyway, the latter case would happen only while recovering the PS in case of a disaster, by specific Barbados tools.

2.4.6.2 Interface of the container_id layer

This layer supports the following operations:

```
container_id  NewContainerId(container_id parent);
bool          DeleteContainerId(container_id pid);
bool          MoveContainerId(container_id pid, container_id newparent);
char*         CidToPath(container_id pid, char *buf = NULL, int sizeofbuf= 0);
container_id* DiskListOfChildren(container_id pid);
container_id  RetParent(container_id pid);
container_id  PathToCid(char *);
bool          Fix(void);
```

[8] Actually, it is a memory-mapped file.

```
bool          DeleteContainerId(container_id pid);
bool          CheckContainer(container_id pid);
void          FreeRes(void);
container_id  MaxId(void);
```

2.4.6.3 The most important container_id_tree interface functions

The most important functions at this layer are `NewContainerId()`, `DeleteContainerId()`, `PidToPath()` and `PathToPid()`.

2.4.6.4 NewContainerId()

The `NewContainerId`() is a very simple function which returns an unused entry in the *container_id* table (it's unused if its contents (its parent) is 0, an invalid value for a `c_id`), and assigns it to a new container, *son* of the one provided as parameter. It has also to create the associated, empty container file. If it is the first subcontainer of its parent, then the associated directory is created (in which all the children containers of a given container, named with the `c_id` code of the container, reside).

2.4.6.5 DeleteContainerId()

These functions put the parent field of the container_id entry given to zero (unused), and deletes the container associated file. If it was the last container of its parent, then the directory associated to the is removed.

2.4.6.6 CidToPath() and PathToCid()

These both functions do the conversion from/to a `container_id` to/from a given path. For example, a path which is "/" has a corresponding *container_id* code of 1.

The path from a pid must be built by traversing the structure through the parent fields, until ROOT is reached.

Please note that these paths are real paths in the underlying OS (Windows), while `/STD/cstring` is a path created with *namedobj's*, valid in Barbados. The physical associated file names are exactly their container_id and the '`.ctr`' extension. When a container is the parent of other containers (i.e., those containers are its children), then a directory is created (again using its *container_id*), and its children is put there. This can be thought recursively for any possible container.

2.4.6.7 Summary

The table 5 summarises all explained concepts. It shows the ROOT container (*container_id* 1) plus two subcontainers of it, the *container_id* 753 and the *container_id* 68. The *container_id* 0 is always used in order to report error conditions. Note that the container 753 (associated file "753.ctr") has a child container (#68), and therefore, container 68 is under a Windows directory with the same name of the parent container (753).

OS Path	***Barbados Path***	***Container_id***	***Value in table***
C:\ps\1.ctr	/	1	0
C:\ps\753.ctr	/example1	753	1
C:\ps\753\68.ctr	/example1/SRC	68	753

Table 5. *Correspondences among different codifications (Barbados id's, OS files ...)*

Chapter 3: State of the Art in Persistence

3 State of the Art in Persistence

3.1 Introduction

There have been many research prototypes of persistent systems, of which none have found their way into widespread use. For example, the E++ system, (Vemulapati e*t al.*, 1995), PJama, (Atkinson & Jordan, 2000), and so on ... However, there are different models of persistence, and even different ways to understand persistence. The basic idea of persistence is to remove the barrier -sometimes called the impedance mismatch- between memory (RAM) and the secondary storage (disk) -sometimes organised in a database-.

All programs must store their settings and data in a suitable format on disk. But this implies most of the times the flattening of data structures in memory, in order to make them fit in a byte stream on disk, when saving; and unflattening them from disk again, when loading -probably every time the program is executed-. These tasks -shown in figure 15- must be performed by the programmer, and they are non-trivial, error-prone tasks. Sometimes, these tasks are also a big part of programs, too. They are specifically the barrier which is expected to be removed by persistent systems, in they variety of approaches.

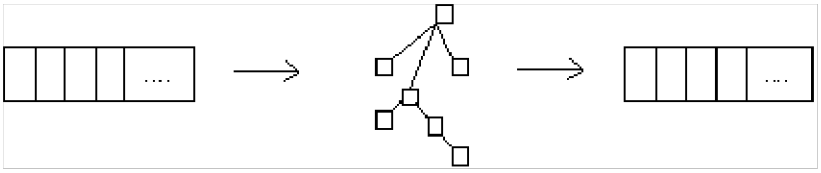

Figure 15. *The data of the application is stored as a stream of bytes, and the first task of the application is to convert it (unflatten it) to a data structure in memory. Once the execution is finished, the last task of the application is to convert it (flatten it) to a stream of bytes.*

There is a wide range of systems providing some kind of persistence. A relevant number of them will be described here, organised in: Persistent Programming Systems (PPS), Object Oriented Operating Systems (OOOS), and Object Oriented Database Management Systems (OODBMS) -in contrast to RDBMS (Relational Database Management Systems)-. For a brief introduction to persistence and some systems offering persistence, the reader can found interesting other sources elsewhere (Loomis, 1993; Srinavasan & Chang, 1997).

Relevant persistent systems in the area of Object Oriented OS (Operating Systems) are Grasshopper (Dearle, *et al.*, 1993), and EROS (Shapiro *et al.*, 1996). OOOS are important because, as they use objects –instead of files in traditional systems- as their basis, they must use some kind of persistence in order to store that objects in the secondary storage.

In the area of Persistent Programming Systems, we can find Integrated Persistent Programming Systems (IPPE) (Atkinson & Morrison, 1986), which provide programmers with an editor, and execution environment and sometimes a debugger. Barbados (described initially in Cooper, 1997), based in C++ (Stroustrup, 1991) is one example. Also, the IPPE Napier (Atkinson & Morrison, 1989), based in the Algol language, is a good example (although it is a quite old system). Other PPS without an integrated environment are, for example, PJama (Atkinson & Jordan, 2000), Orthogonal Persistent Java (Marquez *et al.*, 2000), Oberon-D (Knasmüller, 1996), and so on. These persistent programming systems provide the programmer with a way to store the objects the application uses.

Relevant Object Oriented Database Management Systems are, for example, Gemstone/J (based in Java), (described in Bretl, *et al.*, 1991), ORION, described in (Kim *et al.*, 1991), or O_2 (Banchillon *et al.*, 1992). Object-oriented database management systems use objects and classes in order to represent information instead of records and tables. They implicitly use, therefore, a kind of persistence in order to store data. An interesting system, although outside the scope of this thesis, is Oviedo3, (Álvarez Gutiérrez *et al.*, 1997; Ortín, *et al.*, 1997), a system offering persistence through the intensive use of reflection.

Along this chapter, as more than one system, and therefore programming language, is going to be reviewed, general terms such as '*attribute*' (for member data in C++) and '*method*' (for member functions in C++) will be used, instead of specific ones relative to any language.

This chapter will be organised as follows. A description of the more commonly used model of persistence, the orthogonal one, will be presented. Then, a revision of techniques of swizzling, clustering and schema evolution will be discussed. A complete revision of existing persistent systems (the ones enumerated above and some others) will also be studied. Finally, the reader will find a classification of all these systems, and the conclusions, at the end.

3.2 Orthogonal persistence

Orthogonal persistence was a concept introduced by M.P. Atkinson and R. Morrison

(Atkinson & Morrison, 1995), and it consists of three principles:

- *Data Type Independence*: The persistence mechanism is orthogonal to the type system of the language. Any object in the language is allowed to be persistent, regardless of its type. (cf systems where persistent objects must inherit from a special 'persistent' class).
- *Persistence Independence*: all code should have the same form irrespective of the longevity of the data on which it acts. Therefore, persistent objects are treated exactly the same as transient objects. This prohibits providing special persistent pointers or functions to convert objects to persistent data.
- *Persistence Identifi*cation: There should exist a consistent mechanism for determining the longevity of objects, and should not be related to the type system. This is normally achieved applying an appropriate algorithm, which declares as persistent any object which is reachable from a single 'root of persistence' by following pointers (this is called persistence by reachability).

However, the provision of orthogonal persistence comes with various problems, and some persistence researchers including us do not share the goal of providing orthogonal persistence but rather modified forms of it (Roselló *et al.*, 2001). We accept these three principles as a useful basis framework for persistence, but we think it is too restrictive, since for example the container-based model explained here, which has valuable benefits, could only be classified as *type*-orthogonal persistence (Cooper & Wise 1996). This is because in the container-based model the user must specify at some point in which container he or she wants to store or retrieve his or her data, breaking the *persistence independence principle*. Also, the *persistence identification principle* is broken, because a) we support C++, which forces the use of the **delete** operator in order to specify when the object is not needed anymore; and b) although each time a container is saved, a "*persistence by reachability*" algorithm is run to eliminate all possible garbage, this happens individually for each container, and not globally for all objects in the system.

3.3 Adding persistence to existing systems

Many attempts have been carried out in order to add persistence to an existing programming system. The main ones have chosen as target C++ (although some of them chose Pascal (Cockshott, 1993)).

Normally, the problem consists of designing an object store (a PS (Persistent Store, as defined in Atkinson *et al.*, 1986)), and then make possible the access from the programming language to the store, extending the language in some way.

The level of sophistication varies in a quite large scale, from very primitive persistence support, to complete systems which try to integrate their services through libraries and clever solutions using object-oriented capabilities such as inheritance and constructors/destructors (such as for example, SOS (Shapiro, 1989), discussed below).

Even the creator of C++, Bjarne Stroustrup, suggests the way to achieve some degree of persistence, although he warns that complete persistence in standard C++ is not possible, due to the impossibility of eliminating totally the impedance mismatch (barrier) between first (memory) and secondary (disk) storage (Stroustrup, 1991; Ellis & Stroustrup, 1990).

In general, supporting persistence is complex due to the use of conventional operating systems. They present a set of penalties and lacks which make difficult the build of a persistent

system. Basically, they don't present abstractions suitable for persistence implementation. This is studied in (Dearle *et al.*, 1992). Moreover, current Operating Systems are based on technology designed between 1960 and 1970, i.e., following the UNIX metaphor of files (Balch, *et al.*, 1989). The metaphor of files consists of representing all resources in the system (processes, devices ...) as files, with common operations over them (*open*, *close*, *read* and *write*). Current operating systems probably should be based on objects, such as Grasshopper (Dearle *et al.*, 1992) or EROS (Shapiro *et al.*, 1999), for example. But these operating system haven't achieved enough popularity.

The main drawbacks of systems based on traditional OS's are two: the commented problems derived from the use of a traditional operating systems, which make all developers to create an abstract machine over them; and secondly, that each developer creates a different, incompatible abstract machine, which avoids interoperability among them (Rosenberg *et al.*, 1996). Interoperability is an open research path, explored currently in JSpin (Kaplan *et al.*, 2000). In Barbados, interoperability is not supported, although we plan as future work, to prepare Barbados to be able to export *containers* to Windows[9] DLL's (Pettrieck, 1994).

Other relevant attempts in the past consisted of studies about the use of RDBMS (Chen *et al.*, 1995), and an attempt of a design of an standard layered architecture for persistent object stores (Balch *et al.*, 1989). The latter one is particularly interesting. The definition of a, somehow, universal virtual machine would be very useful as it could provide all users with persistence. This is related to interoperability, but it would mean a significant step beyond, being any system, based on a file system or any other architecture, to recognise an object (in its general meaning) and be able to deal with it.

However, despite of all these problems, a wide variety of approaches for persistence apply. A very primitive way to support persistence is to build some mechanism over the OS resource of "*memory-mapped files*" (for example, Microsoft 1998a). Mapped files consist of files that are actually completely loaded in memory, in read-only mode or in write mode. This section of memory can be modified (if the file is mapped in *write* mode), and the changes are reflected in the file on disk. This can be the basis of a persistence mechanism (Barbados uses memory-mapped files intensively), but not a complete persistence mechanism.

A not so primitive way to support persistence in C++ can be the one defined in (Florentín, 1998). The method described here solely provides the programmer with a semi-automatic serialisation. As the author says, these objects must be pure stand-alone ones: they shouldn't define any relation to any other object. Of course, these objects must be recovered by the same process that created them – this is a big difference in contrast to persistent systems, as in persistent systems the PS resembles somehow a big object-oriented database which can be inspected and used by any process.

A complete description of the requirements of a language such as C++ to support persistence can be found elsewhere (in Shilling, 1994, and in Biliris *et al.*, 1993, as a description of O++). One big problem with C++ is that, as it maintains compatibility with C, classes and objects don't exist as entities at run-time: they are translated to structures (`structs`) and their related functions (with an extra `this` formal parameter which identifies the object/struct). Extra data members are added to these `structs`, in order to support object orientation, such as the `vptr` pointer. The **vptr** pointer points to the **vtable**, a table needed in order to implement polymorphism (i.e., through *virtual* functions in C++). This is deeply discussed in widely known and available bibliography (Stroustrup,

[9] Windows is a registered trade mark of Microsoft, Intl.

1991; Ellis & Stroustrup, 1990). Moreover, C++ didn't have a Run-Time Type Identification service (RTTI) until the ISO/ANSI standard, approved in 1998 (Allison, 1998).

Other attempts have to do with persistence support through relational databases. Objects and classes are mapped to tables in the database (Vadaparty, 1999a; Vadaparty 1999b). Although this can appear to be more sophisticated, the basis for persistence support is the same one: *serialisation.* However, in these cases, objects are stored not in plain files of the operating system, but in more appropriate *blob* (*Binary Large OBject*) fields. Tables can be used in order to build more or less complex "*by class"* classifications of objects.

The problem with all these approaches is that some of them are so primitive that are useful only in very specific situations. Other complex approaches are more widely useful, but never achieve the high degree of integration with programming languages that a pure persistent system would achieve. This means, in summary, that they break in many points the principles of orthogonal persistence discussed above.

3.4 Swizzling techniques

In this section, an overview of relevant swizzling techniques in persistent systems will be discussed. Swizzling (Moss, 1992) basically consists of converting object references from pointers in memory to pointers in disk –when the objects are stored in the PS- and vice versa, when the objects are recovered from the PS. Objects are recovered from disk normally when an object which doesn't exist in memory is referenced. This is known as an '*object fault*'.

This conversion (swizzling) is needed as soon as a general persistence mechanism, embracing saving and restoring simple and complex objects (objects composed by -or related to- other objects) is built. The main need when an object is going to be stored in the PS is to locate all pointers inside it, referencing other objects. If we want the object to have sense after a later recovering, we will have to save also its persistent closure, too (all objects related to the first object, and recursively all objects related (by references or other relations) to the second-level objects and so on ...). These object references can of course be true pointers as in C++ or, for example, references in Java.

In order to be able to detect pointers in objects, it is needed to work with a language providing structural reflection (Kirby, 1992; Cooper & Kirby, 1994). Structural reflection consists of the ability of a language to report about the components of any object built with that language. For example, Java provides programmers with that functionality (Kirby & Morrison, 1998b).

If a language doesn't provide reflection, then a runtime providing that functionality and a compiler generating that information must be designed. This happens for example in Barbados (Cooper, 1997), which supports C++ as its programming language. C++ doesn't provide structural reflection -despite of some information about the class that C++ objects pertain to: this is called RTTI (Runtime Type Identification, introduced in (Allison, 1998))-, so it was needed to create a compiler generating this information for each entity compilation (This is deeply discussed in the chapter "*the container-based persistence model*" of this PhD).

Another kind of reflection is "*behavioural reflection*", which consists of dynamically change the meaning of operators and other constructs of the language (Malenfant *et al.*, 1996; Ortín, 2001). Behavioural reflection is not specifically needed for persistence and it is not studied here.

Once the swizzling mechanism is available, the developer of a persistent system must choose when to apply it. Depending on the method selected, swizzling in the save phase can be necessary or

not, while the load phase is the most critical one.

In the loading stage, there are two main approaches: *eager* swizzling and *lazy* swizzling. When using eager swizzling, all references in objects loaded in the last disk access (due to *clustering* – please refer to next section) are fixed to memory pointers. If the approach chosen is the lazy one, then pointers are swizzled if and only if the pointer is dereferenced.

Note that the process of converting pointers means to transform an on-disk reference to an on-memory reference (a memory address). There are also more than one approaches to on-disk identifiers. Normally, the on-disk reference consists of the OID (Object Identifiers) of the referenced object (for example, Oberon-D (Knasmüller, 1996) or Pjama (Brahnmath *et al*., 1998)). In some systems, such as in Barbados (Cooper, 1997), the on-disk identifiers are just memory pointers. In fact, when the container is loaded in Barbados, and the container is not possible to be placed in the same location it was the last time it was in memory, all pointers are searched and fixed with a *delta* based on the new location of the container and the last one.

Another classification can be done attending to whether object(s) are copied before doing swizzling or not (Moss, 1992). If the object is copied and then swizzling is done, then we have *copy-write* swizzling. The advantage in this case is that objects remain unchanged in the persistent store. The other possibility is *in-place* swizzling. The object in which swizzling is done is the same one in the persistent store. This probably means that the object will have to be unswizzled before storing int again in the PS. The swizzling strategy that better fits with the Barbados approach is *in-place* swizzling, i.e., the container is loaded, swizzled, used, and then unswizzled and saved. The only difference is that we only have to "unswizzle" it if the location in memory of the container has changed.

Therefore, Barbados is an example of *eager* swizzling, as the container is swizzled as soon as it is loaded in memory, and it is then totally prepared for use. Oberon-D is, in turn, an example of *lazy* swizzling. The *nil* exception is trapped in Oberon, and the persistence system determines if the reference that caused the exception was actually a *nil* pointer or an OID. If it was an OID, then the object is retrieved from the PS, and prepared to be used in memory. Some cache techniques are used in order to improve the performance of this system.

Despite of software techniques, hardware ones can be used. For example, in Grashopper, hardware techniques are intensively employed (Vaughan & Dearle, 1992). Swizzling by hardware consists of relying in the virtual memory mechanisms in order to detect when swizzling is necessary. If the *unswizzled* references are set conveniently, such as an invalid memory address, then a fault page error or a memory protection error will be generated. This can be trapped and used as the point in time to load requested objects and convert the *unswizzled* references in references to them. Using these hardware techniques comes with some drawbacks: for example, in (Narasayva & Ng, 1996) it is described how memory pages in which reside unmodified objects become dirty (i.e., they must be written back to disk) anyway just because of doing swizzling in them.

In (Moss, 1992), the performance of *eager* and *lazy* swizzling mechanisms is discussed, concluding that the *lazy* approach is always more expensive in computational terms than the *eager* one. They assume create/load-work-save sessions in the persistent system, which is very suitable for Barbados (Cooper, 1997), as containers are expected and designed to be used that way.

The point in which an OID is referenced and must be converted to a memory address of an object which must be loaded, is known as *swizzle barrier*. Swizzled barriers have been widely explored in literature. For example, in PJama this point was thoroughly studied (Brahnmah, *et al.*,

1998), in order to offer better performance, while in (Korsholm, 1999), a design of an evolutive cache (roughly a cache in which objects pertain to different levels depending of the time they've passed inside it) is presented as an element of performance improvement.

3.5 Clustering techniques

Clustering (Tsangaris & Naughton, 1992) is basically the technique used in order to group objects together, trying to keep disk accesses to the minimum. Normally, a given cluster is loaded due to a reference to one of the objects within it. The success of the clustering strategy employed would be to have loaded within that cluster all objects needed in the near future (i.e., all objects that are used more or less immediately by the object that caused the object fault).

In this section, a brief overview of literature about techniques available for clustering –from manual to automatic ones- will be presented.

As explained, the ideal clustering technique would be the one guaranteeing that, once an object is referenced, and its cluster is loaded, that cluster contains all objects needed by the first object. This is difficult to achieve, as only studying the behaviour of the programmer it would be possible to succeed. Other times, objects are part of an application and their relations are fixed, so it would be very inefficient to try to study which objects are related to which ones.

There is a wide variety of methods for determining which objects should be grouped together. They can be divided in *static* and *dynamic* strategies (Darmon *et al.*, 2000). While static methods always behave in the same way, following the same patterns in order to find which objects should be grouped together, dynamic methods are adaptable algorithms which, by using, for example, statistics of use of a group of objects, change the clustering in the PS, re-grouping most accessed objects together.

For example, IK implements an adaptable clustering algorithm (Sousa & Alves, 1994). Clusters have a *head* object, which is the one from which the rest can be accessed. The system periodically verifies whether other objects could be classified as *head* objects, and creates clusters for them and their related objects.

Orion (Kim, *et al.*, 1991), uses segments in order to group objects in the persistent store. Segments are automatically managed by Orion: they are just composed by a class and all its objects. Sometimes, it would be useful to group together, in the same segment, objects of two or more classes, due to rules of association among them (i.e., they are always used together). For that cases, Orion provides a mechanism for the programmer, allowing him to indicate that special cases, manually. The clustering strategy of Orion constitutes a mixed approach, based on a fixed grouping strategy and a manual mechanism for more complex cases.

In O_2, (Benzaken, *et al.*, 1995), a dynamic technique called Placement Tree is used. A placement tree is a representation of relations among objects. These placement trees are used in order to define clusters. Placement trees can vary in time, and the system changes assigned clusters along with placement trees change.

In (Tsangaris & Naughton, 1992), as well as in (Darmon, *et al.*, 2000), different dynamic clustering strategies in OODBMS are studied, including the commented ones above. In (Darmon *et al.*, 2000), the discussion is centered over statistic methods, which take note of the frequency some objects are used in respect to some others.

Conclusions of these studies are that automatic methods are expensive. They are always

more expensive than any other, fixed, method.

In this sense, some systems, such as Barbados or PerDis (Shapiro *et al.*, 2000), doesn't offer any clustering strategy: the user simply selects the cluster in which objects are going to be stored. The user is obviously expected to store in the same cluster all objects logically related. In the concrete case of Barbados, directories are used as an abstraction over clusters: the user chooses a directory to work in, unconsciously indicating to the system the cluster that must be used for next objects. Also, the user has low-level managing possibilities over containers, being able to create them programmatically, through the Barbados API, if needed.

3.6 Schema Evolution

In traditional systems, programmers have to write software in order to read data of secondary storage and into memory, and back again. This software can often be written to serve an additional purpose: to make existing data compatible with successive versions of the application as it evolves (as an application evolves, record structures change, fields in records are added etc.). But inherent to the concept of persistence is the idea that all data is accessed directly, without intervening flattening/unflattening or conversion software: this signals the need of schema evolution.

A persistent system is a system in which data structures (and perhaps classes and software too), persist through executions of any process. In this scenario, it is possible –although improbable- to write a class and then leave it untouched to the end of the life cycle of the program. It is much more probable to expect this class to be modified a few times during its life (Atkinson *et al.*, 1993), so therefore, the system must provide mechanisms in order to basically a) allow users to modify classes and b) convert instances of that classes to the new definition. All kind of problems arise if an object is out-of-sync with its type definition (class). To avoid these problems, we need a system able to cope with existing data objects when a class is changed, either by converting them *eagerly*, or *lazily*, (or in some cases deleting them or applying old versions of the software to them).

In a non-persistent program, the class would be changed and then the programmer would have to write procedures in order to adapt all existing data –in files- to the new class.

There are two main different types of schema evolution: (a) conversion of "old" objects to new class definitions, and (b) defining multiple versions of an object, allowing other objects to link, a concrete version of a given object or any possible up-to-date version. Versioning can be found interesting for some domains, such as some systems related to CAD applications; however, versioning is not going to be discussed in this document, as although it is a more flexible mechanism, it is also more complex, and it is not widely supported in persistent systems.

When a class changes, then we generally need to convert instances of this class before they can be used. This change can be done in an *eager* or in a *lazy* way: '*Eagerly'* means changing all instances of the given class as soon as the class has been changed. This normally involves putting the object store (or object-oriented database) to an off-line state and then perform the change, which can take a long time. '*Lazily'* means to convert objects only when the objects are going to be used.

The *lazy* approach has the additional advantage that it can be performed without putting the database offline. As a disadvantage, instances of the modified and of the old class will coexist together in time while all instances are not changed (although they are going to be changed as soon as they are loaded in memory).

Moreover, about kinds of change, as a simple, introductory classification, we can find changes affecting: a) the class (adding, deleting, modifying a class member), and b) the class hierarchy (adding or deleting a class). These two categories can be found in all systems, however, many of them define a complete set of subcategories.

Although a variety of efforts have been done in the way to schema evolution, it is the OODBMS field that has mainly paved the way (for example, Tang & Katayma, 1989). One good example is O_2, because it is one of the most recent OODBMS developed, but also GemStone, and Orion, which are older. A review of all these OODBMS's can be found in (Brown, 1991).

Persistent Programming Systems (PPS) are also affected by this problem, so those ones supporting schema evolution, such as for example, PJama (Atkinson & Jordan, 2000), Oberon-D (Knasmüller, 1997), and JSpin (Ridgeway & Wileden, 1998), will be also studied.

3.7 Brief Comparison between the Container-based Persistent Model and the Orthogonal Persistent Model

The more commonly accepted model in the persistence community is the orthogonal persistence model. In this section, this model is compared with model presented in this work, the container-based persistence model.

Concretely, we like to analyse persistent systems within the following framework: the three main issues to be addressed in the implementation of a persistent system. These main issues are: clustering, memory protection and schema evolution.

These three problems are discussed through next sections, along with a comparison of orthogonal persistence in contrast to the alternative approach of containers.

3.7.1 Clustering

If the fine-grained objects which compose data-structures are scattered randomly over the secondary storage, then the system will be unusable because each fine-grained object access will require disk access and will therefore be unfeasibly slow.

Instead, clustering is used to group objects which are likely to be accessed together. There are many clustering techniques, each of them using a different way to define what objects are related to others (Sousa & Marqués, 1994). For example, the inheritance and composition relations may be useful for this purpose.

3.7.1.1 Clustering in Orthogonal Persistent Systems

Orthogonal Persistent Systems normally perform object cluster as a by-product of the garbage-collection algorithm. Objects which are closely related via references are often located physically together. This works fairly well, although clustering will always be better in systems where the programmer defines the clusters.

3.7.1.2 Clustering in Container-Based Systems

In Barbados, containers consist of a root object from which all other objects in the container must be reachable. Organising the PS using clusters of objects in a system (it doesn't matter whether it is an orthogonal persistent system or not), is known to be useful (Sousa & Marques, 1994).

The application programmer indicates which container each object is to be placed in, instead of using automatic clustering techniques, which can be expensive in some situations (Tsangaris & and Naughton, 1992). Normally, this would mean we would have to specify a container in any object creation. However, we do not (normally) have to do that, since the user normally specifies the container in which objects are to be created simply by choosing which directory he wants to work in: the root object of each container is in fact a directory, (and creating a new directory will create a new container).

So we only had to add a simple abstraction layer to containers, in order to *convert* them into directories.

3.7.2 Memory protection

Users of traditional non-type-safe programming languages are used to the concept that if their program has bugs in it, then it can corrupt the entire address space of that process and cause all data in that process to be lost. However, no programmer would be willing to use a persistent system

if a program can corrupt the entire system including all data saved on disk. Yet the very goal of persistence is to make the entire system resemble (in some way) a single giant address space. Therefore memory protection is an issue persistence researchers must specifically deal with, as it can be seen for example in Grasshopper (Dearle, *et al.*, 1993) and PJama (Jordan & Atkinson, 1998).

3.7.2.1 Memory protection in Container-Based Systems

Containers have the property that they must be explicitly opened and closed. Any container which is not open cannot be corrupted. Therefore, if there is a bugged program or stray pointer, it can corrupt the entire container, but it will not corrupt other containers any more than a bugged program can destroy arbitrary files in a traditional system. Furthermore, if a container is opened in read-only mode, then any attempt to write into it will cause a segmentation fault.

In Grasshopper, a process executes in only one container (although other entire containers can be mapped into it), which is expected to provide protection and decrease overheads (as we have seen and we intend to happen in Barbados, too) in such operations as garbage collection or checkpointing (for example, Shapiro, *et al.*, 1999). Barbados differs from Grasshopper in that Barbados addresses issues of fine-grained object management, whereas Grasshopper leaves that up to application developers.

Of particular relevance is the fact that to allow communication and cross-correlation of data between different containers, we have developed a system whereby references from one container to the objects inside the directory of another container, are supported. All other objects are 'hidden' from view from other containers. Moreover, we have found that the garbage collection and checkpointing algorithms are very efficient, because, again, we are referring only to the contents of only one container (as defended in Cooper & Wise, 1996).

3.7.2.2 Memory protection in Orthogonal Persistent Systems

Orthogonal persistent systems normally rely on type-safe languages to eliminate the possibility of a large class of memory corruption errors, (for example, Jordan & Atkinson, 1998).

However, there are still other classes of memory corruption errors which containers can largely prevent against which orthogonal persistent systems can't: for example dropping/overwriting a reference to a data-structure with the unintended consequence of deleting that data-structure, or having stray references into a data-structure leftover in unusual places with the unintended consequence of those pointers being used to modify data in the data-structure. These problems are dealt with by providing checkpointing and resilience mechanisms (for example, Shapiro, *et al.*, 1999).

3.7.3 Schema Evolution

As explained, inherent to the concept of persistence is the idea that all data is accessed directly, without intervening flattening/unflattening or conversion software. Therefore, schema evolution becomes a problem, which can potentially plague persistent systems more so than traditional systems.

Schema Evolution is usually divided in two tasks (Ferrandina & Ferrán, 1995): a) detecting when a class is modified, and b) converting from old types to new types.

3.7.3.1 Schema Evolution in Orthogonal Persistent Systems

Schema Evolution affects all the PS, and all the involved objects must be found and

converted in some way. This means that in many systems (like for example Pjama, Dimitriev & Atkinson, 1999), this conversion must be made in offline mode. In other systems, like O2 (Ferrandina & Ferrán, 1995), the system is able to do a *lazy* conversion, so the first time an object involved in Schema Evolution is used, it is converted, without trying to convert all instances in the PS (*eager* conversion) before allowing the next user's action to be executed.

Schema Evolution is a very expensive process in orthogonal systems employing eager modes, since the whole PS must be inspected in order to detect where the evolved class has been used.

3.7.3.2 Schema Evolution in Container-Based systems

Schema evolution in the broadest sense concerns itself with making sure legacy data can survive any kind of change to the software. Let us consider a slightly more restricted and well defined version of 'schema evolution': let us set as our goal "always knowing the type of any piece of data we have", in other words being able to break an object down to its constituent fields (field-names and field types) no matter how old it is and how much the software which created it has evolved since it created the object, and then converting it as automatically as possible.

The concept of a container naturally suggests a certain solution to this problem: have a copy of every type present in the container, in the container. By the 'type', we mean the type-definition: a sequence of field names, types and offsets but not including member functions. In the prototype the C++ language is supported, and only an 'abbreviated' copy of the full class definition is copied into each container, except the one container that contains the full class definition. So if a container contains 1000 objects of class C in one container, there will only be one copy of the abbreviated class-definition for 'C'. Containers provide a way to 'factor out' type information, while at the same time ensuring that the container is self-contained in the sense that it can migrate to another PS without losing information on how to interpret all the objects inside.

Knowing the type of any piece of data is one thing; but at some point we also need to compare the type with the type the application is expecting, and do something other than report a type-error. The expected action here is to convert affected instances. The key concept is that, in the container-based model, it is possible to implement a convenient mixed approach of *lazy* and *eager* conversions. This will be studied thoroughly in the chapter dedicated to schema evolution.

3.8 Revision of relevant persistent systems

3.8.1 Introduction

In this section, several relevant persistent systems are discussed. Additions of persistence capabilities to existing languages will be presented first. Then, persistent programming systems are exposed, followed by the integrated persistent programming systems. Later, OOOS's and OODBMS's are discussed, without any special order.

3.8.2 Persistent programming systems

Persistent programming systems consist of a programming language and a persistent store. The language must include an API suitable for store and retrieving objects in and from the persistent store. The integration between the PS and the programming language varies in all systems. Normally, those systems which propose persistence as an extension to an existing language, provide a loose integration with the PS, through a very rigid interface –sometimes, persistence is provided, for example, through a library-. Other systems propose a completely new environment and implement a compiler or an interpreter of a given language in that environment, modifying appropriately the language when needed. In these latter systems, integration with the persistent store is very good. In a medium point, we could find those systems which provide high integration with the persistent store through flexible mechanisms –reflection, metaprogramming- found in the programming language, such as PJama (Atkinson & Jordan, 2000).

Some systems go even beyond that point of high integration between the programming language and the persistent store, and, emulating integrated environments widely known nowadays, such as C++ Builder (Borland, 1997), they present an editor, a debugger and a compiler or an interpreter with a very high integration among them: for example hyperprogramming environments (Zirintsis, *et al.*, 2000). The reason this is radically different to other existing systems (such as Code Warrior (Metrowerks, 1999), or Visual C++ (Microsoft, 1998b), among many others and the already commented one from Borland), is because they claim that already existing environments are too biased in the *edit-compile-execute* schema, while they try to make no distinctions among the mentioned stages.

Barbados presents an integrated environment composed by a compiler, an editor and a debugger. All the work is done inside the editor, so, accidentally, the line dividing the compiling and executing stages is very thin -actually, the *commands* (such as `cd()`) available in Barbados are just functions which are executed immediately-.

3.8.2.1 The Napier integrated persistent programming system

The Napier (Atkinson & Morrison, 1989; Morrison *et al.*, 1999) persistent system is a programming environment which supports a variant of Algol, formerly called PS-Algol (Atkinson *et al.*, 1982; Atkinson *et al.*, 1983), in order to provide programmers with persistence.

```
let ps = PS();

project ps as X onto
env: use X with User: env in
            begin
                        type complex is structure {rpart, ipart: real)
                        in User let add = proc(a,b: complex → complex)
                                    complex (a(rpart) + b(rpart),
                                                a(ipart) + b(ipart))
            end
default: {} ! This is executed if the projection fails
```

Figure 16. *Storing a procedure add in the PS, User environment.*

Napier is not an object oriented programming, as it is based in the Algol language: instead, it offers the *type* construction, which can have pointers to functions as members. This allows the programmer to create an object of a given type and then initialise all the pointers to the actual functions.

In the figures 16 and 17, the saving and retrieval processes of a procedure in the PS are shown. Specifically, in the former figures the procedure 'add' (within the type 'complex') is defined and made persistent. In the latter one, the procedure 'add' is retrieved and used. Objects are stored in environments in the PS, which are a kind of clusters, with the extra property that the relations among environments don't follow any scheme (such as the hierarchical one), and therefore an environment can reference any possible number of other environments. Environments are roughly collections of tuples *(name, type, value)*.

As can be seen in the figures referenced above, any manipulation of persistent objects must be preceded by a call to the PS() procedure and a projection of the PS. If this is not done, then variables, procedures ... etc are not made persistent. For example, in figure 17, the variables *one* and *two* are not going to persist (specifically, these variables have not been made reachable from the persistent root).

The Napier language is a type-safe language (Connor, 1988): this means that strict type-based rules are used in order to decide if an operation can be applied to a given object, avoiding memory errors. This also means that, for example, the kind of type casts or pointer arithmetic which are possible in C and C++ (Stroustrup, 1991) are not allowed in a language such as this one.

Also, data protection in the Napier Language is provided (Morrison *et al.*, 1990) through mechanisms widely used and/or known nowadays such as subtyping, abstractions and views.

Finally, schema evolution is not specifically addressed in this system.

The authors call their approach an IPPE, which stands for Integrated Persistent Programming Environment; the concept of a self-contained persistent programming environment was introduced by them, in an article previous to Napier, but after PS-Algol (Atkinson & Morrison, 1986). The main idea supporting the basis of their environment is the concept of hyperprogramming. This means that hyperprogramming is an abstraction which unifies source code, executable code and data. Persistent languages are specially suitable, as they unify the long-term and short-term data, and executable code is normally (as in Napier or PJama) a first-class entity in them, too.

```
type complex is structure (rpart, ipart : real)
let ps = PS();

project ps as X onto
env: use X with User: env in
     use user with add: proc (a, b: complex → complex) in
            begin
                        let one   = complex(1.0, 1.0);
                        let two   = complex(2.0, 2.0);
                        let three = add(one, two);
            end
default: {} ! This is executed if the projection fails
```

Figure 17. *Storing a procedure add in the PS, User environment.*

The main advantage is that only one environment is needed in order to manage source code, data and debugging. Hyper-links are created as embedded links and managed in order for the programmer to be able to follow all relations among different parts of source code or even data. This way, a Napier project is a network of objects linked by hyper-links. Processes such as compilation and linking are considered as “accidental”, and are hidden to the user.

An special need for hyper-programming is structural reflection being provided by Napier. This is discussed in (Kirby, 1993).

Other authors have built and hyperprogramming persistent environment for PJama, which is discussed in its main differences with the one in Napier, and about its main architecture in (Zirintsis *et al.*, 2000).

3.8.2.2 The IK persistent programming system

The IK system (Sousa & Alves, 1994; Chennupati & Saiedian, 1997) is an object oriented platform, designed with the aim of simplifying the development of persistent and distributed applications. Objects are assigned to clusters by the system in such a way that there is a single root for each cluster, which is the only known object of that cluster in the rest of the system. The rest of objects in the cluster don't need a global name as they are referenced only inside the cluster.

The PS in IK is a set of storage servers, and a GRL (generic runtime library) is linked to the application, so that this library is in charge of supporting the distributed PS at run time. Its main purpose is to locate objects in foreign stores. This library holds a table called KOT (Known Object Table). At the beginning, each object in the table is supposed to be the root of a cluster, but objects are traversed in order to know if they really pertain to a given cluster. This is done by running over all objects reachable from the KOT, and marking them with a sign different for each root. If an object is marked by more than one root in the KOT, then the object is promoted to be a root object of a new cluster. Objects with only one mark are enclosed inside a cluster with its root being the KOT object of that mark.

The IK platform also includes a Name Server which is in charge of creating unique names for persistent objects. The names are stored in the form, `<user-name, global-name>`, as links which can be readable by the user as if they were files. In order to distinguish among nodes (i.e. computers in an IK network, using IK terminology) in the IK net, each node has an assigned, unique tag, which can be combined with other tags in order to make unique identifiers.

To the knowledge of the author of this thesis, schema evolution is not specifically addressed in IK.

In this system, the subjacent idea of containers with a single public root is clear. Barbados' approach to clustering is similar to the IK's one, although Barbados doesn't support automatic clustering, and it has a more stronger concept of a file system divided in directories, a concept which IK only touches.

3.8.2.3 The Arjuna persistent programming system

Arjuna (Dixon *et al.*, 1989) is a persistent distributed programming system, based in C++. It focuses a lot of attention in hardware/software faults, and one of its main claims is to be fault-tolerant. The system implements a network protocol in order to allow objects to execute methods of other objects. This network protocol has been built over the RPC (Remote Procedure Call) mechanism, wrapped in a C++ class.

The main failures Arjuna must deal with are node failures (system crashes) and communication failures. Computations are considered as atomic actions with certain properties, such as serialisability, failure atomicity, and permanence of effect. The first one ensures correct concurrence among objects which access common resources. The second one assures that a computation succeeds or that it doesn't take effect at all. The latter one ensures that once a computation is finished, its results are not lost due to a later failure.

These properties explain why Arjuna is structured in objects and actions, the latter one operating between objects. The latter is the mechanism which permits the fault-tolerant behaviour of Arjuna.

Arjuna needs certain internal classes to be used inside applications, in order to assure its fault-tolerant behaviour, such as `LockManager`. This way, the code will include references to methods of these system classes in order to assure the principles enumerated above.

The naming scheme in Arjuna consists of a pair, associating names (composed by the tuple: *instance* name, *class* name and *node* name), and the *uid* (Arjuna's unique identifier) of the object. The naming scheme is implemented through the class *ArjunaName*, which can manage partially-qualified names. For example, if only the name of an instance and its class are specified, then the current node is assumed. The naming mechanism consists basically of a table stored in each node, for all objects and classes contained in that node. This implies that the programmer needs to know where the object he wants to reference is.

In order to build new classes, programmers must inherit their persistence classes from the `SateManager` class, which provides the *uid* identifier management, and the virtual member functions (methods) in order to save (`save_state()`) and restore (`restore_state()`). Some function members are provided, in order to ease the task of flattening and unflattening the state of an object; these methods are `pack()` and `unpack()`, for each primitive type. An example can be found in figure 18.

```
class SpreadSheet: public LockManager
{
            int elements[SPRDHT_SIZE][SPRDSHT_SIZE];
    public:
            SpreadSheet(int *, ArjunaName * = NULL);
            ~SpreadSheet();

            Outcome Set(int, int, int);
            Outcome Get(int, int, int *);

            // Arjuna Specific Operations

            virtual void save_state(ObjectState *);
            virtual void restore_state(ObjectState *);
            ...
};
```

Figure 18. *A persistent user class in Arjuna.*

The PS is managed by the `ObjectStore` class. The clustering mechanism consists of grouping all instances of the same class in the same cluster.

Schema evolution is not addressed in Arjuna, to our knowledge.

The Arjuna's approach to persistence has a big lack of orthogonal properties. It's clear that objects can be serialised, but applications must know the class of that objects in order to be able to restore instances of that class. This is justified because of the early date in which Arjuna was developed (before the apparition of the orthogonal principles); and makes clear that Arjuna was one of the valuable precursors in the field of persistence.

3.8.2.4 The SOS persistent programming System

The SOS (Shapiro, 1988; Shapiro *et al.*, 1989) persistent system is based on an extension of the ANSI C++ compiler, and a runtime object management system. SOS supports persistence, migration and therefore distribution. For the SOS developers, migration is just a kind of persistence, in which the destination is the PS of another machine in the SOS net.

Programmers must create persistent objects deriving their classes from the class `sosObject`. The authors claim that persistence management adds an important overhead to persistent objects, so it is important to keep plain, efficient, C++ objects accessible. This base class provides default code for basic actions, such as the default constructor, which makes the main tasks of interfacing with the operating system.

SOS provides the programmer with a new type of pointer, `permPtr`, which must be used in order to assure persistence for the transitive closure of a given persistent object.

Schema evolution is not specifically addressed in SOS. Probably the authors of SOS expect users to create new classes and migrate existing objects manually.

SOS clearly doesn't fulfill the requirements of '*Persistence independence*' and '*Type-Orthogonal persistence*', but this persistent system is quite old and therefore it is one of the more important and useful experiences in persistent programming systems, specifically in how to add persistence to a programming language such as C++.

3.8.2.5 The TEXAS Persistent Store

TEXAS (Shingal *et al.*, 1992) is implemented as a C++ library, and it is expected to be usable from any programming language with run-time type identification.

TEXAS uses the technique, based on hardware, described in the section dedicated to *swizzling* of the present chapter, which mainly consists of using the virtual memory hardware in order to load objects from the persistent store using the so called "*object fault*" trap. The technique basically consists of having pointers to memory-resident pages (*swizzled* ones) and to pages marked as 'not accessible' (*unswizzled* ones). The dereferencing of a pointer such as like this one , the object fault trap is triggered, and the page with objects is read from the PS, swizzling all objects inside it before leaving the page for its use by the system.

As TEXAS is provided in the form of a library, it must support persistence through a strong API, so the barrier between the objects of the programmer in RAM and the objects in the persistent store is still there.

TEXAS doesn't support schema evolution, to our knowledge.

3.8.2.6 The JSpin persistent programming system

JSpin (Kaplan *et al.*, 1996) is a Java-based persistent programming system, which, as far as we know, doesn't provide an integrated programming environment. Its main objective is to support many languages over a persistent platform (Kaplan *et al.*, 2000), of which Java (JSpin is the name of the Java part) is one of them. This persistent platform is the SPIN framework, composed by an OODBMS and an interface, designed in order to be usable by more than one language. The three languages supported by now are C++, Java, and CLOS. Their approach is based on the TI/Darpa Open Object-Oriented Database, and the objective is to be able to manage objects which have been totally or partially created by any of the possible languages. At the present time, it is possible to create a set of related objects in one language and access them in another one, although it is not possible to modify those objects in any other language other than the one used to create them.

The research in schema evolution in JSpin is not as advanced as in the other systems which have been discussed in this document (or which are going to be discussed next), as it is partially implemented. JSpin currently supports the change of any class by another supplied class, provided the new class accomplishes three conditions: byte-code suitability, same name as the original class and a type-compatibility test.

The authors provide a theoretical contribution (Ridgeway & Wileden, 1998) about the impact of class change (related to schema evolution) in their Java-based system.

3.8.2.7 The PJama persistent programming system

PJama (the evolution of this system can be studied in Atkinson, *et al.*, 1996; Jordan & Atkinson, 1998; Atkinson & Jordan, 2000) is a persistent programming system, although a hyper-programming environment has been presented for PJama (Zirintsis, *et al.*, 2000), which would convert Pjama in an IPPE. PJama is based in the Java[10] language, and its development has been directed by Sun Microsystems in the Sun's research project called '*Forest*'.

A special class, `PersistentStore`, is added to the API of Java. Trough this class, it is possible to retrieve objects from the persistent store (as shown in figures 19 and 20). The way to store objects in the PS is to declare an object, which is the root of a data structure, as a persistent root. This way, and through reachability from this persistent root, the transitive closure of this root object is stored in the PS.

[10] PJama and Java are registered trade marks of Sun Microsystems, Ltd.

```
public class SaveSpag {
 public static void main (String[] args){

  Spaghetti sp1 = new Spaghetti(27);
  Spaghetti sp2 = new Spaghetti(5);

  sp1.add("Pesto");
  sp1.add("Pepper");
  sp2.add("Quattro Formaggio");

  try { //catch store exceptions

    //obtain a pers'nt. store
    PJavaStore pjs = PjavaStore.getStore();

    //make a persistent root
    pjs.newPRoot("Spag1", sp1 );

  } catch (PJSException e){ ... } //handle exceptions
 } //end of main
} //end of SaveSpag
```

Figure 19. *Storing objects in the persistent store.*

The use of this mechanism to provide persistence is justified by one of the objectives of the PJama project: to keep back compatibility with the non-persistent version of Java.

This justifies why PJama doesn't fulfill all criteria of orthogonal persistence systems (Atkinson & Morrison, 1995): as transient and persistent objects are not managed the same way, PJama doesn't accomplish the persistence independence principle (as briefly discussed in Jordan, 1996).

```
public class SpagShow {
  public static void main (String[] args){
   try { //catch store exceptions

    //obtain a pers'nt. store
    PJavaStore pjs = PJavaStore.getStore();

    Spaghetti sp = (Spaghetti)pjs.getPRoot("Spag1");
    sp.display();
   } catch (PJSException e){ ... }
  } //end of main
} //end of SpagShow
```

Figure 20. *Retrieving objects from the persistent store.*

PJama also offers schema evolution capabilities, although all that functionality must be accessed through command-line tools. PJama has a mechanism which has been strongly based on the one developed for O_2. The evolution tool supports a special language in order to make the primitives of evolution available to users.

They can only provide an immediate database transformation, as their PS is not able to support the indirection on classes (Dmitriev, 1999) required for a *lazy* transformation in their system. Therefore, as the evolution tool is a command-line program, and as only one process can run over the PS at one given time, its use therefore means to put the PS in off-line mode.

In (Dmitriev & Atkinson, 1999; Atkinson *et al.*, 2000), the authors have implemented the

new persistent store (called Sphere), although they still allow only immediate conversions of all objects, as a) the problem of complex conversion functions is still unsolved (although they plan to use reflection in order to deal with it), and b) they haven't yet decide a method allowing them to detect an old instance being accessed.

- Changes to classes or to the class hierarchy
  - Insert a new class in the middle of the hierarchy.
  - Delete a class from the existing hierarchy
  - Replacing a class in the hierarchy (includes renaming).
- Persistent data conversion
  - Changing objects (immediately) of old class versions.
- Conservative/non-conservative changes. This distinction has to do with the evolution tool. Conservative changes only involve one class, and don't affect the class hierarchy lattice.
  - Conservative: Changes to the class C don't break the relation between C and the rest of the classes in the hierarchy.
  - Non-conservative: some elements of the public interface of the modified class C are now not present.

Figure 21. *Types of available changes in PJama.*

The evolution primitives that PJama supports are the ones listed in the figure 21. The authors of the system distinguish among changes which affect the class hierarchy, changes affecting persistent data and conservative/non-conservative changes. The main reason to add the last type of change is to allow the evolution tool distinguish whether it must deal with the class hierarchy lattice. In order to decide whether a class can substitute another one, the authors have developed a complete set of rules which are verified by the tool before doing any modification.

About inserting a new class, they expect programmers to recompile and supply the (new) subclasses to the evolution tool. Moreover, if a class **C** is deleted, all its subclasses are now direct subclasses of the superclass of **C**: again, the programmer must recompile affected classes and pass them to the tool. The verifier of the evolution tool will test that all necessary classes are available. PJama supports conversion (when a class changes), migration of instances to another class (which applies when a class is going to be deleted), and bulk conversion (which is provided as a feature to the programmer in order to let him or her modify all instances of a given class).

Conversion functions are available to the user, if he or she selects *custom* (or *programmer-defined*) conversion instead of *default* conversion. In the latter case, the evolution tool will convert instances in the old class definition to instances in the new class definition, just using default conversion rules for fields of given types. Default conversion is used also when the programmer deletes a class. In that case, provided that another class is supplied, then the orphans of the deleted class are converted to the given class.

Custom conversion is useful when more complex transformations are needed: for example, calculating a field which stores sizes in meters to store sizes in centimeters, or even inches … etc.

There are two ways in PJama to do *custom* conversion: *bulk conversion*, when all instances can be changed in the same fashion (the programmer will have to write a *conversion method* in order to perform the transformation (the available *conversion methods* can be found in figure 22)), and *fully controlled conversion*. The last one implies to substitute the default control method for evolution, so that the programmer is in charge of the task of running over the PS, looking for

instances of the class, and it is the responsibility of the programmer to convert all instances, or make the unconverted ones unreachable.

In PJama, *conversion* is also understood in a very wide way. *Migration* to subclasses/superclasses and simple modification is also embraced by this schema evolution mechanism.

The user always has the opportunity to modify the set of instances of a given class, even in the case that there is no modification to that class or to the hierarchy tree of classes, through *bulk conversion.*

```
(a)
public static void convertInstance(C$$_old_ver_ c0, C c1);
public static C convertInstance(C$$_old_ver_ c);
public static Csuper convertInstance(C$$_old_ver_ c);

(b)
public static void convertInstance(C c, NewC nc);
public static NewC convertInstance(C  c);
public static Csuper_NewCsuper convertInstance(c);

(c)
public static void migrateInstance(C c0, Csuper_sub c1);
public static Csuper migrateInstance(C c);

(d)
public static void modifyInstance(C c);
public static C modifyInstance(C c);
public static Csuper modifyInstance(C c);
```

Figure 22. *Available signatures for conversion methods for schema evolution for class C in PJama. These methods are placed in a class with an arbitrary name, and are passed to the evolution tool The (a) set is used when C is modified, the (b) one in case of C being modified and renamed,the (c) one when C 'orphan' instances must migrate to another [sub]class, and (d) when the programmer simply wants to modify all instances of a given class C .*

This is shown in figure 22, in which the methods for all kinds of conversion are listed. These methods must be enclosed in an arbitrary class and be passed to the evolution class, together with the affected classes.

3.8.2.8 The Oberon-D Persistent Programming System

Oberon-D (Knasmüller, 1996) is a persistent programming system which supports the Oberon programming language. Oberon-D also supports schema evolution (Knasmüller, 1997).

Oberon is a general purpose, structured language in the tradition of Pascal and Modula. One of its strong points is that supports modular programming, and its weakness consists of the lack of database capabilities such as persistence or recovery. Oberon is not an object-oriented language, although types and modules can be combined conveniently to have an object-oriented-like programming (Stroustrup, 1991). Anyway, Oberon-D uses in the documentation of the persistence system the term *object* in its general sense. In this section, the term object will be used in the same sense, as any possible Oberon language construct.

The way the system identifies persistent objects is by reachability from a persistent root, which is declared by the user, by registering it with the function: `Persistent.SetRoot(obj, key)`, where *obj* is a plain pointer to an Oberon object and *key* is a string that identifies the object in the PS. Programmers can recover persistent roots from the PS (as well as their transitive closure) by

the function `Persistent.GetRoot(obj, key)`. The root corresponding to *key* is pointed by the pointer *obj*, after the execution of this function.

The implementation of persistence is based on a persistent heap (the Oberon-D term for the persistent store), as well as a transient one, in memory, which is cleaned up periodically by a garbage collector; deleting the objects which are not going to be used anymore, or mapping them to the persistent store if they have been marked as persistent.

Objects are loaded from the persistent heap to the transient heap, and, once they are in memory, they can interact with transient objects. The authors have changed the Oberon compiler, modifying the `nil` trap. When an OID (an *unswizzled* reference, i.e., an invalid memory address) is dereferenced, then an added handler is executed, which determines if the register that provoked the trap is zero or a positive number. In the first case, the normal `nil` handler is called. In the second case, the object loader retrieves the object from the PS.

Making an object persist:

```
PROCEDURE MakeStringPersistent;
VAR s: POINTER TO ARRAY OF CHAR;
BEGIN
            NEW (s, 32); s := "Oberon-D";
            Persistent.SetRoot (s, "myroot")
END MakeStringPersistent;
```

The following source code shows how to access this string afterwards:

```
PROCEDURE AccessAndPrintString;
VAR s: POINTER TO ARRAY OF CHAR;
BEGIN
            Persistent.GetRoot (s, "myroot");
            Out.String (s^)
END AccessAndPrintString;
```

Figure 23. *Example of use of Oberon-D.*

Objects are deleted from the persistent store through a call to the function `Persistent.RemoveRoot(n)`. The objects reachable from that root that are not needed anymore, are removed through a call to `Persistent.Collect`. An example of use can be seen in figure 23.

In order to map objects to the PS, the system provides special objects called *mappers*, which are in charge of flattening and unflattening data structures to disk. Mappers are needed in Oberon-D because, although the language is supported in many systems, each compiler has its own representation of data formats.

This makes the user writing mappers for his or her data structures, out of the primitive types, which are already covered by the default mappers provided by Oberon-D. The system, using structural reflection, allows to inspect the types of the objects.

Although Oberon-D should be classified as an orthogonal persistent system, the way it stores objects to disk is not orthogonal, as the user must specify a mapper. This means that, while all types can be made persistent, the way the user must cope with persistent objects, -though there are not special pointers for them- is different. The second point is the proxy class `Persistent`, which is the interface with the persistent store. The use of this mechanism for providing an interface for persistence support is for example discussed in (Jordan, 1996). The main criticism which could be

done to this approach for representing the PS in Oberon-D is the existence of this barrier between the programmer and the PS. So, although perhaps this could be suitable for PJama, which tries to provide back compatibility with Java programs, in Oberon-D, a system which claims to provide users with orthogonal persistence, is not transparent enough.

3.8.2.9 The E (and SHORE) persistent programming system

E (Richardson *et al.*, 1993) is a persistent programming system over the EXODUS (Carey *et al.*, 1990) OODBMS. The supported programming language is a version of C++ (Stroustrup, 1991). It was developed at the University of Wisconsin.

The basis of E is to combine the expected efficient behaviour of C++ with database capabilities such as persistence. That's why E separates this two different worlds: the data types which are going to be transient are created using the plain C++ `int, float, struct` and `class`, etc … data types, while the persistent ones are created using especial data types, with a 'db' prefix in their names. For example, '`dbclass`' or '`dbstruct`' are valid E types for persistent data. Also, there is support for fundamental types, such as '`dbint`', '`dbfloat`', '`dbshort`' … and so on, with a special '`dbpointer`'. Finally, there is a special support for arrays of *dbtypes*, as well as collections of a given persistent type through the use of `FileOF[T]`, a template of the E library.

EXODUS is a DBMS which provides support for the PS of E. EXODUS is not useful as a DBMS for the final user, as it is provided as a set of libraries and tools which are expected to be able to manage any database need. The E compiler substitutes each dereference to persistent objects in the source code by appropriate buffer handling, and read/write operations to the EXODUS database (Carey *et al.*, 1990). E is expected to optimise this kind of code in order to minimise the use of EXODUS.

E has many characteristics in common with O++ (Biliris, 1993). O++ is a precompiler of C++, which also links programs to an OODBMS.

Development in E was still continued, and the last addition to E was the support for dynamic linking of classes for incremental loading (Vemulapati, *et al.*, 1995).

E has two parallel type systems, in which only objects of the '`db`' type system can persist. This supposes the system is not type-orthogonal, as not all types of objects can persist. Persistent objects must be marked with the '*persistent*' modifier, in order to make E store them in the PS. The authors claim that one could program only with *dbtypes*, as not necessarily all objects created this way must be made persistent. But then, in this case, the '*persistence independence*' principle is violated.

SHORE (Carey *et al.*, 1994) grew out of the E and EXODUS persistent programming system. SHORE includes a persistent store which is able to understand the types of the objects stored within it. One of the new types in SHORE is `directory`, similar to the UNIX directories but full of fine-grained objects, instead of files. The system maintains a strict tree of directories, although allows the use of '*links*' to directories or objects, in the same way UNIX allows symbolic links.

Objects are stored in packets which are known as '*registered objects*'. This consists of a single object –a persistent root similar to the PJama ones-, and other objects accessible from the root but not from outside.

This system can be accessed from multiple languages, and when accessed from C++, it provides two types of pointers. Although this supposes a violation of the '*persistence independence*'

principle, the author thinks that this is justified due to the multiple language support offered.

3.8.2.10 The PerDis Persistent Programming System

The PerDis persistent programming system (Shapiro *et al.*, 1997; Shapiro, *et al.*, 2000) is a platform which makes easier the tasks involved in developing a persistent distributed application with a number of nodes interacting in a collaborative manner.

The authors call this network (the architecture of PerDis) a PDS (Persistent Distributed Store). A distributed garbage collection mechanism, implementing the paradigm of *persistence by reachability*, is used in order to keep the PDS clean of unused objects. Each individual PS in a node is divided in clusters, grouping related objects.

The naming scheme consists of a "directory" for each cluster, identifying each persistence root (the head of a sub-group of objects inside each cluster) with a string containing a user name. The system transparently manages the references between clusters, and between nodes, supporting the pointer dereferencing of languages such as C++.

A running PerDis system consists of the PD (PerDis Daemon), the GCD (Garbage Collector Daemon), and the processes of users. The latter processes access the PerDis system by an API to a library linked to the program.

PerDis is relevant for this thesis because it is the first persistent system which simply presents a distributed PS, in which each node is divided in clusters, and the programmer must explicitly indicate which cluster his or her data is going to be stored in. This is more or less similar to Barbados, as the user must choose which directory his or her data is going to be stored in, through the command `cd()`. Certainly, we claim that the use of a metaphor of directories (against the raw model of clusters in PerDis) provide the programmer with a better degree of abstraction of the system which is running under his or her application.

PerDis has many points in common to Barbados: for example, the division in clusters and the directories naming scheme. Other points are otherwise far from by Barbados, as PerDis doesn't support an environment, and it's only a *middleware*. About this point it is more similar for example to E, although E transparently compiles and links the code.

3.8.3 Object Oriented Operating Systems

Object Oriented Operating Systems are based in objects instead of files as the basis for information storing. This involves persistence in order to be able to retrieve objects related to the state of an application, or even related to parts of the kernel This is resolved in a variety of ways in the systems discussed here.

The *address space* is the range of addresses that a process can refer to in traditional operating systems. In persistent operating systems, the address space refers to the addresses in the PS, as it is the only place of storing. Objects are loaded transparently from the PS to RAM and swizzled on request, so the idea of two separate ranges of addresses is not present for the user of an object oriented operating system. That absence of difference is strengthened if the operating system offers orthogonal persistence.

There are basically two main possibilities when designing address spaces: the first possibility is to consider a single and unique address space, using PID's (Persistent identifiers) 2^{64} or even 2^{128} or more long. This is called the "*Single flat address space*" (Vaughan & Dearle, 1992; Dearle *et al*., 1993).

But it is also possible to partition the address space in some way. This is the most common technique used. For example, Grasshopper, provides a partition of the PS called "*container*"[11]. Grasshopper's model of address space is a "S*ingle partitioned"* one. Although partitions are clear and definite, some regions of these partitions can be intersected and used (in fact, in Grashopper it is possible to map a container inside another container). A third model appears when the partitions don't share any common region: this is the "*Fully partitioned address space*".

These concepts are extensible of course to other persistent systems, even though they are not Object-Oriented Operating Systems. For example, Barbados (Cooper & Wise, 1995; García Perez-Schofield *et al.*, 2001b) provides the programmer with the structures called containers (which are not exactly the same as in Grasshopper), under a thin layer of directories. These containers can communicate with each other, but they can't share regions such as in Grasshopper. Containers in Barbados provide therefore a fully partitioned address space.

3.8.3.1 The Grasshopper Object-Oriented Operating System

Grasshopper is a container-based, distributed, OOOS (Vaughan & Dearle, 1992). Containers in Grasshopper (Lindström *et al.*, 1995) are available as a storing means for the application developer, and not necessarily for the user of final applications (it depends on the application developer in Grasshopper (Lindström, *et al.*, 1994)). Containers are therefore the way to store information in the PS, and a container can access the information stored inside another one by simply mapping the whole container inside it (Dearle *et al.*, 1993).

Grasshopper also includes other characteristics, such as capabilities (Dearle *et al.*, 1994). Capabilities are designed in order to define the characteristics of objects related to protection. Every container needs to present a capability in order to be allowed to invoke another container, for example. This system tries to go further than the typical fixed schema of permissions present in file systems.

In (Dearle & Hulse, 1995), the authors describe a resilience system which claim would be suitable for Grasshopper. This resilience system is based on optimistic checkpointing, consisting of saving the state of all containers (individual address spaces) that are being modified at a given time. They find that the partition of the space address on containers was an useful aid in order to provide resilience. A final description of the main schemes in Grasshopper (naming, recoverability .... etc.) is presented in (Rosenberg *et al.*, 1996).

Implementation of persistence of Grasshopper is largely based on hardware (Vaughan & Dearle, 1992). They used virtual memory mechanisms in order to load referred objects transparently and save them or discard the unreferenced ones.

This is related to the penalties of current operating systems in order to support persistent object systems, in general (Dearle *et al.*, 1992).

Grasshopper doesn't provide support for fine grained -user- objects (this is partially addressed in Lindström, *et al.*, 1994), as this should be provided by the applications running over the system. This is the main difference with Barbados, which provides the user with an integrated environment, and a complete layer of service for objects through C++.

The authors implemented a UNIX-like OS over Grasshopper (called *Hoppix*) as a test in order to demonstrate the maturity of the system (Bem *et al.*, 1996).

3.8.3.2 The EROS Object-Oriented Operating System

The EROS OOOS (Shapiro *et al.*, 1999) includes persistence through a checkpointing mechanism. The checkpointing mechanism synchronises disk with the contents of memory by a

[11] Containers in Grasshopper are related to containers in Barbados, though they don't represent the same concept.

fixed period of time (Shapiro *et al.*, 1996). This makes possible for EROS to recover the last stable state, if a crash happens, for example, or simply restore from the last state when the systems starts up after a shut down.

EROS also provides protection through capabilities, in a similar way Grashopper does (as discussed above). EROS divides memory in segments, in a similar way UNIX-like operating systems use pages for virtual memory. The authors use these segments in order to partition the chekpointing process. They mark all dirty segments, and through a circular log, segments are updated.

On disk, EROS has a complete object store which maintain objects alive when they are not in memory (each one with their one OID), as well as an specific part reserved for chekpoints.

EROS provide the programmer with totally transparent, and therefore orthogonal, persistence. Checkpointing supposes an overhead of only the 0.3% of the tasks pending of being executed.

EROS doesn't provide support for fine-grained (user) objects, as no applications (no persistent systems) are still available for this OS.

3.8.4 Object Oriented Database Management Systems

These database management systems use classes and objects, instead of records and tables, in order to represent data stored inside of them. Many times, they support more or less complete flavours of object-oriented languages in order to provide the user with a DML and sometimes even a DDL. This makes sometimes difficult to difference a PPS from an OODBMS. Many times the only difference is in which part of the system the stress is put.

3.8.4.1 The ORION Object-Oriented Database Management System

Orion (Kim, *et al.*, 1991) is an OODBMS supporting an adaptation of SQL, similar to OQL (Cattel, 1993) -from the Object Management Group (OMG, 1996)- as its DM and DD languages. Orion has also been designed as a single-user, multitask database, intended for Artificial Intelligence applications, multimedia documents and computer aided design.

Orion uses segments in order to group objects in the persistent store. Segments are automatically managed by Orion: they are just composed by a class and all its objects. The system also provides a mechanism for the user to allow him to indicate that special cases, manually.

Orion supports both versioning and schema evolution. In the table 6 the types of changes supported can be found, while the table 7 shows the available changes to methods.

Orion provides a set of invariants developed in order to assure that all changes are done inside a formal framework. The invariants are as follows:

- class lattice invariant: the class lattice is a rooted and connected by a DAG (Directed Acyclic Graph). This lattice has only one root, a system-defined object called OBJECT. There are not isolated nodes.
- distinct name invariant: All instances of a class have distinct names, and all members (attributes or methods) of a class have distinct names, as well.
- distinct identity invariant. This means that all members have a unique origin. This is useful in the case of name clashing when inheriting from more than one class. This way, although we have more than one member with a given name, it is possible to distinguish among them using the origin invariant, and assign distinct names.
- full inheritance invariant. A class must inherit all members from its superclasses, provided this doesn't violates the invariants b) and c).

- domain compatibility invariant. If an attribute *b* of class **S** is inherited from an attribute *a* of class **C**, then the domain of attribute b is the same of the attribute *a*, or a subclass of it.

Attributes	
Add an attribute	No considerations.
Remove an attribute	The modification is propagated to all subclasses of the target class. All values are lost. In the case of a name clash with other attributes in any superclass, that new variable is the one considered as inherited.
Change a name of an attribute	No considerations
Change the domain of an attribute	No considerations
Change the inheritance (parent) of an attribute	Inherit another attribute with the same name.

Table 6. *Schema evolution operations for attributes in Orion.*

The Orion model establishes also a set of rules for schema evolution, taking always into consideration the background of the class lattice preserving all invariants in all situations. These rules are categorised in four different groups: rules related to default conflict resolution, which permits choosing a single inheritance option when there is a name conflict; property propagation, which refers to the modification of the name, domain, default value, shared value, or composite link of a class; DAG manipulation rules, which refer to the manipulation of the class hierarchy; and composite object rules, which refer to the semantics of objects related by the PART-OF relation.

Orion also provides a mechanism to create versions of existing objects in the system. This can be combined with lazy conversion, where the deletion of attributes is filtered in order to avoid the problem of system conversion mechanisms referencing attributes in an object which have disappeared due to its own evolution. Users can't specify conversion functions.

Changes to the class hierarchy	
Make a class s a superclass of class c	No considerations
Remove class s from the superclass list of c	No considerations
Add a new class	No considerations
Drop an existing one	No considerations
Change the name of a class	No considerations

Table 7. *Schema evolution operations for methods in Orion.*

3.8.4.2 The Gemstone/J Object-Oriented Database Management System

Gemstone/J is an OODBMS which supports Java as its DML and DDL languages. The original Gemstone is an OODBMS supporting an adaptation of Smalltalk as the DD and also DM languages. Both versions support Schema Evolution, or in the GemStone terms, "*Schema Modification*". Gemstone is built on the top of the OPAL system. A full set of invariants and

considerations are enumerated in the GemStone documentation (Bretl, *et al.,* 1991), in order to discriminate between allowed/not allowed modifications to the schema. Also, a group of primitives, which can be found (simplified) in the table 8, are provided (the concrete case of methods is found in table 9). The approach of GemStone is to provide a *screening* mechanism for object conversion, until garbage collection is done, and then execute a conversion for all instances, as *conversion* is an expensive task. As it can be seen, *screening* is used as synonym for "*lazy conversion*", while *conversion*, in Gemstone terms, means to convert all instances in the PS (although, as explained, they do not contemplate to do that conversion as soon as the class has changed). GemStone supports the possibility of using conversion functions, which can be modified by the user, in order to let him or her control how the modification in each individual instance is to be done.

The invariants in GemStone were developed in order to provide a formal framework for the development of schema evolution. They basically try to assure that a) all objects are related to a class, b) the class hierarchy is a tree of classes (no isolated classes), c) there are not dangling references and d) there is not information lost when classes change. This means that, if a user modifies a class which has instances belonging to another user, then that other user must be able to rebuild all information lost due to schema modification.

As has been discussed in the introduction to this point, the group of allowed changes can be

Primitive	**Explanation**
Rename	It is possible to change the name of the members of a class, provided the new name doesn't clash with already existing ones.
Add a member data	Add a data member to a class. It is checked that there is not a member data already with that name: in the class or in any of its subclasses. The new member is propagated to all subclasses.
Remove a member data	A member data can't be removed if it is inherited from a superclass. The modification is not propagated to all subclasses of the class.
Remove a class	A class cannot be removed if it has instances. The superclass of all subclasses of the removed class is changed to the superclass of the removed class.
Add a class	Adding leaves to the class hierarchy is allowed. Also it is allowed to insert classes between other ones in the class hierarchy, specifying exactly the classes involved.

Table 8. *GemStone primitives for Schema Evolution.*

separated between changes affecting a class and changes affecting the class hierarchy. Normally, all changes must be decomposed by the user using the provided primitives. For example, in Gemstone it is not possible to delete a class if it has any instance in the PS, but it is possible to delete all instances and then delete the class.

Gemstone/J was mentioned in the introduction to this section. This is a variant of Gemstone (mentioned in Dmitriev & Atkinson, 1999) called "GemStone/J". Gemstone/J has been developed as a continuation of Gemstone, employing Java as its DDL/DML language. Evolution can't be performed in a concurrent way, although Gemstone/J allows multiple processes (multiple Java VM's) to access the same PS. Conversion functions are not available, and the programmer must specify the map (using the standard Java container class) of new and old attributes (i.e., this process is not automatically done, although this way a chance of control is given to the user) and pass it to

Methods	
Add a new method to a class	No considerations
Insert a new method in a class	No considerations
Delete a method in a class	No considerations
Change the name of a method	No considerations
Change the code of a method	No considerations
Change the implementation of a method	No considerations
Change the inheritance (parent) of a method	Inherit another attribute with the same name.

Table 9. *Schema evolution primitives for methods in Orion.*

the schema evolution API, having then the transformation done eagerly.

3.8.4.3 The O2 Object-Oriented Database Management System

O_2 is an OODBMS (Ferrandina & Ferrán, 1995), in which the DD and DM languages are the *O_2C and C++*, a special adaptation of C/C++ to the characteristics of O_2.

The characteristics of this system are compiled in (Banchillon *et al.*, 1992). O_2 is a commercial OODBMS which strong characteristics about schema evolution, as will be explained below.

Clustering in O_2 (Benzaken, *et al.*, 1995) is based in a structure called *Placement Tree*. The purpose of this structure is to be able to adapt the clustering strategy to changes in relations among objects. This means that O_2 supports dynamic clustering.

1. A deleted attribute is ignored (it's not going to be present in the final version).
2. A new attribute is initialized with the default initial values for each type (tipically 0 for integers, chars and so on …)
3. A modified attribute is transformed using certain rules (casts, `sprintf()`, `atoi()` … etc) present in the language.

Figure 24. *Default conversion rules.*

The schema in O_2, is a set of classes related by inheritance and/or composition links. All objects are descendants of the `object` base class. Persistence is achieved by reachability, as the persistent objects are the ones which are attached to a persistent root belonging to the schema. The primitives for schema evolution are the following ones:

- Creation of a new class.
- Modification of an existing class.
- Deletion of an existing class.
- Renaming of an existing class.

- Creation of an inheritance link between two classes.
- Deletion of an inheritance link between two classes.
- Creation of a new attribute.
- Deletion of an existing attribute.
- Modification of an existing attribute.
- Renaming of an existing attribute.

```
create schema Car_showroom;
class Vendor type tuple (name:string,
          address: tuple (city:string,
                          street: string,
                          number:        real),
                          sold_cars: list(Car))
end;

modify   class   Vendor   type   tuple(name:
string,address: tuple (
            street:string,
            number:integer),
      sold_cars: set(car))
end;
```

***Figure 25**. Modification example.*

The list of default conversions are presented in figure 24. An example can be found in figures 25 and 26. Conversion functions can be defined by the user. The system provides a separate mechanism called "*Object Migration*", in order to move objects from one class to another one.

The system provides two ways to modify the object database: the *deferred* and the *elective* upgrade. Elective transformations of the object database can be done simply by entering the command "`transform database`".

```
begin modification in class car;
delete attribute horse_power;
create attribute kW:integer;
conversion functions;
conversion  function  mod_kW  (old:  tuple  (name:string,
price real, horse_power:real)) in class car
{
          self->kW = round(old.horse_power / 1.36);
}
end;
```

Figure 26. Modification example.

Conversion functions defined by the user are always applied after the default conversion functions have been executed (the ones which copy the old fields of the object still present in the new one, whether compatible, and initialise the new ones to their default values).

The authors of this system distinguish between *simple* and *complex* conversion functions: the difference is that simple conversion functions don't use any other information apart from the data in the object being modified, while the complex ones do.

This system has a complete set of invariants and rules (Bréche & Wörner, 1995), which are based on the ones developed for Orion (please see the section about Orion above), as well as the

most complex and complete schema evolution mechanism known by the author of this thesis.

3.8.4.3.1 The problem of complex conversion functions

Conversion functions can be defined for either *eager* or *lazy* conversion. Complex conversion functions have the flexibility of being able to reference objects external to the ones of the involved class, including objects of other classes, pending of conversion. When eager conversion is selected, then only one class is converted each time; when lazy conversion is chosen, then all pending conversions for classes are done as soon as any instance of these classes is referenced.

In the case of lazy conversion, if the system references an object of class **A** pending of conversion, then the conversion functions are automatically called. If these functions are complex conversion functions, then it is possible they refer other object of another class **B** pending of conversion. If the conversion function for class **B** is also complex, it could be possible they reference objects of the former class (**A**): in this situation, we have an infinite cycle.

The same authors have developed a large amount of research in this area, as can be seen in (Ferrandina *et al.*, 1994, Ferrandina & Ferrán, 1995 and Ferrandina *et al.*, 1995): as they establish as an equivalence criteria, their objective is to assure that "*The result of a conversion function implemented in a lazy database transformation must be the same as if the same conversion had been done by an immediate database transformation*". Although this statement can seem obvious, it must be noted that in the example above, the result of conversion depends on the order the user expect the conversions to be done, and on the strategies followed by the system: for example, if the user expects the class **A** to be converted before the class **B**, and the system doesn't try to update objects of class **B** (although **B** instances have been marked to be converted), then the results will be the expected (correct) ones.

But in the case that the real order of execution is not the expected one, then the results will not be (obviously) correct. Moreover, the system can choose between two main options when finding two related, modified classes: to change classes as they are accessed, or to not do it at all until each individual instance conversion has been done. Although the first approach can seem the more correct one, it can lead to an infinite cycle. The second one can only lead to correct results if the order selected to actualise classes is the expected one, which can happen only in a very lucky case.

Initial and simple solutions are to restrict conversion functions to only simple ones, or to launch an immediate transformation when a complex conversion function is detected. These solutions are found by the authors as too restrictive to be real solutions.

Their final approach (Ferrandina *et al.*, 1994) is to use what they call a "*screening and blockable*" algorithm. The "*blockable*" part consists of marking classes being modified in order to be able to avoid infinite loops, blocking those cycled references. The *screening* part consists of marking attributes as deleted or modified ones instead of actually delete or modify them. This way, the information is still there, and it can be used by the conversion functions, avoiding the need of immediate transformations.

3.8.4.3.2 Object Migration

O_2 provides the possibility of changing an object from one class to any of its subclasses. An example is provided in figure 27 for the syntax of the language in order to do this kind of change.

```
class sport_car inherit car
          type tuple (speed:integer)
end;

migraton function migrate_cars in class car
{
          if (self->kW >= 100)
                    self->migrate("Sport_car");
};
```

Figure 27. *Migration function example for O_2.*

The `object` class (the root one) provides a method, `migrate()`, which is called by the migration function defined for objects in a given class (superclass) which are going to be transformed in the class in which is being defined. This is a useful mechanism, when for example, a class is going to be deleted. The objects which are still valuable can migrate to another superclass firstly.

Object migration can be done immediately (again, using the command "`transform database`"), or in a *lazy* way.

3.9 Conclusions

One of the most important worries is normally the performance of all prototypes and commercial products commented above. Performance many times is shown individually for each characteristic of these systems, and must be searched in individual papers of the literature. Some comparatives have been commented above; for example, in (Brown, 1991), many OODBMS are commented and briefly compared. Benchmarks for persistent systems have been also developed, and applied to relevant systems. For example, in the case of OODBMS, the OO7 benchmark is available (Carey *et al.*, 1993).

General performance of some object stores is commented in (Chennupati & Saeidian, 1997), while some other papers (for example, Srinavasan & Chang, 1997), provide the reader of comparative evaluation about some systems.

A comparison with these systems and Barbados has been carried out along all this chapter. In general, Barbados simplifies the orthogonal persistence model in order to obtain a more useful, pragmatic model.

In general, there is a wide range of persistent systems, in the form of operating systems, databases or programming environments. Barbados many times shares characteristics with them (for example, some swizzling mechanisms, the language supported ... etc.) and sometimes, in contrast, is radically different (for example, the clustering mechanism, the *intel-native* compiler, ... etc.).

The tables 10 and 11 summarise many characteristics of the systems described in this chapter. Table 10 compares different persistent programming systems.

Syst./Charact.	*Environment*	*Schema Evolution*	*Interoperability*	*Distribution*
Barbados	X	X		
PJama		X		
JSpin			X	
Napier	X			
IK				X
Arjuna				X
Oberon-D		X		
E				X
PerDis			X	X

Table 10. *Classification of the Persistent Programming Systems discussed in this chapter.*

The table 11 shows a comparative relation of different characteristics for the OODBMS, commented in this chapter. This table is based on another one that appears as part of a wide study about object-orientation and storage in (Chih-Ting Du & Wolfe, 1996). The contents of the table have been updated.

Characteristics/ systems	***O_2***	***Orion***	***GemStone***
Backup/Logging	Yes	Yes	Yes
Multiple Inheritance	Yes	Yes	No
Object Migration	Yes	No	Yes
Program Interfaces	Yes	Yes	Yes
Transactions	Yes	Yes	Yes
Versions	Yes	Yes	No

Table 11. *Brief comparison of some characteristics in OODBMS.*

Chapter 4: Design and Implementation of Schema Evolution in the Container-based Model

4 Design of Schema Evolution in Barbados

4.1 Introduction

Persistence in Barbados is based on a special structure: the container. A container is a group of logically related language-level objects, a data-structure or set of data-structures, which is treated as an object for many system-level tasks. Containers are simply directories of named objects. This way, in contrast to other full-orthogonal persistent systems, in Barbados all objects belong to a container, and although a persistent store can contain many gigabytes of objects and information, containers themselves are expected to be in the order of kilobytes or megabytes. Furthermore, this mechanism, the container, provides Barbados with a natural way to distribute data.

Typically, a Barbados application will consist of a set of *containers* containing software (i.e. mainly functions and classes) while the data (i.e., mainly variables and objects) will be in separate containers. However, it is possible to mix software and data within the one container (as shown in the figure 28; ideally, in container #68, there shouldn't be any object of class 'a'). Also, in the container holding the application software, there will be at least one entry point for the program. An entry point in ANSI C++ is just the *main* function, while in Barbados any public function in a container can be directly invoked.

Moreover, the schema evolution mechanism must take into consideration, and must work on the top of, the structure of containers. The concept of a 'container' in fact greatly assists us in the task of schema evolution, because we can implement a system whereby containers are transformed *lazily,* but at the lower level, inside a container which we have decided to transform, all the fine-grained objects are transformed *eagerly*. This mixed approach benefits from the best characteristics

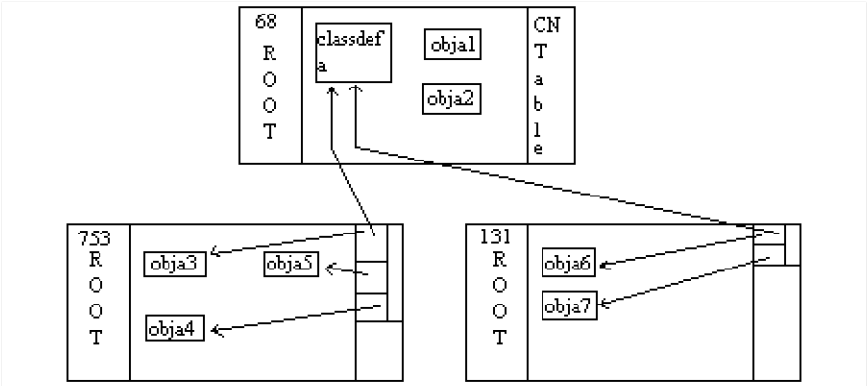

Figure 28. *Objects 'obja1' and 'obja2' are in the same container (68) as well as the classdef for 'a', the class of all of them. Other objects are in other containers (753 and 131), which have appropriate entries in the CN-Swizzling table to container #68.*

of both approaches: the persistent store might be so large that it is infeasible to transform all objects eagerly, and so a lazy algorithm is desired; but on the other hand we don't have the complications and run-time overheads of having objects of different versions of the class being mixed within the one container, waiting for their access in order to be updated.

Containers are independent from each other in the sense that when a container is closed, all their external references are checked and resolved, compaction and garbage collection inside them are done, and the container is left in the PS ready to be loaded into main memory for the next time. When a class in a container is modified, the update of the whole PS can wait until the individual, affected containers are loaded in memory. At that point, the transformation of these containers (the ones holding instances of the modified class) is done.

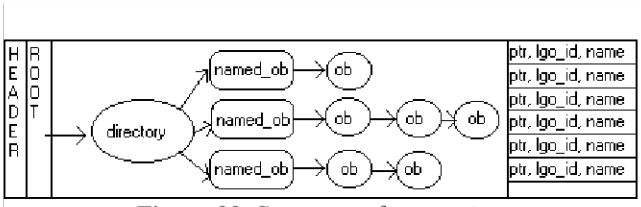

Figure 29. *Structure of a container*

The architecture of containers (as explained in the chapter "*the container-based model of persistence*") is as follows: containers always have a root object, which is an instance of the `directory` *metaclass*. This object is a directory in the traditional sense, i.e. a set of (name,object) mappings. These mappings are called *named objects*. A subset of these *named objects* in this root directory are called '*interface objects*': they are the ones which have a public name, which is global and unique when considered as part of a pair *(container_id, name)*, and furthermore they are the only ones which can be referenced from another container. The rest of the container is not accessible for objects in other containers. This structure is shown in the figure 29.

Programmers are expected in Barbados to put their applications in a container, and data in another one. The first ones will be accessed from the second ones in read mode, i.e., when the data containers are loaded in memory. This takes advantage of the model of containers, and of the CN-Swizzling mechanism, as the data container can reference any function/method in the program container operating over its objects (as shown in the figure 28). Under this view, the 'program'

container would hold the class definitions, while the 'data' container would hold the instances, and conversion of instances inside the same container of the class definition wouldn't be necessary.

Anyway, the evolution mechanisms are able to cope with any situation, not only with the ideal one –from the point of view of Barbados' architecture- described above.

4.2 Overall design

4.2.1 Terminology

- *Prerequisite container*: a container which is needed by the container currently being loaded, to resolve C-N pointers
- *Conversion function*: a function which converts objects from an old format to the new format.
- *Local class*: a class defined in the same container as one of its instances to be converted.
- *Foreign class*: a class defined in a container other than the one containing the instance to be converted.

4.2.2 Summary

The problem of schema evolution will be divided in two parts: (a) detecting the need for conversion before objects are accessed and (b) applying the conversions to the instances of the modified classes. The first part of the (a) point is itself divided into another two parts. The first part consists of detecting schema evolution inside the container, i.e., when it is already in memory and one of the classes inside it is recompiled. The second part is mainly involves detecting schema evolution outside the container, i.e., while a given container and its prerequisite containers are being loaded, and the main container has instances of class defined in one of the prerequisite containers, which has changed.

An object's class can either exist in the same container ('*local class*') or another container ('*foreign class*'). The '*foreign class*' case is the most important case, as it is the most common situation (note that we encourage users to put software in separate containers to the data), and it is the case which allows us to implement an hybrid *eager/lazy* approach for a schema evolution mechanism.

As has been said, containers will be in the order of kilobytes or few megabytes, which makes simple eager evolution suitable for solving schema evolution inside them. However, the complications of relations and synchrony among multiple containers are still present. This is precisely the lazy part of the mechanism, as containers which store instances of modified classes will not be converted until those instances are used –i.e., until their containers are loaded in memory-.

In this design, we the authors will provide application programmers with templates of conversion functions which automatically convert objects in simple cases (e.g. a data member is added or deleted or simply related type-conversions happen), which they can then choose to enhance for extra flexibility, as explained in (Ferrandina & Ferrán, 1995).

Finally, the selected mechanism must take into account two key points in the architecture of the container-based model: the first key point (I) is that, once a container is loaded in memory, this container must be completely ready to work. The second key point (II) is that, once a container is

saved, then the container has referential integrity (and if requested[12], is type-safe), through the mechanism of garbage collection. This implies that a) the evolution mechanisms can't be lazy inside a container, as that would imply to implement a runtime mechanism detecting unconverted objects of modified classes, which is not possible taking into account (I). Also, b) any pending conversion inside containers must have been resolved when the container is closed (II). And c), all schema evolution process among containers must be resolved in the loading phase for containers, *before* the containers are presented to application programmers (I).

4.3 Schema Evolution at Load Time

This case is the one which affects objects in containers which are closed at the time the class is modified. This is by far the most common and important case. It is also the easier case to implement, because we can restrict all schema evolution operations to the loading phase, i.e. at the time the data containers are loaded into memory.

Value	*Meaning*
SEConvert	Converts all objects to concord with the new class definition Quiet mode.
SESplit	Creates the old class in the affected container, separating the two containers, erasing any relation.
SEFail	Loading simply fails. The return value of error is E_SCHEVOL
SEAsk	User is asked about the way to take (the answer is again one of these possibilities, excepting ask, of course).

Table 12. *Available possibilities for the SEvolution parameter of OpenContainer().*

This is the main schema evolution point in Barbados, because as it has been said, schema evolution will have to cope with relations among containers (however, this is not a disadvantage: it offers a natural opportunity for a *lazy* approach, as explained above). As discussed in the chapter about the container-based model and its implementation, the CN Swizzling table (a table which stores all pointers which point to objects in other containers[13]) stores also the *abbreviated classdef* of all foreign referenced classes for a given container. From this abbreviated *classdef*, it is possible to re-build the class, excepting methods ('*member functions'* in C++ notation), which are not stored.

When a container is loaded, all the prerequisite containers (García Perez-Schofield *et al.*, 2001b) that it references, directly, or indirectly are loaded as well (unless of course they are already in memory). They are loaded in read-only mode if they are not already present in memory. Once CN Swizzling and local swizzling is finished, each abbreviated *classdef* (which represents the format of the instances in the container being loaded), is compared with the true *classdef* –the new one- in the other container. If they are different, the evolution process is triggered for those classes.

The evolution process follows the principles outlined in the introductory section. The whole process depends on the argument related to schema evolution in the `OpenContainer()` API function. The possible values of this variable are shown with their respective meaning in table 12. The argument used for interactive operations, should be SEAsk; so for example, the `cd()` command will make the call to `OpenContainer()` with *SEAsk*, in order to make Barbados ask the user what action to take in the case an *out-of-sync* relation is found. *SEConvert* will apply a default conversion without asking anything of the user, provided any class evolution is needed. In that case we assume

[12] Although we don't implement Java yet, we envisage a situation where Java applications and C++ applications are mixed in the one persistent store. This can only work if containers last modified by a C++ program can be certified type-safe and Java compatible.

[13] Under certain restrictions. Only interface objects can be referenced.

that an appropriate conversion function already exists (if not, then it unfortunately becomes an interactive operation). This is useful in the case that containers are being accessed programmatically instead interactively. *SESplit* implies that, if evolution is needed, then the CN relation between the two container is deleted, making the system create a copy of the class following the information stored in the CN relation link. *SEFail* is available as a value to be passed, which means Barbados abort any opening if evolution is needed.

If the *classdef* hasn't been modified, but deleted, then the only possible actions would be to '*fail*', or to '*split*'.

4.3.1 The process

The process will be illustrated following a simple example step by step. The answers of Barbados appear in inverse video. Let the following set of instructions be an interactive session in Barbados:

```
cd(/);
mkdir(test);
Barbados> test: container
cd(test);
mkdir(program);
Barbados> program: container
cd(program);

// Container /test/program

class Counter {
     float count;
public:
     float getCount(void) { return ++count; }
     void reset(void)     { count = 0;  }
};
barbados> class Counter {};

void count10(Counter *& c)
{
     if (c==NULL)
                c = new Counter;
     c->init(0);

     for(int i=0;i<100;++i)
     {
                cout << c.getCount();
                if (i<9)
                       cout << ',' << ' ';
                else   cout << '.' << endl;
     }
}
barbados> void count10(reference to pointer to Counter)

cd(..);

// Container /test/data
mkdir(data);
barbados> data : directory

cd(data);
```

```
/test/data/Counter *c = NULL; // C-N Swizzling relation "data -> program"
                              // between the two containers
barbados> c : pointer to Counter

/program/count10(c);
barbados>1, 2, 3, 4, 5, 6, 7, 8, 9, 10.
*c;
barbados> { 10 }

c->getCount();
barbados> 11

cd(..);
mkdir(data2);
data2: directory
/test/program/Counter c;
c: Counter
```

After these instructions, we will have two containers (*data* and *data2*), which have a relation with the *program* container as their prerequisite container, as can be seen in figure 30. Concepts as '*namedobj'* and so on are deeply explained in the chapter about the *container-based* model.

As can be seen, the software of the application is created in the *program* container, while the data itself (`Counter *c`, i.e., the object pointed by `c`) is created in the *data* container. The table of *AbbreviatedClassdef's* for container *data* has now an entry such as this one:

```
Abbreviatedclassdef: Counter. Referenced in address: 0x00056874
Counter|count|f|
```

…which corresponds with the definition in the source container. This relation can be also seen in

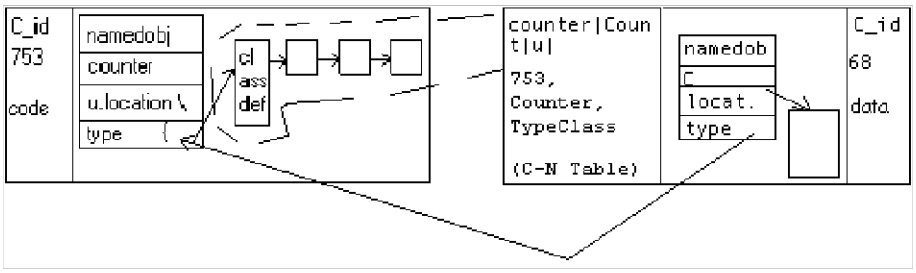

Figure 30. *The relation between the Abbreviated classdef and foreign class definitions.*

figure 31.

The problem of schema evolution appears this time when the class *Counter* changes in the container `/test/program`. Once the user opens again the `/test/data`, the latter one must change (evolve) all instances of class *Counter*.

First of all, we must explain what happens when containers of modified classes are eventually saved to disk. This is naturally and transparently achieved when one container is left without use in main memory. It also happens naturally and transparently when Barbados is shut down.

When the saving process is activated, the containers are written to disk. This implies to do

garbage collection (based on *reachability* (Cooper, 1997)) and compaction. Also, the C-N Swizzling table is appended to the container, which contains the location of all pointers that point to objects outside the container. This table will be used later in order to load all prerequisite containers in the loading process of containers.

The next step happens when the user enters in the *program* container and changes the definition of `class Counter`.

```
cd(/test/program);
edit(Counter);
class Counter
     unsigned int count;
public:
     void init(unsigned int x)    { count = x; }
     unsigned int getCount(void) { return count++; }
     void reset(void)             { count = 0; }
};
```

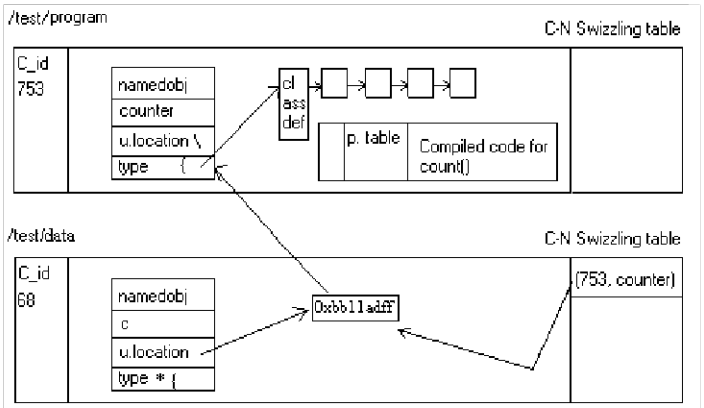

Figure 31. *A diagram of the proposed scenario for the discussion on Schema Evolution*

```
cd(..);
```

The user has changed the definition of the class *Counter –count* is now an unsigned int-, and this means that all the instances of the class counter are out-of-sync. In this very simple example, we only have one instance of `Counter` in the `data` container pointed by '`c`'. This instance must be converted once the `data` container is loaded in memory. This container has the *AbbreviatedClassdef* for class *Counter* in the CN Swizzling table, because of the link in the `/test/data/c` pointer to container #753[14], as this object is an instance of `/test/program/Counter` class. For the sake of simplicity, this is a very tiny and easy example.

As explained before, the need for Schema Evolution is detected precisely when loading containers. In this example, this happens when the user types '`cd(data);`' in order to enter the data container. The action to take is to change all instances of changed data structures –in this case, 'c'-.

Once the need for schema evolution is detected, there are 3 courses of action the computer

[14] Container identifiers are managed transparently, so they are hidden for the user.

can take: '**Transform**', '**Split**' or '**Abort**'. The choice of which of the 3 possibilities is taken is based on the parameter passed to `OpenContainer`(): in the present case, in which the container is opened through a cd() command, the user will be asked which one of the 3 actions he requests.

- '**Convert**': the conversion mechanism is triggered, and following the steps described in the above points, the instances are converted using a conversion function modifiable by the user.
- '**Split**': the system creates a *classdef* (and an appropriate *namedobj*) in the container of the instance, based on the *abbreviated classdef*. The container being loaded is explored (as it is when the '**Convert'** option is taken, too) for objects of the evolved class (`Counter`, in this example), in order to make them point to a new class in their same container: in the example, this class is created in the `program` container, which is called `Counter`. The pointer to the foreign container is eliminated, simply by making all objects in the container point to the recently created class. This build is possible because of the availability of the *AbbreviatedClassdef* in the CN Swizzling table.
- '**Fail**': the `OpenContainer`() call fails with an error code of E_SCHEVOL.

If the '**Ask'** possibility is taken, then the user is asked to take one of the remaining actions.

In our example, the '*convert'* possibility is taken by the user, who is prompted for a conversion function, as shown in the following lines.

```
cd(../data);
Barbados> class Counter doesn't match previous definition:
<C>onvert, <S>plit, <F>ail: C

void convertInstances(Counter_$$old *old, Counter *new)
{
        new->count = old->count;
};
Converting instances ...
Done.
```

The user accepts the given conversion function (the default one, as the class *Counter* hasn't been converted before) as the correct one. The function is provided assigning the common and compatible data member items from the old to the new object. In this case, the unique, compatible, data member in both classes (the new and the old one) is '*count'*, an `unsigned int`. The conversion function, however, is totally modifiable by the user, who is able to even delete that line. For example, the user could behave this way:

```
cd(../data);
Barbados> class Counter doesn't match previous definition:
<C>onvert, <S>plit, <F>ail: C

void convertInstances(Counter_$$old *old, Counter *new)
{
        new->count = 10;
```

```
};
Converting instances ...
Done.
```

Or even any other complex processing suitable to be implemented in a function. These conversion functions doesn't follow the normal rules about encapsulation, as they need to be able to modify all possible data members present in the new and in the old class definition, not just the public ones. The way to achieve this depends basically on the language supported. This is deeply discussed in the section below about conversion functions.

Another possibility about conversion (the conversion API will be discussed in detail in the next section) would be:

```
cd(../data);
Barbados> class Counter doesn't match previous definition:
<C>onvert, <S>plit, <F>ail: C

void convertInstances(Counter_$$old *old, Counter *new)
{
        new->count = 10;
};
Converting instances ...
Done.
void deleteLittleCounters()
{
            Counter *it;
            InstanceIterator cit("Counter");
            it = cit.getFirstInstance();
            while(it != NULL)
            {
                        if (it->getCount() < 20)
                                    delete it;
                        it = cit.getNextInstance();
            }
}
deleteLittleCounters: function (void) returning void
deleteLittleCounters();
```

In the last example, the user creates a C++ function which uses the `InstanceIterator` class, provided by the conversion API, in order to run over all instances of class `Counter` in the container and delete all of those with a count less than 20. (In this second example, we hope that the destructor for the instances performs the work of removing them from whatever data-structure they are part of).

Now, let's suppose that the user opens the '`data2`' container programmatically, the container created with the only contents of one instance (a public object) of the class *Counter*.

```
container &d2 = OpenContainer("../data2", READWRITE, SEConvert);
```

```
Barbados> class Counter doesn't match previous definition.
Converting instances ...
Done.
CloseContainer(d2);
```

In this case, the system finds the previous conversion function, which is stored in the `/test/program` container, and it automatically applies it. So, nothing is prompted to the user.

As another example of the schema evolution capabilities, let's suppose that another modification is done in class *Counter*.

```
cd(/test/program);
class Counter
       unsigned int count;
       unsigned int limit;
public:
     void init(unsigned int x)      { count = x; }
     unsigned int getCount(void)    { if ((++count) > limit) reset();
                                      return count++; }
     void reset(void)               { count = 0; }
     void setLimit(unsigned int x) { limit = x; }
};
cd(..);
```

Now, containers *data* and *data2* are out-of-sync again. The user goes again to the *data* container.

```
cd(../data2);
Barbados> class Counter doesn't match previous definition:
<C>onvert, <S>plit, <F>ail: F
Barbados> Schema Evolution Error
pwd();
Barbados> /test/program
cd(../data);
Barbados> class Counter doesn't match previous definition:
<C>onvert, <S>plit, <F>ail: C

void convertInstances(Counter_$$old *old, Counter *new)
{
            // new->count = old->count;
            // new->limit = 0;
       new->reset();
       new->setLimit(old->count);
};
Converting instances ...
Done.
```

The two first lines, commented by the user, are filled by the system. The user decides to get rid of them: he or she has to give a new value to a new data member, *limit*, which has been added to the class *counter*. The user chooses to give it the same value the *count* data member had in the *old* instance. The way to do this was to call the *setLimit*() method, instead of assigning directly the wanted value. This is of course an available possibility.

The user changes now to the `data2` container:

```
cd(../data2);
Barbados> class Counter doesn't match previous definition:
<C>onvert, <S>plit, <F>ail: S
Barbados> Creating counter class in the present container ...
Barbados> Splitting container. Detaching class definition from c_id 753
Barbados> Done.
edit(Counter);
class Counter {
        unsigned int count;
        unsigned int limit;
};
```

In the example above, the user chooses to cut the relation between the program and the data container, choosing *'split'* as the action to take. Barbados creates this way the class *Counter* as it was described in the *AbbreviatedClassdef*, and then iterates through all objects in the container, changing their *classdef* links to the new one when they belong to the C*ounter* class. The user is able to edit then the *Counter* class, as it exists now in the present container, and modify it. However, the body and the prototype of the member functions are not present, as this information is not stored in the *AbbreviatedClassdef*.

4.4 Schema Evolution inside Containers which are already Open

This section discusses what happens to objects in containers which are open at the time a class is modified. This case is important because a) in the container-based model, it is possible to use a cache of containers, as *Barbados* does. This means that sometimes, a container with instances of a modified class is in memory and therefore the mechanism of SE between containers won't apply. Also, but less important, b) it is possible to find instances of a modified class in the same container.

Containers are converted *lazily*, if they are not in memory. However, if the container is already in memory, then the conversion is done *eagerly*. The point in which the system decides conversion is needed is when a class is compiled, but another version of the class already exists.

It must be noted that, although a container is not limited in size, (it is effectively limited in size to the available RAM and strictly limited to the available virtual memory), typically it will be much smaller than a gigabyte. Containers can range in size from very small to very large and the efficient use of containers ranging from 512 bytes to 64 megabytes is supported. The number of objects will be limited, in the sense that it will not be needed to cope with the whole PS, only with objects inside a set of individual containers.

The system iterates through the container in order to find all instances of the modified class. If no instances are found, then the schema evolution mechanism finishes. In other case, a conversion

function is prompted to the user and the conversion of all instances in the container is done.

4.4.1 The process

The process in which evolution applies in this case happens solely when the user modifies an existing class. The way to do it is just recompilation, as it can be seen in the lines below. In this example we return to the moment in which the class *Counter* is modified to have a *'limit'* extra data member.

```
cd(/test/program);
Counter mycounter;
Barbados> mycounter: Counter
mycounter.reset();
mycounter.getCount();
Barbados> 0
mycounter.getCount();
Barbados> 1
mycounter.getCount();
Barbados> 2
edit(Counter); // This can also be done by simply re-typing the class
class Counter {
        unsigned int count;
        unsigned int limit;
    public:
        void init(unsigned int x)      { count = x; }
        unsigned int getCount(void)    { if ((++count) > limit) reset();
                                         return count++;}
        void reset(void)               { count = 0; }
        void setLimit(unsigned int x) { limit = x; }
};
```

Now, the user has compiled an existing class, which triggers the mechanisms of schema evolution, as the *classdef* is already there. The difference now (from the examples in the last section) comes from the fact of the existence of an instance of *Counter* in the container program. This makes *Barbados* ask for a conversion function and find the affected instances.

```
Barbados> class Counter doesn't match previous definition:
<C>onvert, <F>ail: C
void convertInstances(Counter_$$old *old, Counter *new)
{
        new->count = old->count;
        new->limit = 10;
};
Converting instances ...
Done.
```

If the *'Fail'* option is taken, then the objects inside the container are not converted, and the returned error code is E_SCHEVOL. The need for schema evolution will be detected again the next

time the container is accessed.

As before, a conversion function is proposed and fulfilled with the common fields between the old and the new *classdef*, and initialisation of the new fields.

4.5 Converting instances

4.5.1 Introduction

Barbados allows classes to be recompiled. The system will detect that a class definition has been recompiled if an already existing one with the same name is found in the same *container*. Also, the system will detect that conversion is needed if a container has instances of a class defined in another container which are found to be out-of-sync, while loading the former one. After detecting one of these two situations, the system will trigger the conversion process, presenting to the user the conversion functions to be used, and converting every instance. So, the process of conversion, in these two cases (the only possible ones) is always the same one: it happens inside, at least, one container, and the employed algorithm (explained below) is an *'eager'* one.

Another keypoint which contrasts to other systems described in the "*state of the art*" chapter, is that Barbados doesn't offer a specific language in order to perform changes in classes. As previously explained, it all happens as a result of the (a) C++ recompilation of the class, or (b) loading a containers with out-of-sync instances, which both lead to (c), the application of a user-defined conversion –initially fulfilled by the system- function written in C++. We the authors think that the addition of a new, auxiliary, language would be confusing, and unnecessary.

4.5.2 The algorithm

The process will use two tables: the *table of changed classes* and the *table of relocated objects*. The first table is needed in order to know which objects are affected by the evolution process. This table will consist of the initial modified class as well as the derived ones, and also those which have member objects of the affected classes, if they aren't pointers. The "*table of relocated objects*" is a table used in order to know which objects have been relocated in the container. Objects are relocated if the class has increased in size and a given object can't be increased in size in its existing location in the heap of the container.

About the table of relocations, this table must contain the old address of the relocated object, its new address and size. The system must be able to browse this table as well as it must have a mapping in order to be able to associate pointers in the actual objects and the old addresses of the relocated object. This is because in C++, it is possible to reference not only an entire object, but also any public member within it.

Once the Conversion Process is triggered, it is divided into the following steps:

1. Create the *table of classes*, inserting those classes needing evolution. These classes are the directly modified ones and also any other class which inherits or contains a member of that class.
2. For each class in the table of classes:
    i. Create the *table of relocations* as an empty table.
    ii. If evolution happens at load time, then create the old *classdef* from the *AbbreviatedClassdef* in the CN Swizzling table. This rebuilt *old classdef* will be deleted after the conversion process.

iii. Rename the old classdef with its name plus the suffix `_$$old`. If the class has members, pointers, of its same type, then rename their type as well.

iv. If conversion takes place in the same container of the modified class (i.e., it's a recompilation), take note in the "*table of relocations*" of the new *classdef,* new position and size if it has changed its location.

v. Iterating through the container, find all instances of this class, and if any is found:

    1. Prompt for a conversion function: if a compatible conversion function is found in the container of conversion functions, then present it as a template. Or else, make the default template and present it to the user.

vi. Convert any instance found, using the conversion functions. For each found instance:

    1. Make a copy of the instance in a temporal place.
    2. Replace the old instance by a new space of the size of the new *classdef*, if possible. If not, then allocate space in another place in memory, and use the table of relocations.
    3. Then, conversion is done by applying the default transformations and initialisations, and then applying the `convertInstances()` function (i.e., the conversión function) to the new instance and the old one. No *constructor/destructor* method is called (because the conversion function will fix up anything needing fixing up). The instance will have to have its *classdef* pointer changed to the new one.

vii. A second pass over the container will fix every pointer pointing to any relocated object, using the *table of relocations*. This is done by iterating over all tiles in the container.

viii. Delete the old *classdef*, as it won't be needed anymore.

4.5.3 Conversion functions

Barbados offers a complete *Conversion API* which include the conversion function. This is because we want to give to the user the opportunity to control the whole process; an iterator class and a conversion function, seems to be for us the best way to cover the two main tasks: find instances and later convert them. The first part of the process is not modifiable by the user when Barbados does a conversion due to evolution. This is because we want to offer an easy minimum for schema evolution matters. The iterator class is available for the user when he or she wants to do a bulk conversion (Dmitriev, 1999), i.e., a conversion of all instances of a given class not due to schema evolution.

	int	float	double	char	char *
int			assign	assign	atoi()
float	assign		assign	assign	atof()
double	assign	assign			atod()
char	cast	cast	cast		
char *	itoa()	fcvt()	fcvt()	assign	strcpy()

Table 13. *Compatible conversions for template conversion functions and automatic conversions.*

The interface for class modification is therefore strongly inspired by the one used in PJama (Dmitriev, *et al.*, 1999). Programmers will use a conversion function which will be in charge of the conversion part of the evolution process. However, there is not a conversion tool, a command line program, in order to make the persistent store evolve. This part of the evolution mechanism is just an user program.

```
void convertInstances(a_$$old *old, a *new);

class InstanceIterator {
            InstanceIterator(char *classname);
            void *getFirstInstance();
            void *getNextInstance();
}
```

Figure 32. *Conversion API function and classes.*

The whole Conversion API for Barbados is shown in figure 32. The automatic conversion when a SE need is detected is carried out by the system: the only task the user must do is to modify appropriately (if needed) a conversion function. This is done through the *convertInstances*() function. When conversion is not automatic, when the user starts a bulk conversion of all instances of a given class, then the *InstanceIterator* class is used in order to safely find all instances. In the latter case, no special conversion function is used: any conversion must be done through ordinary functions.

Returning again to automatic conversion, the *convertInstances*() function accepts two parameters which represent the old instance and the new one. The programmer will be able to modify the instance with a high degree of freedom.

The body of the conversion function will be initially created by Barbados. The programmer can add or delete all the additional statements he or she thinks are necessary, using the full power of C++.

In many cases this initial version will not need much modification. For example, in the examples shown in sections above, `count` is the only data member in the class *Counter*, and Barbados will automatically create the source line of code "`new->count = old->count`". All data members would be referenced in this function this way, in the case of having more than that one. The conversion function which only assigns the compatible (please see table 13) data members from the old instance to the new one is known for us as the *Common Field Mapping* function, as it creates an exact copy of the common fields of the two class definitions (the new one and the old one). The *Common Field Mapping* conversion function is always prompted to the user (although it

can be empty sometimes), who is able to modify it in any possible way. When in the new class definition are fields which don't exist in the old one, then these fields are always set to zeroes, and appropriate initialisation lines are written to the default template of the conversion function to be presented to the user.

Conversion functions, once they are created by the user from the given template, are stored in a special subdirectory of the container in which the affected class exists. This way, the subdirectory `.convfunctions` of a given container will store the conversion functions used by the classes in that container. We will rely in the plain function overloading capabilities of C++ to store functions with the same name, distinguishing them by their different formal arguments. So, the programmer will be prompted with the last conversion used for a given class, provided that class is modified again, instead of the default template.

In order to be able to modify appropriately the instances, these methods can't follow the normal encapsulation rules. It must be noted that the objective is to put all data members (the state of the object) in a ready-to-use status. Although some classes can have methods designed for this purpose (initialisation), many classes, precisely because of data hiding, don't. As Barbados supports C++, conversion functions will therefore be marked as friend functions of all the affected classes, assuring the access to all data members. Similar techniques would apply for other languages.

Barbados' conversion functions are complex ones, as the same operations that are allowed for plain methods are allowed for them, including reference to external data to the method present in the container. This raises the problem of complex conversion functions becoming cyclic or making lazy conversion different from the eager conversion. As evolution inside containers is done in an *eager* way, the problem will only be found when converting classes among containers. This happens in the situation in which there are two (or more) modified classes, they must be converted, and the conversion functions for each class reference objects of the class pending of conversion or already converted. Barbados wouldn't fail in an infinte loop, as it only converts instances of one class in each go, but the results would be different depending on the order in which they are converted, which is not acceptable, as the statement "lazy conversion must produce the same results eager conversion does", would be violated (Ferrandina *et al.*, 1994).

The solution in the container-based model is, as the evolution mechanism is started once all *swizzling* has been done, and all classes needing of evolution have been detected, to avoid compiling conversion functions if they reference objects of a class which is within the set of classes affected by evolution.

4.6 Example

This example tries to give a real application of the evolution mechanisms designed for *Barbados*.

```
/faculty/
      program/          * Classes and functionality in general
      data/             * all instances and creation of new ones.

Source code:
cd(/);
mkdir(faculty);
cd();
mkdir(data);
mkdir(program);
```

Figure 33. *The structure of containers needed for this example.*

Code will be stored in one container while data will be created in a second one: this second

container will be used for class instances.

As can be seen in figure 33, the `faculty` directory has been created along with two sub-directories. The process needed to create this structure is also shown.

```
cd(/faculty/program);

const int MAX = 125;

class item { public: char name[MAX]; };

class person: public item {
public:
        char address[MAX];
        char phone[MAX];
};

class teacher: public person {
public: int level;
};

class student: public person {
public: int course;
};

class subject: public item {
public: int course;
};

class assign_subjects: public item {
public:
             teacher **lecturer;
             student **students;
             subject * subj;
};
```

Figure 34. *Initial set of classes for the example.*

The purpose of the application is to address the management of a faculty, simplified in order to adapt it to an example. The program must register new students, new teachers, new subjects, and to associate students and teachers with subjects. This way, listings, reports and other information can be shown. The structure used is the one shown in figure 34. Again, these classes are very simplified, and methods are not shown.

In the container `. . ./data`, all instances of classes *person*, *teacher*, *student* and *subject* will be created, as shown in figure 34. The structures holding that information are implemented as hash tables. Also, in the container `. . ./program`, all parts of the program will be created. A possible listing of functions of the program can be found in figure 35.

```
class reports {
public:
            static void students(void);
            static void teachers(void);
            static void subjects(void);
            static void students_per_subject(void);
};
class certificates {
public:
            static void student_certificate(char * name);
            static void teacher_certificate(char * name);
};
```

Figure 35. *Example of the possible functionality in the program container.*

The need for Schema Evolution is detected precisely when loading containers. The action to

take is to change all instances of changed data structures. So, in this scenario, the `/faculty/program` container is going to be changed sometime, in some way. Then, once the `/faculty/data` container is loaded again, the changes will be applied to the objects.

The programmers decide they need the following information in the *student* class: the *birth_date* and the *beginning_date* in order to be able to calculate the age of students, as well as be able to know when they started to study there. So the programmers would enter the *program* container and modify the classes as shown in figure 36.

```
cd(/faculty/data);
class date {
    public: int day, month, year;
};
edit(student);
class student: public person {
public:
       int course;
       date birth_date, beginning_date;
};
```

Figure 36. Classes modified in the data container

As can be seen in figure 36, to update a class definition, the user only needs to recompile it (by editing it or retyping it). The next step is to go to the container in which the instances exist, and help Barbados upgrade them. Conversion will be done when the class is recompiled in its container, but there are no instances in this one.

Then, as shown in figure 37 (Barbados answers are in inverse video), programmers are asked whether they want to '**convert'** instances, to '**split'** the container (which consists of creating a copy of the *classdef* in this container, detaching it this way from the *program* one), or to **Fail**, in which case the *cd* command wouldn't complete (instead an error code would be returned).

The conversion function is presented to the user, although he can change it in any way he wants.

```
cd(../data);
Barbados> class student doesn't match previous definition:
<C>onvert, <S>plit, <F>ail: C

void convertInstances(student_$$old *old, student *new)
                      {
                                    strcpy(new->name, old->name);
                                    strcpy(new->address, old->phone);
                                    strcpy(new->phone, old->phone);
                                    new->course = old->course;
                                    memset(new->birth_date,          0,
sizeof(../defs/date));
                                    memset(new->beginning_date,      0,
sizeof(../defs/date));
                      }
};
Converting instances ...
Done.
```

Figure 37. *Conversion process being done.*

4.7 Implementation of the Schema Evolution Mechanism

4.7.1 Introduction

The implementation consists of a layer over the layer of Containers in Memory (as has been seen in the chapter about the container-based model). The `OpenContainer()` method of the *Conim* class must be modified in order to check type matching errors for classes. This is done through the use of the class for management of the *AbbreviatedClassdef's* (figure 38), in the table of CN Swizzling (figure 39) of the open container, and the *SchevolManager* class, which gathers all management related to schema evolution (figure 40).

Also, the compiler must be modified in order to detect the existence of a *classdef* for the class which is being compiled (which would mean that it is being recompiled) in the current container. In that case, the compiler must apply the schema evolution mechanisms to all containers in memory.

In this section, the implementation of Schema Evolution in Barbados will be presented. The current state of the implementation consists of modification of single classes. Classes and code here have been simplified.

4.7.2 The ReducedClassdef class

This class (shown in figure 38) is used in two tasks: the construction of the CN Swizzling table, and the comparison of the *AbbreviatedClassdefs* in the CN Swizzling table with the actual *classdefs* of classes in other container, once the main container has been loaded in memory.
In order to do that comparison, the *CNSwizzling table* is employed. In fact, the *CNSwizzling table* is a valuable input for the mechanism of SE. As shown in figure 40, the SchevolManager takes as argument for its construction precisely a pointer to that table.

```
class ReducedClassdef {
                public:
                        class field {
                        public:
                                field(const field &x);
                                field &operator=(const field &x);
                                ~field();
                                char *getName(void) const;
                                char *getType(void) const;
                                void setName(const char *&n);
                                void setType(const char *&t);
                        };
                        ReducedClassdef(const char *&desc);
                        ReducedClassdef(const ReducedClassdef &);
              ReducedClassdef(classdef_type);
                        char *getName() const;
                        char *getDescriptor() const;
                        char *getTotalDescriptor() const;
                        const int  &getTotalSize() const;
                        field getFirstMember(void);
                        field getNextMember(void);
                        bool compareTypeOfCurrentMember(namedobj_type);
                        char *toStr(void);
};
```

Figure 38. *The ReducedClassdef class.*

The important task carried out by the class `ReducedClassdef` class, related to SE, is obviously to allow comparisons between the *classdef* it represents with a real *classdef*. The *AbbreviatedClassdef* is only a string, a sequence of chars, in which the fields are separated by '|',

vertical bars, each field including the name and the type of each data type.

A *classdef* is composed by a *classdef* class and a linked list of named objects (*namedobj*), which represent each one of the fields (as explained in the chapter "*the container-based model of persistence*").

The `ReducedClassdef::field` class represents a field of a given class, in text, and is used by the methods `getFirstMember()` and `getNextMember()`. This way each field is extracted from the *AbbreviatedClassdef* one by one. Another member, `compareTypeOfCurrentMember(namedobj *)`, compares the last member extracted from the *AbbreviatedClassdef* with another, real one, passed as argument. This way, it is possible to assure through a simple loop if a `ReducedClassdef` object represents a given *classdef*.

`ReducedClassdef` objects can be built from other `ReducedClassdef`, from a string representing a `ReducedClassdef`, and from a pointer to an actual *classdef*. In order to build the CN Swizzling table, the `toStr()` method is used.

```
class CNInfo {
                public:
                      bool isAbsolute()  const;
                      bool isRelative()  const;
                      bool isTypeClass() const;
                      bool isCorrupted() const;

                      int  getPtrListSize(void) const;
                      void *getPtrIndx(int) const;
                      container_id getForeignContainer(void) const;
                      namedobj_type getNamedobj(void) const;
                      char * getReducedClassdef(void) const;
                      char * getName(void) const;
                      static CNInfo *loadCNInfo(char *&);
                      void           saveCNInfo(HANDLE) const;
                      void rebasePtrs(int);
};
class CNInfoList {
                      CNInfo **list;
                public:
                      CNInfo *lookforRegNamedObj(container_id,
                                          namedobj_type, char) const;
                      CNInfo *addPtrToNamedobj(namedobj_type obj,
                                          char Type,
                                          void *ptr,
                                          container_id cont = 0);
                      CNInfo *addCNInfo(CNInfo *reg);
                      int getNumberOfEntries(void) const;
                      CNInfo *getEntryIndx(int) const;
                      CNInfo *operator[](int) const;
                      void deleteRedundantRClassdefs(void);
                      void clear(void);
};
```

Figure 39. *The class CNInfo and CNInfoList, which manage the CN Swizzling table.*

4.7.3 The SchevolManager class

This class is the engine of the schema evolution mechanism. In the figure 40, the public interface and a few private methods are shown for the `SchevolManager` class. The schema evolution mechanism is triggered in loading time. The first step is to create a `SchevolManager` object, with a pointer to the *CNSwizzling* table as argument. While compiling *classdefs*, the static method `int convertInstances(classdef_type, classdef_type, Conim *)` is used. It accepts the old *classdef*, the new *classdef* and the container in memory which holds the instances. The

```
class SchevolManager {
            protected:
              static tile_type
              convertInstance(tile_type, classdef_type, classdef_type);
              static classdef_type
              createClassdefFromRC(ReducedClassdef*, Conim *);
              static void
              removeCreatedClassdef(classdef_type cd);
              static namedobj_type
              lookForMember(classdef_type, char *);
              static int getNumOfDataMembers(classdef_type);
            public:
              SchevolManager(CNInfoList *);
              bool checkReducedClassdef(CNInfo *);
              void evolveContainersIfNeeded(void);
              void convertInstancesForEntry(CNInfo *);
              static int convertInstances(classdef_type,
                                          classdef_type, Conim *);
};
```

Figure 40. *The ShevolManager class.*

mechanisms at loading time also use this method, but they have to create the *classdef* represented by the corresponding `ReducedClassdef` before. This is done through the method `classdef *createClassdefFromRC(classdef *, Conim *)`. This method interprets the type string, creating a complete *classdef* from the information stored in. For example, for a string containing the type information "`a|x|i`", this method will create a *classdef* for a class called 'a' with a member field called 'x', of type `int`.

The `evolveContainers`() method does various calls to `checkReducedClassdef`() for a given entry. This method uses the `lookForMember`() one in the *classdef* to conclude if the `ReducedClassdef` is representing the current classdef. If not, the `convertInstancesForEntry`() method is called, creating a new `classdef` through the method `createClassdefFromRC`(). Finally, the `convertInstances`() method does the conversion.

4.8 Conclusions

A complete design of the evolution mechanism for Barbados has been presented in this chapter, including an *step-by-step* example. *Containers* are presented as a natural way to implement a mixed approach between *eager* and *lazy* conversion of objects.

Conversion only affects to member data changes. For example, the modification of member functions or the addition of a member function will not trigger any conversion. This is because we think that the 90% of changes are due to modifications in the implementation of member functions. This is particularly true in the case of software being developed from the beginning, in which constant modifications of the code inside classes are carried out.

We claim that containers are a useful abstraction which justifies the needed relaxation in the orthogonal rules for persistent systems, as objects of modified classes are converted when the containers in which they live are loaded in memory, while, on the other hand, the objects in the same container in which the class exists are converted eagerly. This provides users with a high flexibility mechanism.

Chapter 5: Performance tests in Barbados

5 Performance tests in Barbados

5.1 Introduction

The present chapter describes a performance test which was carried out in order to prove that the prototype accomplishes our claims about the container-based persistent model.

The objective is to test if Barbados, with its native compiler, and its implementation of persistence is really as efficient as we expect it to be. i.e., whether it is comparable to other non-persistent systems or not.

The test is described in the next section, firstly in its plain C++ version and later in its port to Barbados, and a study of the performance results. After the next section, the conclusions and the future work are discussed.

5.2 The Test

We chose our test program keeping in mind that it should be able to run both inside

Barbados and outside Barbados. This way, it would be also possible to obtain feedback of the execution of the two -slightly different- versions of the performance program, mainly regarding to execution times. Another requirement was that the test uses a considerable amount of data, in the order of megabytes. This way, we would test the capabilities of Barbados working with a large container in its persistent store. At least, the persistent store would be larger than the size of the typical tests we ran in order to do functional checks.

The firm *Smarts Pty Ltd*, of which Tim Cooper is one of the directors, creates software which deals with those requirements. The software tracks the movements of the financial market, looking for insider trading and market manipulation and firing alarms if necessary, among many other capabilities. We found that a part of the functionality of this software could be used as a test program, as it fits the requirements we stated before. Also, *Smarts* has enough data filed which would make possible for us to use it as input for the program, and to compare different inputs.

Finally, we decided to create a non-persistent C++ program implementing the performance test, based on the *Smarts* software, and later port it to Barbados (García Perez-Schofield *et al.*, 2001b). This way, we would know whether a) Barbados is able to run a real application managing megabytes of data, b) the port of a real application to Barbados is as simple as we thought it was, and c) the performance of the C++ compiler of Barbados, is comparable with a commercial one.

We presented the claim of Barbados being very efficient because its particular architecture and other characteristics, in (García Perez-Schofield, *et al.*, 2001c), and we partially demonstrated this fact in (García Perez-Schofield, *et al.*, 2002).

5.2.1 The problem to solve

The data which feeds the *Smarts* software comes in the form of messages. Messages are stored by one of the *Smarts* subsystems in a file. This file is read continuously by the subsystem in charge of keeping track of the market. Each message in this file has a specific function, relating to the creation and deletion of orders and trades. An order to buy shares is nicknamed '*bid*', and an order to sell shares is nicknamed '*ask*'. The table 14 shows the main message types.

Message name	Function
ENTER	Create an order. The order can be an *ask* (sell) or a *bid (*buy*)*. This is represented appropriately in a flag. This message also incorporates the number of shares to be sold or bought, and their price.
TRADE	This message includes the order-id of a *bid* and the order-id of an *ask.* It means the orders have matched, resulting in a trade. It implicitly means that the relevant orders will be deleted or reduced in size.
AMEND	Modify some fields of an order (usually price or volume). This message is relative to an *enter* message, and can therefore affect to a *bid* or an *ask*. This is represented again by a flag. The function of this message is to modify the price of the *bid* or *ask* or the volume of shares involved.
DELETE	This message includes an identifier of a previous ENTER message, and means that the ENTER message must be removed from the system.

Table 14. *Financial messages managed by the software*

Our test program corresponds to the Smarts subsystem that keeps track of the market. The purpose of the program is to run over a complete file of a given day in the financial market (between 20-50 megabytes of data), and for each stock (roughly a stock is associated to each firm in the market), build two separate lists of *bids* and *asks*, ordered by their price. These two lists for each stock will be stored in a special structure which is called, in the *Smarts* terminology, an "*orderbook*". These bids and asks will be processed, applying the ENTER, AMEND, DELETE and TRADE messages, as soon as these messages are read from the file. As output, it displays the number of *bids* and *asks* left over at the end of the day of a given stock.

5.2.2 The non-persistent C++ program

The non-persistent C++ version of the program was created before the Barbados one (remember that one of the objectives was to know how complex a port would be). This program uses three modules: the *structures* module, the *fav* module, and the *orderbook* module.

The structures module defines the *list*, *orderedlist* and *inthash* classes. *list* and *orderedlist* are used in order to create the lists of *bids* and *asks*. These lists are generic and can hold data of any type. The *inthash* class maps integers to data of any kind. Two source files, `structures.cpp` and `structures.h`, compose this module. `structures.h` is also included in the *orderbook* module.

```
C:\>proc 20010125.fav 277
20010125.fav file
Processing …….
379962 messages processed.
Output for firm 277:
377 bids
158 asks
```

Figure 41. *Output of the proc.exe program.*

The *fav* module holds all definitions of the data structures needed in order to deal with financial market data files. Along with the declarations of the data format, the *FavProcessor* class is declared. Two source files, `fav.cpp` and `fav.h`, comprise this module. `fav.h` is also included in the *orderbook* module.

The *orderbook* module is the main one of the system. It declares the `OrderBook` class. The `OrderBook` class uses an `inthash` object, mapping identifiers to structures holding two objects of the class `orderedlist` (figure 42). These lists store the *bids* and *asks* for each stock. The `OrderBook::processMessage()` method is in charge of all operations which involve each message: simple storing (ENTER), modification (AMEND), deletion (DELETE), or modification and deletion (if needed) (TRADE).

The `main()` function (the entry point of the application) is found in the *proc* module, with the `procFavFile()` function, as well. The second one is the real heart of the module, while the first one only takes the arguments from the system and calls `procFavFile()` after checking that those parameters have been really passed to the application. The accepted parameters are a) the name of the data file, and b) the identifier of a stock. The `procFavFile()` function checks the existence of the financial data file, and creates an `OrderBook` object and an `FavProcessor` object for that file. Inside the main loop, it calls the method `getMessage()` of the `FavProcessor` object, and passes the read message to the `processMessage()` method of the `OrderBook` object, until the `FavProcessor` object indicates that the end of the file has been reached. Then, `provFavFile()` outputs the number

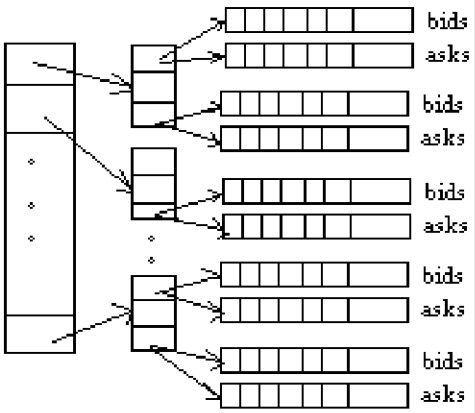

Figure 42. *Structure of an orderbook. A hash function decides in which box the structure for the two lists is placed.*

of *bids* and *asks* for a given stock and returns.

In order to compile the program, the Borland C++ compiler, version 5.2 for C++ Builder was used. The program was executed in a computer running the Windows XP operating system. The figure 41 shows the output of the program.

5.2.3 The adaptation to Barbados C++

```
cd(/);
mkdir(smartstest);
smartstest: directory;
cd(smartstest);
mkdir(structures);
structures: directory;
mkdir(fav);
fav: directory;
mkdir(orderbook);
orderbook: directory;
dir();
structures/     fav/     orderbook/
```

Figure 43. *Structure of the application ported to Barbados.*

As expected, the port of the non-persistent C++ program to Barbados was easy[15] and simple. Note, however, that the obtained program was still a non-persistent one, although running in a persistent system. Barbados supports operations with files, in order to provide interoperability with current (i.e. non-persistent) operating systems. However, by its own nature, a persistent system doesn't need files to operate, as data simply persists due to transparent processing by the system.

As it is a port of the program described in the previous section, its output is the same one.

A one-to-one correspondence with modules and directories was simply achieved: this is shown, along with the commands needed to build the directory structure, in figure 43 and figure 44. Barbados supports directories (García Perez-Schofield *et al.*, 2001b), allowing the programmer to combine the advantages of a file system with a persistent store. These directories, instead of storing files, store objects, which are made persistent transparently by the system.

[15] Although it supposed to improve and debug the compiler of the prototype.

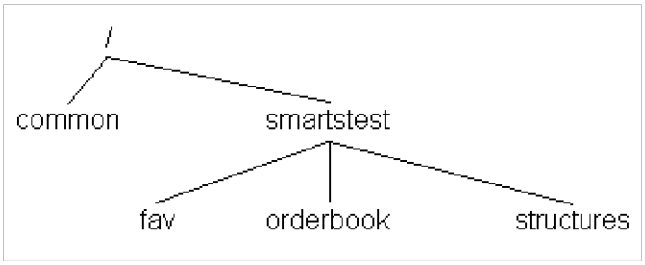

Figure 44. *Tree of containers holding the Smarts test*

The adaptation of all modules to Barbados basically consisted of combining the header files –the ones with extension `.h`- (Barbados doesn't use nor need header files, and it doesn't support the directive `#include`) with the program files (the ones with extension `.cpp`), defining the whole content of the directory.

As Barbados doesn't support header files –nor extern modifiers-, some method must exist in order to permit a module to know about functions or classes in other modules. This communication is allowed, using paths for objects. For example, the `procFavFile()` function in the *orderbook* module/directory, defines an object of the `FavProcessor` class by the line "`/smartstest/fav/FavProcessor favfile;`", or an object of the orderbook class by the line "`/smartstest/orderbook/OrderBook odb;`". Another way to achieve this communication would be to add a line at the beginning of the *orderbook* directory, declaring a reference to the *FavProcessor* directory: "`directory &fav_module = /smartstest/fav;`". Setting that reference, programmers don't have to put a path before each identifier, in a very similar way to the ANSI C++ *namespaces* (Allison, 1998; Stroustrup, 1991) feature.

The simplicity of the process proved us correct saying that the language supported would make back compatibility easy with existing C++ programs and would smooth the learning curve for a programmer interested in Barbados, too.

5.2.4 The persistent version of the performance program

A persistent version of the program described above was developed. The conversion was a matter of writing a still non-persistent *booter* which fills a container (`favdata`) with the data stored in the financial file, in order to 'port the data' from a non-persistent file to a data-structure inside the persistent store. We assume that the starting point for our persistent program is this persistent data-structure (a linked list of messages). The new structure can be seen in figure 45.

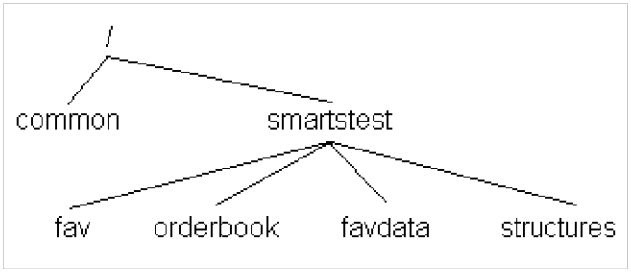

Figure 45. *Structure of the persistent version of the test program*

Then, the program loads the container, and runs over the stored data processing it, identically as in the non-persistent version. The only difference is that the program must run over a linked list, and not over a file. The output of the program is again identical to the first one.

5.3 Performance

Barbados includes an *Intel-native* C++ compiler. This made us suppose that the performance of Barbados would be near to a non-persistent C++ similar environment.

The financial files used were the ones generated during June of 2002, from '20020501.fav' to '20020531.fav' (apart from weekends and other holidays). The mean number of messages is of about 950000, as shown in table 15.

Barbados executing the performance tests achieves quite similar times to the Borland C++ version, being slower in mean time. The results shown in table 15 were obtained.

FAV File	#Messages	Borland	Barbados port	Persistent version
20020501.fav	975616	33	49	37
20020502.fav	1052952	44	68	51
20020503.fav	967357	40	64	55
20020506.fav	1025398	38	65	48
20020507.fav	1102421	42	68	47
20020508.fav	976273	35	58	42
20020509.fav	1037510	39	65	48
20020510.fav	926117	31	51	37
20020513.fav	911948	33	54	39
20020514.fav	910090	31	51	37
20020515.fav	1155062	41	71	55
20020517.fav	932090	34	51	41
20020520.fav	967131	33	53	40
20020521.fav	931071	34	55	40
20020522.fav	915316	34	55	40
20020523.fav	865163	32	52	39
20020524.fav	797369	30	49	36
20020527.fav	845374	30	51	42
20020528.fav	803178	27	44	34
20020529.fav	894999	33	54	39
20020530.fav	823734	30	52	42
20020531.fav	1093351	40	61	47
Mean values	***950432.73***	***34.73***	***56.41***	***42.55***

Table 15. Performance results, obtained through processing 22 financial data files, along the month of June, 2002.

In mean time, the Barbados non-persistent version of the program is a 62% slower than the non-persistent program, and the Barbados persistent version is, again in mean time, only a 22% slower than the non-persistent Borland program. The figure 46 shows a chart comparing the performance results (ordered by the number of messages of each financial file, as shown in table 16).

Sometimes the execution time does not apparently correspond to the size (the number of messages) of the financial file. This comes from the fact that this software doesn't deal with all kind of messages. For example, the FAVID message is used in order to apply a textual identify to a given firm, and it is not managed by the performance software.

Very early versions of Barbados, working with a semi-interpreted language were proven to be

FAV File	Borland	Barbados port	Persistent version
24	30	49	36
28	27	44	34
30	30	52	42
27	30	51	42
23	32	52	39
29	33	54	39
14	31	51	37
13	33	54	39
22	34	55	40
10	31	51	37
21	34	55	40
17	34	51	41
20	33	53	40
3	40	64	55
1	33	49	37
8	35	58	42
6	38	65	48
9	39	65	48
2	44	68	51
31	40	61	47
7	42	68	47
15	41	71	55

Table 16. The same performance results, as shown in the previous table, ordered by number of messages of each financial file.

20 times (2000%) slower than a commercial C++ compiler.

We claim this proves that persistent systems can be as fast or faster as non-persistent ones; perhaps one of the problems is that persistent prototypes have been using always interpreted or semi-interpreted programming languages, which degrades their performance a lot. In fact, this performance is quite encouraging, given that the compiler currently implements very few optimisations.

We therefore defend that Barbados is near enough to commercial products, provided it is still a prototype, and that the purpose of having a system with a performance comparable with other traditional products (García Perez-Schofield *et al.*, 2001c; García Perez-Schofield *et al.*, 2002) has been achieved.

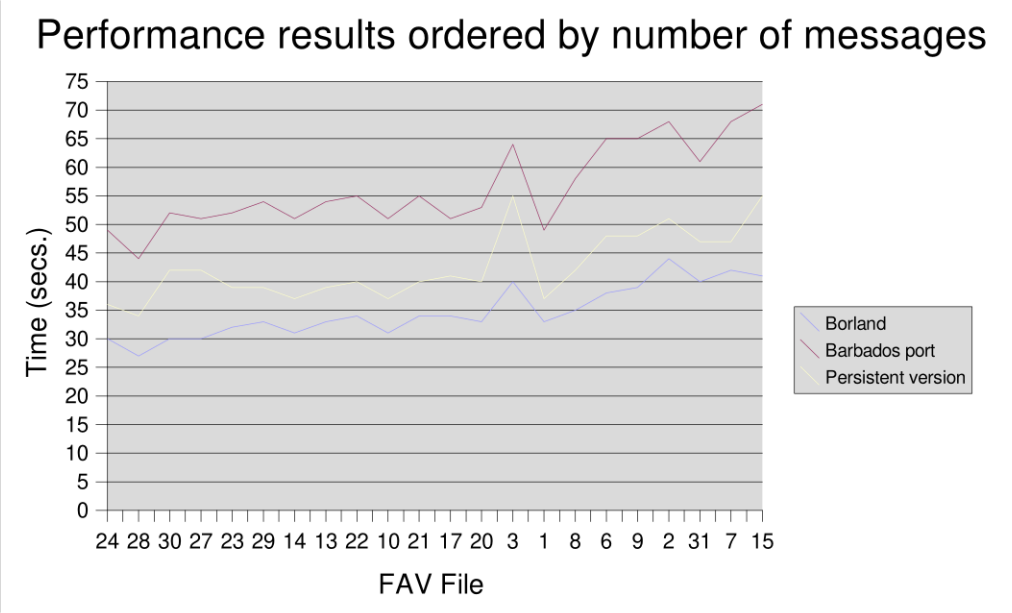

Figure 46. Graphic representation of the performance results, ordered by number of messages.

5.4 Conclusions

The tests enumerated above have shown the way Barbados works, the capabilities of its compiler and its suitability to run easily adapted C++ programs, interoperating with current operating systems. So we find it is very useful, and that the test currently carried out is comparable to other tests for other persistent systems, such as the implementation of Unix in Grasshopper (Bem, *et al.*, 1996).

The performance of Barbados running these tests has demonstrated that the model based on containers is good in terms of performance, and that the *intel-native* compiler makes this persistent system efficient enough, i.e., Barbados has comparable performance with non-persistent programs. We claim that this underlying technology in Barbados, exposed in this PhD thesis, has been proved to be appropriate.

Chapter 6: Final Conclusions and Future Work.

6 Final Conclusions and Future Work

6.1 Final Conclusions and Future Work

In this PhD thesis, the *container-based* model of persistence has been presented in all its detail, along with its implementation issues. This model of persistence has the theoretical flaw of not fulfilling the orthogonal model of persistence.

From the beginning, we considered the orthogonal model as somehow a framework, a start point for research. But instead using it as a rigid framework, we used it as an inspiration for the container-based persistence model. Although the orthogonal model has considerable advantages, such as transparency, we found that it was very rigid and that it came with an important drawback: performance.

The main objective of the design of the container-based model was to simplify the orthogonal model, in order to obtain a more efficient model by its own nature. This was achieved using containers. Containers, instead of being a partition element of the PS, hidden to the user, is now a first-order element in the system which the user can deal with. The idea of providing some

organisation to the persistent store was something which could be found in literature, as has been shown in the "*state of the art"* chapter. But the orthogonal model of persistence is too rigid in that sense in order to admit anything else than recovering or saving a persistent root.

Another idea which can be found in the literature is that an organisation system such as directories in a file system, was something very understandable and was more or less implemented in many systems. The idea of folders classifying data (files or objects) hierarchically is very easy to understand and very intuitive. That's why, over the container, the abstraction of directories was built. So, although the user must specify the container in which he or she is going to work in, this is hidden inside the operation of creating a directory or changing to a directory, which, again, is very intuitive. Also, the user has the possibility of a low-level management of the creation, the closing and the opening of containers through the Barbados C++ API.

But the most important justification behind containers is performance. Performance is one of the big flaws in the research in the field of persistence. Orthogonal persistent systems have a bottle-neck in their implementations which is called the "*swizzle barrier*" (García Perez-Schofield *et al.*, 2001c). Objects are loaded *lazily* in memory, as "*object faults*" happen. The performance of the system depends on the success of the clustering techniques and the efficiency of their swizzling mechanisms and their object caches. That's why they author's main research paths are cache techniques and automatic clustering (orthogonality requires of non-manual methods for clustering).

Of course, containers are not the only cause of success in Barbados' performance, as Barbados incorporates an *intel-native* compiler. This is particularly infrequent in the area of persistence systems. Although the orthogonal model doesn't avoid to use native compilers, they are not employed in practice (PJama and JSpin use Java-native compilers).

Cooper was indicating in his PhD thesis (Cooper, 1997) that the performance of Barbados could be near to any traditional, non-persistent system. In this work we have carried out enough performance tests that proved him true.

Another big issue is schema evolution. Although it is not completely implemented in the prototype, we have presented a complete design which is already available in simple cases. Schema Evolution is based in a mixed *early-lazy* conversion scheme. The PS (i.e., the set of all containers) is not converted immediately when a class changes (which would be very expensive in availability for the system), but simply the system waits until containers with out-to-dated objects are loaded. When this happens, the conversion process is carried out transparently or with the supervision of the user though conversion functions, under user's desire.

Future work consists of completing implementation of schema evolution, and carry out performance tests proving its capabilities. Another important point, not still studied is the implementation of an appropriate resilience mechanism. Although by its own nature, it's guaranteed that the corruption of a container doesn't affect other containers in the PS, the user needs a checkpoint mechanism assuring him or her that in the case of any crash in the system his or her work is going to be preserved. Finally, interoperability with other systems is an interesting characteristic of persistence systems. This is explored in the next section.

6.2 Future work: Interoperability

The Barbados prototype is expected in the future to be able to export the container files to the DLL format (known as PE, Pettriek, 1994), in order to be able to share the code created with the Barbados C++ language.

Note that the format Barbados uses in order to store the different containers is a proprietary, convenient format (in fact, it's very similar to the format of a container in memory (again, in a

similar way to DLL's), which makes easier the restoring and storage processes). The possibility of exporting data has been taken into account in order to increase the interoperability with other programming systems.

DLL's have an important characteristic which make them very suitable for that task: one of the sections of the file (`.reloc`) is expected to store all the *fixups* that must be made in order to have the DLL correctly loaded in memory. These *fixups* are exactly the *swizzled pointers* needed in Barbados.

The use of DLL's as a vehicle for containers, allows a wide range of interconnectivity possibilities with other systems (in contrast with the approach of allowing multiple languages running on the top of the persistent system, as it is exposed in Kaplan, *et al.*, 2000). One of the most important limitations for Persistent Systems is the fact of these systems of being auto-contented, i.e., the only possibility the programmer could take advantage of code done in other systems, is to program a compiler of the language the code is written for in the Persistent System, and then compile it.

Although the possibility of writing a compiler is available, this is nearly impossible in any serious approach to an application created with various tools. In a real project, there is no possibility to write an entire compiler in order to take advantage of code previously developed for another language. An automated mechanism is therefore needed.

Using the DLL format not only for exporting code, but in order to import code into containers too, would allow us to clear the way for the two problems: take advantage of code done in other systems and other languages (with limitations), and export code to other systems and other languages (also with certain limitations). The DLL format (PE, Portable Executable format) is accepted in every Win32 platform, which includes Windows 3.11 © with Win32s, Windows 95 ©, Windows 98 ©, Windows NT ©[16] and Windows 2000/XP ©.

Barbados C++ differs from ANSI C++ in the way it treats Run-Time Type Information (RTTI). In Barbados, classes are first-order entities (there is a 'class *classdef*' to represent them), while in ANSI C++ they aren't. In Barbados, a '*classdef*' object is available to the user, but it is also exactly the same object which the compiler uses and contains all the information available about the class (member functions, friends, visibility etc.) By contrast, the ANSI C++ RTTI (Ellis & Stroustrup, 1990; Stroustrup, 1991) only has information about data field names, offsets and types. While our aim is to implement ANSI C++ quite strictly, this area could introduce a source of incompatibility with existing C++ programs. We have not yet explored how to make the ANSI C++ RTTI feature compatible with Barbados.

The advantages of using the DLL file scheme are twofold:

- The Win32 platform can run on the top of many hardware. This makes possible the immediate communication among other compilers done for the Win32 platform. It includes C, C++, Pascal ... etc
- The DLL file format (PE) is a well-known, well-documented format. It supports different kinds of machines, including Big-Endian and Less-Endian specification ... So it would be feasible to write algorithms loading Barbados code in a DLL and converting it into other systems, including traditional ones.

[16] All these systems are registered trade marks and products of Microsoft Intl.

6.3 Published material on the subject of this PhD thesis

The following material has been already published along the elaboration of this PhD thesis.

1. **Barbados, design of the persistence mechanisms** (García Perez-Schofield *et al.*, 2001a). This is a technical report published in the department of informatics, at the University of Vigo. The architecture of Barbados is deeply discussed here. Language: english.
2. **Extending Containers to Address the Main Problems of Persistent Object-Oriented Operating Systems: Clustering, Memory Protection and Schema Evolution** (García Perez-Schofield *et al.*, 2001b). In this work, a review of the Container-based persistence model is done, focusing the dicussion on solving the main problems of persistence systems. The communication was presented at the *Object Oriented Operating Systems Workshop* held during the *ECOOP'2001*. Language: english.
3. **Performance in Persistent Systems** (García Perez-Schofield *et al.*, 2001c). This communication discusses the main drawbacks in performance of persistent systems, and makes the proposal implemented in the prototype Barbados in order to solve them. This communication was presented at the *Jornadas de Ingeniería del Software y Bases de Datos (JISBD).* Language: spanish.
4. **Swizzling in Container-based systems: Container – Name Swizzling** (García Perez-Schofield *et al.*, 2001d). The container-based model, and its main complexity, swizzling among containers, is discussed in this communication. It was presented at the *IEEE Congreso en Ciencias Computacionales, CICC'01.* Language: spanish.
5. **First impressions about executing real applications in Barbados** (García Perez-Schofield *et al.,* 2002). Again in the ECOOP's workshop, this communication is totally dedicated to the measurement of the prototype. As commented, it was presented at the *Object Oriented Operating Systems Workshop* held during the *ECOOP'2002*. Language: english.
6. **Schema evolution in the Container-based persistence model** (García Perez-Schofield *et al.,* 2002b). This is an article sent to the "*Software, Practice & Experience*" prestigious international journal. The schema evolution mechanism is deeply discussed in the article. It has been accepted, and is pending for publication. Language: english.

Chapter 7: Bibliography

7 Bibliography

Allison, Chuck (1998). "What's new in standard C++?". *C/C++* Users Journal, December 1998, **pp** 69-81.

Álvarez Gutiérrez, D., Tajes Martínez, L., Álvarez García, F., Díaz Fondón, M. A., Izquierdo Castanedo, R., Cueva Lovelle, J.M. (1997). "An Object-Oriented Abstract Machine as the Substrate for an Object-Oriented Operating System". Proceedings of the *1st ECOOP Workshop in Object- Orientation and Operating Systems*, as part of ECOOP' 1997.

Atkinson, M., Jordan, M. (2000). "A Review of the Rationale and Architectures of Pjama: A Durable, Flexible, Evolvable, and Scalable Orthogonaly Persistent Programming Platform". *Sun Microsystems Technical Report SMLI TR-2000-90*, June 2000.

Atkinson M.P., Morrison R. (1986) "Integrated Persistent Programming Systems." *In Proc. 19th International Conference on Systems Sciences, Hawaii (1986) pp 842-854.*

Atkinson, M.P., Morrison, R. (1989). "Napier88 Reference Manual." *Universities of Glasgow and St. Andrews*, Persistent Programming Research Report PPRR-77-89.

Atkinson M.P., Morrison R. (1995). "Orthogonality Persistent Object System", *VLDB Journal* **v**4 **n**3, **pp**319-401, ISSN: 1066-8888

Atkinson, M.P., Chisholm, K.J., Cockshott, W.P. (1982) "PS-Algol: An algol with a Persistent Heap", *ACM SIGPLAN Notices* 1982;**v** 17 **n** 7,**pp** 24-31

Atkinson M.P., Bailey P.J., Chisholm K.J., Cockshott P.W., Morrison R. (1983) "An Approach to Persistent Programming", *The Computer Journal*, **v** 26, **pp** 360-365

Atkinson, M. P., Morrison, R., Pratten, G.D. (1986). *"Designing a Persistent Information Space Architecture"*. Proceedings of 10th IFIP World Congress, Dublin, **pp** 115-120.

Atkinson, M.P., Sjöberg, D.I.K., Morrison. M. (1993). *"Managing Change in Persistent Object Systems"*. Proceedings of JSSST International Symposium on Object Technologies for Advanced Software, Kanazawa, Japan, **pp** 315-338.

Atkinson, M.P., Daynes, L., Jordan, M., Printezis, T., Spence, S. (1996)."An Orthogonally Persistent Java". *ACM SIGMOD Record*, **v** 25, **n** 4, December 1996

Atkinson, M.P., Dmitriev, M., Printezis, T. (2000). "Scalable and Recoverable Implementation of Object Evolution for the PJama Platform". *In proceedings of the 9th Intl. Workshop on Persistent Object Systems*, Lillehammer, Norway, **pp** 255-268.

Balch, P., Cockshott, W.P., Foulk, P.W. (1989). "Layered Implementations of persistent object stores". *Software Engineering Journal*, March 1989.

Bancilhon, F., Delobel, C., Kannellakis, P. (1992). "*Building an Object-Oriented System: the story of O_2*". Morgan Kauffman Publishers Inc. San Mateo CA (1992).

Brahnmath, K., Nystrom, N., Hosking, A., Cutts, Q. (1998). "Swizzle barrier optimizations for orthogonal persistence in Java". *Proceedings of the Third Intl. Workshop on Persistence and Java.*

Bem, E., Lindström, A., Norris, S., Rosenberg J. (1996). "Hoppix - An Implementation of a Unix server on a Persistent Operating System". *Proceedings of the Intl. Workshop on Object-Oriented Operating Systems (IWOOOS'96).*

Benzaken, V., Delobel, C., Harrus, G. (1995). "Clustering Strategies in O_2: an overview". Chapter in "*Building an Object-Oriented System: the story of O_2*".

Biliris, A., Dar, S., Gehani, N.H. (1993). "Making C++ Objects Persistent: The Hidden Pointers." *Software, Practice & Experience*, **v**23 **n**12, **pp** 1285-1303.

Borland, Intl. (1997). "Borland C++ Builder Documentation".

Breche, P., Wörner, M. (1995). "How to remove a class in an object database system". In *Proceedings. of the 2nd International Conference on Application of Databases*. San José, December 13, 1995

Bretl, R., Maier, D., Otis, A., Penney, J., Schuchardr, B., Stein, J., Williams, E.H., Williams, M. (1991). "The Gemstone Data Management System**".** Chapter 12 in Brown A. (1991) ."*Object Oriented Databases*". McGraw-Hill.

Brown, A. W. (1991). "*Object Oriented Databases*". McGraw-Hill. Intl. Series in Software Engineering. ISBN 0-07-707247-2.

Carey, *et al.* (1990). "The EXODUS extensible DBMS project: An overview". *Readings in Object Oriented Databases*, S.Zdonik,and D. Maier, Eds., Morgan-Kaufman, San Mateo, CA.

Carey, M.J., DeWitt, D.J., Naughton, J.F. (1993). "The OO7 Benchmark". *SIGMOD ACM Special Group on Management of Data* **v**22 **n**2**. pp** 12-21.

Carey et al. (1994). "Shoring Up Persistent Applications". In Proceedings of the *ACM_SIGMOD Conference on the Management of Data.*

Cattel, R. (editor) (1993). "*The Object Database Standard: ODMG – 93*", Morgan Kaufman, Menlo Park (Calif.)

Chen J., Huang, Q.M., Sajeev, A.S.M. (1995). "Interoperability between Object-Oriented Programming Languages and Relational Systems". Proceedings of the *TOOLS Conference*, **pp** 52-66. Melbourne, December 1995. Prentice-Hall.

Chennupati, K., Saiedian, H. (1997). "An Evaluation of Object Store Management and Naming Schemes in Persistent Object Systems". *Journal of Object Oriented Programming, October 1997.*

Chih-Ting Du, T., Wolfe, P. M. (1996). "An implementation perspective of applying object-oriented database technologies.". *IIE Transactions* **n**29, **pp**733-742.

Cockshott, W.P. (1993). "Persistent objects in Turbo Pascal for Windows". *Journal of Object Oriented Programming*, **v**6 **n**2, **pp** 68-73, May 1993.

Connor, R.C.H. (1988). *"The Napier Type-Checking Module"* Universities of Glasgow and St. Andrews Report PPRR-58-88.

Cooper R.L., Kirby G.N.C., (1994). *"Type-Safe Linguistic Run-time Reflection - A Practical Perspective"* ESPRIT Basic Research Action, Project Number 6309 - FIDE2, 1994.

Cooper, T.B. (1997). *"Barbados: an Integrated Persistent Programming Environment"*. Ph.D. thesis. Basser Dept of Computer Science, Sidney University.

Cooper, T.B., Wise M., (1995). *"*The Case for Segments*"*, *Proceedings of Intl Workshop on Object Oriented Operating Systems (IWOOS'95).*

Cooper, T.B., Wise M., (1996). "Critique of Orthogonal Persistence", *Proceedings of Intl Workshop on Object Oriented Operating Systems (IWOOS'96)*.

Darmon, J., Formantín, C., Reguier, S., Schneider, M. (2000). "Dynamic clustering in Object Oriented Databases". *Symposium of Objects and Databases*, at ECOOP 2000.

Dearle, A., Rosenberg, J., Henskens, F., Vaughan, F., Maciunas, K. (1992). "An Examination of Operating System Support for Persistent Object Systems". In *Proceedings of the 25th Hawaii International Conferences on System Sciences* **v**1 **pp**779-789.

Dearle, A., di Bona, R., Farrow, J., Henskens, F., Lindström, A., Rosenberg, J., Vaughan, F. (1993). "Grasshopper an Orthogonally Persistent Operating System". GH Technical Report in *Computer Systems, 7,3*. University of Sydney. Available by FTP ftp://ftp.psrg.cs.usyd.edu.au/pub/gh/reports/GH-03.ps.gz

Dearle, A., di Bona, R., Farrow, J., Henskens, F., Hulse, D., Lindström, A., Norris, S., Rosenberg, J., Vaughan, F. (1994). "Protection in the Grashopper Operating System". *Proceedings of the Workshop on Persistent Object Systems*.

Dearle, A., Hulse, D., (1995). "On Page–based Optimistic Process Checkpointing". *Proceedings of the Intl Workshop on Object Oriented Operating Systems*.

Dixon G., Parrington, G., Shrivastava S., Wheater, S.M. (1989). "The Treatment of Persistent Objects in Arjuna". *Proceedings of the European Conference of Object Oriented Programming (ECCOP'89)*.

Dmitriev, M. (1999). "The First Experience of Class Evolution Support in PJama". In *Proceedings of the third Persistence for Java Workshop*.

Dmitriev, M., Atkinson, M. (1999). "Evolutionary Data Conversión in the Pjama Persistent Language". *Proceedings of the Workshop on Object Orientation and Databases, held in ECOOP 1999*.

Ellis, M. A., Stroustrup, B. (1990). "*Annotated Reference Manual*". Addison-Wesley

Ferrandina F., Meyer T., Zicari R.(1994) "Correctness of Lazy Database Updates for an Object Database System". *In Proc. of the 6th Int'l Workshop on Persistent Object Systems*.

Ferrandina, F., Ferran, G. (1995). "Schema and Database Evolution in the O_2 Object Database System". *Proceedings of the 21st Very Large Databases* in Zürich.

Ferrandina, F., Meyer, T., Zicari, R. (1995). "Schema Evolution in Object Databases; Measuring the Performance of Immediate and Deferred Updates". *Proceedings of the Workshop on Object Database Behaviour, Benchmarks and Performance*, held during OOPSLA'95.

Florentín, A. (1998). "Creating Simple Persistent Objects". *C/C++ Users Journal*. August 1998.

García Perez-Schofield, B., Cooper,. T., Roselló, E., Pérez Cota, M. (2001a). "Barbados: Design of the Persistence Mechanisms". *Technical Report in the Department of Informatics, University of Vigo*.

García Perez-Schofield, B., Cooper, T., Roselló, E., Pérez Cota, M. (2001b). "Extending Containers to Address the Main Problems of Persistent Object-Oriented Operating Systems: Clustering, Memory Protection and Schema Evolution.". *Workshop on Object Orientation and Operating Systems,* held during the ECOOP in Budapest, June 2001

García Perez-Schofield, B., Cooper, T., García Roselló E., Pérez Cota, M. (2001c). "Rendimiento en Sistemas Persistentes". *Proceedings of the Jornadas de Ingeniería del Software y Bases de Datos (JISBD), Almagro, España (Spain)*

García Perez-Schofield, B., Cooper, T., García Roselló E., Pérez Cota, M. (2001d). "Swizzling en sistemas basados en containers: Container – Name Swizzling". *Proceedings of the Congreso en Ciencias Computacionales, CICC'01.*

García Perez-Schofield, B., García Roselló E., Cooper, T., Pérez Cota, M. (2002). "First impressions about executing real applications in Barbados". In the proceedings of *Workshop on Object Orientation and Operating Systems,* held during the ECOOP in Málaga, España (Spain), June 2002.

García Perez-Schofield, B., García Roselló E., Cooper, T., Pérez Cota, M. (2002b). "Managing schema evolution in a container-based persistent system". *Software, Practice & Experience* (to be published).

Jordan, M. (1996). "Early Experiences with Persistent Java". In *Proceedings of the 1st Intl. Workshop on Persistence and Java.*

Jordan M., Atkinson M., (1998). "Orthogonal Persistence for Java - A mid-term Report". In proceedings of *The Third International Workshop on Persistence and Java(tm)* (PJW3) 1998.

Kaplan, A., Myrestrand, G. Ridgway, J. V. E, Wileden, J.C. (1996). "Our Spin on Persistent Java: The JavaSPIN Approach". In proceedings of the *First International Workshop on Persistence and Java(tm)* (PJW3) .

Kaplan A., Ridgway, J.H.G., Schmerl, B.R. (2000). "Toward Polylingual Persistence". In Proceedings of the *Ninth International Workshop on Persistent Object Systems*. **pp** 202-217

Kim, W., Ballou, N., Chou, H., Garza, J., Eowlk, D. (1991). "Features of the ORION Object Oriented Database System". Chapter 11 in Brown, A. (1991) "*Object Oriented Databases*" . McGraw-Hill.

Kirby, G. N.C. (1992). *"Persistent Programming with Strongly Typed Linguistic Reflection"*, Proc. 25th Intenational Conference on System Sciences, Hawai 1992, **pp** 820,831.

Kirby, G. N.C. (1993). *"Reflection and Hyper-Programming in Persistent Programming Systems"*, PhD Thesis - University of St. Andrews, 1993.

Kirby, G., Morrison, R. (1998a). "A Persistent View of Encapsulation". Chapter in *Computer Science '98*, a book edited by Springer-Verlag.

Kirby, G., Morrison, R., (1998b). "Linguistic Reflection in Java". *Software, Practice and Experience*, **v**28 **n**10, **pp**1045-1077.

Knasmüller, M. (1996). "Adding persistence to the Oberon-System". *Technical Report no. 6*, Institute for Computer Science, Johannes Kepler University.

Knasmüller, M. (1997). "Adding schema evolution to the Persistent Development Environment Oberon-D". *Technical Report no. 10*, Institute for Computer Science, Johannes Kepler University.

Korsholm, S. (1999). "Transparent, Scalable, Efficient OO-Persistence". *Proceedings of European Conference of Object Oriented Programmin*g, Lisbon, June 1999.

Lindström, A., Dearle, A., di Bona, R., Farrow, J.M., Henskens, F., Rosenberg, J., Vaughan F. (1994). "A Model for User-Level Memory Management in a Distributed, Persistent Environment". *Proceedings of the Seventeenth Annual Computer Science Conference ACSC-17* Part-B. Ed. Gepal Gupta, Chirschtchurch, New Zealand, pp 345-354

Lindström, A., Rosenberg, J., Dearle, A. (1995). "The Grand Unified Theory of Address Spaces". *Fifth Workshop on Hot Topics in Operating Systems*. May 4-5, 1995.

Loomis, M. E. S. (1993). "Making objects persistent". *Journal of Object Oriented Programming*, **v**6, **n**6, **pp**25-28.

Marquez, A., Blackburn, S.M., Mercer G., Zigman, J. (2000). "Implementing Orthogonally Persistent Java". In *Proceedings of the Ninth Intl. Workshop on Persistent Object Systems*, pp 218-232.

Malenfant J. , Demers M.J., Demers F.N. (1996). "A Tutorial on Behavioral Reflection and its Implementation". *Proceedings of Reflection'96*, San Francisco, USA, April 21-23.

Metrowerks Inc. (1999). "*Code Warrior Pro 5*."

Microsoft Corp. (1998a). "*Microsoft® Win32® API Programming*". Microsoft Press.

Microsoft Corp. (1998b). "*Microsoft ® Visual Basic ® 6.0 Programmers Guide*". Microsoft Press. ISBN 1-57231-863-5.

Moss, J. E. B. (1992). "Working with Persistent Pointers: To Swizzle or not to Swizzle". IEEE Transactions on Software Engineering. **v**18 **n**8, **pp**657-573.

Morrison, R., Brown, A.L, Connor, R.C.H et al. (1990). "Protection in Persistent Object Systems". *Proceedings of Intl. Workshop on Security and Persistence, Bremen. Springer-Verlag*, **pp** 48-66

Morrison, R., Connor, R.C.H., Cuts, Q.L., Kirby, G.N.C, Munro, D.S., Atkinson, M.P. (1999). *"*The Napier88 Persistent Programming Language and Environment.*"*. To appear in: *The FIDE Book*.

Nrasayva, Y. Ng, Tze Sing. (1996). "Reducing the Virtual Memory Overhead of Swizzling". *Proceedings of IWOOS (Intl. Workshop on Object Oriented Operating Systems) 1996*.

Object Management Group (1996). "The Common Object Request Broker: Architecture and Specification", July 1996. (http://www.odmg.org)

Ortín Soler, F., Álvarez Gutiérrez, D., Izquierdo Castanedo, R., Martínez Prieto, A., Cueva Lovelle, J. M. (1997). "The Oviedo3 persistence system". In proceedings of the *III Jornadas de Orientación a Objetos*. Sevilla, España (Spain).

Ortín Soler, F. (2001). "*Sistema Computacional de Programación Flexible Diseñado sobre una Máquina Abstracta Reflectiva no Restrictiva*" PhD thesis, Departamento de Informática, University of Oviedo.

Pettriek, M. (1994). "Peering Inside the PE: A Tour of the Win32 Portable Executable File Format". *Microsoft Systems Journal*, March 1994

Richardson, J.E., Carey, M.J., Schuh, D.T. (1993). "The design of the E programming language", *ACM Transactions of Programming Languages and Systems*, **v**15 **n**3.

Ridgway, J., Wielden, J.C. (1998). "Toward Class Evolution in Persistent Java". In *proceedings of Third Persistence and Java Workshop*.

Rosenberg, J., Dearle, A., Hulse, D., Lindström, A., Norris, S. (1996). "Operating System Support for Persistent and Recoverable Computations". *Communications of the ACM* **v**39 **n**9, **pp**62-69, September 1996.

Roselló E.G., Ayude J., García Perez-Schofield B., Pérez M. (2001). "Towards orthogonal concurrency principles in object-oriented systems design". *Journal of Object Technology (JOT).*

Shapiro, J.S, Farber, D.J., Smith, J.M. (1996). "State Caching in the Eros Kernel". *In proceedings of the Seventh Workshop on Persistent Object Systems*.

Shapiro, J.S., Smith, J.S., Farber, D.J. (1999). "EROS: A fast capability system". *Operating Systems Review*, **34**(5): 170-185, December 1999.

Shapiro, M. (1988) *"Persistence and Migration for C++ Objects"*, Proceedings of the European Conference on Object Oriented Programming 1988 (ECOOP88).

Shapiro M. *et al.* (1989) "SOS: An Object-Oriented Operating System - Assessment and Perspectives", *Computing Systems* **v** 2 **n** 4 **pp** 287-337.

Shapiro, M., Kloosterman, S., Riccardi, F. (1997). "PerDis. A Persistent Distributed Store for Persistent Applications". 3rd Capernet Plenary Workshop

Shapiro, M., Ferreira, P., Richer, N. (2000). "Experience with the PerDis large-scale data-sharing middleware." *In proceedings of the Workshop on Persistent Object Systems.*

Shilling J.J. (1994). "How to roll your own persistent objects in C++". *Journal of Object Oriented Programming* **v**7 **n**4 **pp** 25-32.

Singhal, V., Sheetal K., Wilson, P. (1992). "Texas: A Portable, Efficient Persistent Store". In Persistent Object Systems: Proceedings of the Fifth Intl Workshop on Persistent Object Systems, **pp** 11-13.

Sousa, P., Alves Marques, J. (1994). "Object Clustering in Persistent and Distributed Systems". *Proceedings of the 6th Workshop on Persistent Object Systems*, **pp** 402-414.

Srinavasan V., Chang, D.T., (1997). "Object persistence in object-oriented applications". *IBM Journal*, **v**36 **n**1.

Stroustrup, Bjarne. (1991). "*The C++ Programming Language*". Addison-Wesley. ISBN 0-201-53992-6

Tan L., Katayama T. (1989). "Meta operations for type management in object-oriented databases - a lazy mechanism for schema evolution." *Proceedings of the First International Conference on Deductive and Object-Oriented Databases*, DOOD `89., Kyoto, Japan. W. Kim, J.-M. Nicolas and S. Nishio (eds.). North-Holland. 241-258.

Tsangaris, M., Naughton, J. (1992). "On the Performance of Object Clustering Tecniques". In *Proceedings of the ACM Sigmod Intl. Conference of Management of Data*, in San Diego, USA, June 2-5

Vadaparty, K. (1999a). "Bridging the Gap between Objects and Tables: Object and Data Models". *Journal of Object Oriented Programming*, v12 n2, pp60-64.

Vadaparty, K. (1999b). "Mapping Objects to Tables". *Journal of Object Oriented Programming*, v12 n4, pp45-47.

Vaughan, F., Dearle, A. (1992). "Supporting large persistent stores using conventional hardware". *Proceedings of the Fifth Intl. Workshop on Persistent Object Systems*, San Miniato, Italy, September 1992.

Vemulapati, M., Sriram, R.D., Gupta A. (1995). "Incremental Loading in the persistent C++ language E". *Journal of Object Oriented Programming* v8 n4, pp34-42., July-August 1995

Zirintsis, E., Kirsby, G., Morrison, R. (2000). "Hyper-Code Revisited: Unifying Program Source, Executable and Data." In *Proceedings of the Ninth Intl. Workshop on Persistent Object Systems.* **pp** 202-217.