Bases de datos ODMG/C++ orientadas a objetos

Cuando se utiliza una base de datos orientada a objetos bajo C++, se emplea un mismo modelo de objetos tanto en la base de datos como en las aplicaciones.

La declaración de clases de la aplicación C++ sirve como esquema de definición de la base de datos, permitiendo a los desarrolladores el empleo de un lenguaje y un esquema de diseño familiares.

Los desarrolladores disfrutan de las siguientes ventajas:

- No necesitan definir el diseño de la aplicación en C++ y además en un lenguaje de base de datos.
- No necesitan traducir la estructura de la información entre el modelo utilizado en la aplicación y el utilizado por la base de datos.
- Comparte un sistema único y extensible entre la aplicación y la base de datos. Este modelo compartido es la característica primaria de una base de datos orientada a objetos y la causa de su utilidad.

En este capítulo analizaremos las siguientes operaciones:

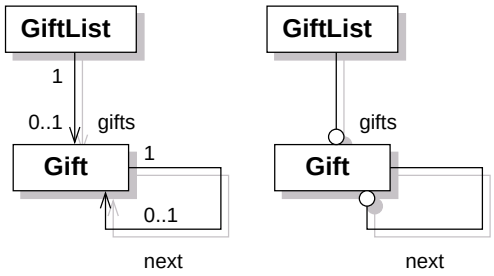
- Definir el esquema de la base de datos.
- Crear nuevas instancias de objeto en la base de datos.
- Asociar un nombre con una instancia y acceder a la instancia mediante dicho nombre.
- Establecer y navegar a través de las relaciones entre instancias.
- Modificar y borrar una instancia.

Ejemplo de una aplicación

En este ejemplo desarrollaremos una aplicación simple para gestionar una lista de los regalos que han de recibir un conjunto de individuos. El modelo orientado a objetos consiste en dos clases:

- La clase `Gift`.
- La clase `GiftList` que representa al conjunto de personas y los regalos que recibirán. La clase `GiftList` mantiene una lista enlazada simple de objetos `Gift`.

En la figura se representan estas clases y sus relaciones.



Ejemplo de una aplicación ...

Las clases se declaran en un fichero *header* C++ (supongamos que nombrado `giftlist.hpp`) cuyo listado sería:

```
class Gift {
  friend class GiftList;
  friend ostream& operator<<(ostream &, const GiftList &);

public:
  Gift(const char *fn, const char *ln, const char *gift,
       unsigned long c);
  Gift();
  friend ostream& operator<<(ostream &, const Gift &);
  friend istream& operator>>(istream &, Gift &);

private:
  char first_name[16];
  char last_name[16];
  char gift_descr[30];
  unsigned long cost;
  Gift *next;
};

class GiftList {
public:
  GiftList(const char *n, unsigned long b);
  ~GiftList();
  friend ostream& operator<<(ostream &, const GiftList &);
  void addGifts(istream &);
private:
  char name[20];
  unsigned long budget;
  Gift *gifts;
};
```


Ejemplo de una aplicación ...

```
#include <giftlist.hpp>

GiftList::GiftList(const char *nm, unsigned long b)
  : budget(b)
{
  strncpy(name, nm, 19);
  name[19] = 0;
}

Gift::Gift(const char *fn, const char *ln,
           const char *gift, unsigned long c)
  : cost(c)
{
  strncpy(first_name, fn, 15); first_name[15] = 0;
  strncpy(last_name, ln, 15); last_name[15] = 0;
  strncpy(gift_descr, gift, 29); gift_descr[29] = 0;
}

Gift::Gift()
  : cost(0)
{
  first_name[0] = 0;
  last_name[0] = 0;
  gift_descr[0] = 0;
}
```

Ejemplo de una aplicación ...

Supongamos ahora que decidimos que las instancias de esas clases deben almacenarse en una base de datos orientada a objetos:

- Necesitaremos realizar algunos cambios sobre las clases para conferirles persistencia.
- En este contexto, persistencia significa que el objeto se almacena en la base de datos y de este modo sobrevive a la ejecución de la aplicación.
- Las instancias de una clase con capacidad de persistencia pueden almacenarse en una base de datos.

En primer lugar, la aplicación debe crear una base de datos.

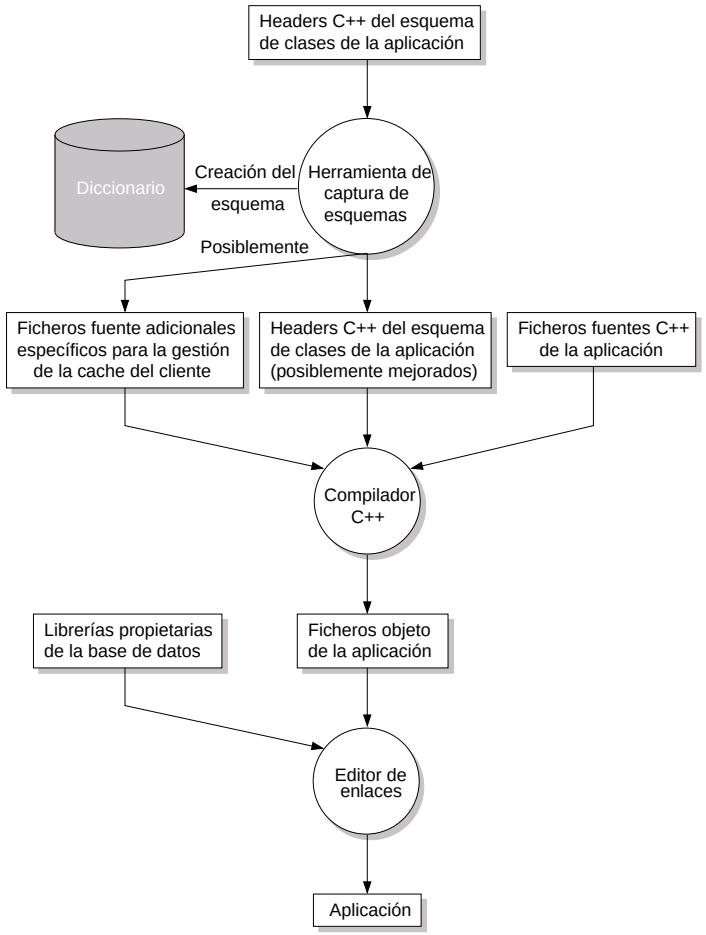
- Los desarrolladores de DBMSs proporcionan un comando o una interface gráfica con este objetivo.
- El estándar ODMG no establece ningún dictado a este respecto, dejando que el vendedor suministre el procedimiento que estime conveniente.
- Supongamos que hemos creado una base de datos denominada *Gifts* utilizando una de las herramientas suministradas por el vendedor.

En siguiente paso consiste en definir un esquema.

- En una base de datos C++, el esquema deriva directamente de las clases declaradas en la aplicación C++.
- Una utilidad, denominada *Schema Capture Tool* lee las cabeceras (*headers*) de la aplicación y genera el esquema correspondiente.
- El *Schema Capture Tool* puede generar código adicional que también deberá enlazarse (*link*) en la aplicación.

Ejemplo de una aplicación ...

En la figura se describen los procedimientos y componentes implicados en la compilación de una aplicación de base de datos orientada a objetos.



Ejemplo de una aplicación ...

Es necesario realizar algunos cambios a las clases C++ antes de poder ejecutar la herramienta de captura del esquema.

1. En ODMG, una clase es persistente si deriva de la clase base `d_Object`. Las clases `Gift` y `GiftList` deben, por tanto, derivar de la clase `d_Object`.
2. Un puntero puede contener la dirección en memoria de un objeto, pero esa dirección carece de significado para cualquier otro proceso. Por tanto, no basta con almacenar el valor del puntero en la base de datos y esperar que puede ser reutilizado por otra aplicación. ODMG define una clase puntero inteligente (*smart pointer class*) denominada `d_Ref<T>` para hacer referencia a un objeto de la clase `T`. Las instancias de esta clase pueden almacenarse en la base de datos y utilizarse para hacer referencia a objetos de ésta. Es requisito indispensable que la clase `T` derive de la clase base `d_Object`. Para almacenar instancias de nuestras clases `Gift` y `GiftList` en la base de datos es necesario cambiar los punteros al tipo `d_Ref<T>`.

Estos dos cambios son los únicos requeridos en las definiciones de las clases `Gift` y `GiftList` para convertirlas en persistentes.

Ejemplo de una aplicación ...

El *header* modificado quedaría:

```
#include <odmg.h>

class Gift : public d_Object {
  friend class GiftList;
  friend ostream& operator<<(ostream &, const GiftList &);

public:
  Gift(const char *fn, const char *ln, const char *gift,
       unsigned long c);
  Gift();
  friend ostream& operator<<(ostream &, const Gift &);
  friend istream& operator>>(istream &, Gift &);

private:
  char first_name[16];
  char last_name[16];
  char gift_descr[30];
  unsigned long cost;
  d_Ref<Gift> next;
};

class GiftList : public d_Object {
public:
  GiftList(const char *n, unsigned long b);
  ~GiftList();
  friend ostream& operator<<(ostream &, const GiftList &);
  void addGifts(istream &);
private:
  char name[20];
  unsigned long budget;
  d_Ref<Gift> gifts;
};
```

Ejemplo de una aplicación ...

Una vez que hemos ejecutado la utilidad de captura y definido un esquema, cualquier aplicación puede insertar objetos en la base de datos.

- Toda operación de interacción con una base de datos debe enmarcarse en una *transacción*, que se representa como una instancia de la clase `d_Transaction`.
- Utilizamos la clase `d_Database` para abrir la base de datos.

En la figura se muestra el código de una aplicación que crea una instancia persistente de `GiftList` y le asigna un nombre.

```
#include <odmg.h>

d_Database db;
const char * const db_name = "Gifts";

int main(int argc, char *argv[])
{
  db.open(db_name);                    // Abre la base de datos
  d_Transaction tx;
  tx.begin();                          // Inicia una transaccion

  char *name = argv[1];
  unsigned long budget = atoi(argv[2]);

  GiftList *giftlist = new(&db,"GiftList")
    GiftList(name, budget);
  db.set_object_name(giftlist, name);

  tx.commit();                         // Valida la transaccion
  db.close();                          // Cierra la base de datos

  return 0;
}
```


Ejemplo de una aplicación ...

- Declaramos una instancia global de la clase `d_Database` con el nombre `db` para que sea accesible por todas las funciones de la aplicación.
- El operador `new` invocado ha sido redefinido en la clase `d_Object` para que devuelva un puntero a una instancia de `d_Database`. Esta llamada crea una instancia de `GiftList` permanente en la base de datos asociada a la variable `db`.
- La siguiente línea utiliza `db` para asociar un nombre con la instancia `GiftList`.
- Sólo es necesario añadir dos líneas para crear y dar nombre a la instancia.
- La aplicación no requiere el uso de un lenguaje de bases de datos embebido como SQL.
- Tampoco es necesario traducir el objeto entre la representación de la aplicación y de la base de datos.

Ejemplo de una aplicación ...

Escribamos ahora una aplicación que acceda a la instancia `GiftList` por su nombre y la imprima en la salida estándar (*standard output stream*). El código resultante es:

```
#include <odmg.h>

d_Database db;
const char * const db_name = "Gifts";

int main(int argc, char *argv[])
{
  db.open(db_name);                    // Abre la base de datos
  d_Transaction tx;
  tx.begin();                          // Inicia la transaccion

  char *name = argv[1];
  d_Ref<GiftList>giftlist = db.lookup_object(name);
  if ( giftlist.is_null() ) {
    cerr << "No existe la lista de regalos con el nombre";
    cerr << name << endl;
    return -1;
  }
  cout << *giftlist;

  tx.commit();                         // Valida la transaccion
  db.close();                          // Cierra la base de datos
  return 0;
}
```


Ejemplo de una aplicación ...

- La función `lookup_object` miembro de la clase `d_Database` devuelve una referencia a la instancia nombrada. Si el nombre no se corresponde con algún objeto de la base de datos, la función devuelve una referencia nula.
- La variable `giftlist` del tipo `d_Ref<GiftList>` se declara e inicializa con el valor de retorno de `lookup_object`.
- La función `is_null` es del tipo `d_Boolean`, definido por el ODMG y que puede tomar dos valores `d_False` y `d_True` (cero y no-cero respectivamente). La función `is_null` devuelve *true* si `giftlist` tiene un valor nulo y *false* si hace referencia a una instancia de `GiftList`.
- El siguiente bloque imprime el objeto en la salida estándar. El operador `<<` requiere una referencia a un objeto de la clase `GiftList`, por lo que aplicamos a `giftlist` el operador de derreferencia (`*`) de `d_Ref<GiftList>`.
- Este operador de derreferencia provoca que el objeto sea accedido automáticamente desde la base de datos y sea instanciado en el espacio de direcciones de la aplicación.
- La aplicación no requiere software adicional para proyectar el objeto desde la base de datos a su representación C++.

Ejemplo de una aplicación ...

La clase `GiftList` cuenta con una función miembro denominada `addGifts`.

- Esta función lee un regalo por línea desde la entrada estándar hasta que se alcanza un `EOF`.
- Cada línea contiene la información necesaria para inicializar una instancia `Gift`.

El código de la función `addGifts` es:

```
void GiftList::addGifts(istream &is)
{
  Gift *gift;
  mark_modified();
  while ( is.peek() != EOF ) {
    gift = new(&db, "Gift") Gift();      // Crea un Gift is ¿¿ *gift;
// Lee de is gift-¿next = gifts; gifts = gift;
```

- Esta función modifica un objeto `GiftList`: es necesario realizar previamente una llamada a `mark_modified`.
- Las asignaciones a los miembros `gifts` de `GiftList` y `next` de `Gift` establece una relación entre las instancias en la base de datos.
- La función `addGifts` puede mejorarse permitiendo que rechace la adición de un nuevo regalo si se sobrepasa el presupuesto asignado (`budget`).

Ejemplo de una aplicación ...

Modificaremos ahora el código anterior para que, en lugar de imprimir el objeto `GiftList` invoque a la función `addGifts`. El listado correspondiente es:

```
#include <odmg.h>

d_Database db;
const char * const db_name = "Gifts";

int main(int argc, char *argv[])
{
  db.open(db_name);                          // Abre la base de datos
  d_Transaction tx;
  tx.begin();                                // Inicia la transaccion

  char *name = argv[1];
  d_Ref<GiftList>giftlist = db.lookup_object(name);
  if ( giftlist.is_null() ) {
    cerr << "No_existe_la_lista_de_regalos_con_el_nombre_";
    cerr << name << endl;
    return -1;
  }
  giftlist->addGifts(cin);

  tx.commit();                               // Valida la transaccion
  db.close();                                // Cierra la base de datos
  return 0;
}
```

El operador `->` está definido en la clase `d_Ref<GiftList>` y devuelve un puntero a la instancia `GiftList` correspondiente.

Ejemplo de una aplicación ...

Para verificar que las instancias de la clase `Gift` se han añadido correctamente al objeto `GiftList`, podemos imprimir este objeto haciendo uso del operador `<<`. Dicho operador es reemplazado (*overloaded*) en la clase `GiftList`.

```
ostream& operator<<(ostream &os, const GiftList &gl)
{
  d_Ref<Gift> gift;
  os << gl.name << " " << gl.budget << endl;
  for (gift = gl.gifts; !gifts.is_null(); gift = gift->next)
    os << *gift;
  return os;
}
```

El bucle `for` recorre las instancias `Gift` asociadas a la instancia `GiftList`. El recorrido entre instancias mediante una referencia a objetos como `d_Ref<T>` es el mecanismo más elemental para acceder a objetos relacionados.

Siguiendo con el ejemplo, necesitamos también borrar una instancia `GiftList` y todas sus instancias `Gift` asociadas de la base de datos.

```
GiftList::~GiftList()
{
  d_Ref<Gift> gift, giftn;

  for (gift = gifts; !gift.is_null(); gift = giftn) {
    giftn = gift->next;
    delete gift.ptr();
  }
}
```


Ejemplo de una aplicación ...

Mediante este ejemplo, hemos mostrado como utilizar referencias a objetos para crear instancias en la aplicación. Una aplicación mayor puede contener muchas referencias a la misma instancia.

- El objeto se crea en la aplicación cuando se produce la primera referencia y la base de datos mantiene la información acerca de qué objeto está siendo accedido por la aplicación. Por tanto, cualquier llamada subsecuente para instanciar este mismo objeto devolverá un puntero al área de memoria de la aplicación que lo contiene.
- Este proceso permite acceder de forma eficiente a los datos, ya que no es necesario volver a acceder a la información contenida en almacenamiento secundario.
- Esta capacidad proporciona una gran ventaja a los DBMSs orientados a objetos en cuando a rendimiento y velocidad de acceso.

Las bases de datos orientadas a objetos se caracterizan por su *transparencia*:

- Puesto que el mismo paradigma es compartido por la base de datos y la aplicación, los objetos pueden proyectarse de un lado al otro de forma no intrusiva.
- La base de datos es mucho menos visible a la aplicación que cuando se emplean otras tecnologías de bases de datos.
- Una consecuencia directa de estos aspectos es la importante reducción en la complejidad y en los costes de desarrollo.

Ejemplo de una aplicación ...

Conclusiones:

- El modelo de objetos de C++ sirve no sólo como esquema sino también como interface con la base de datos. A medida que la aplicación itera sobre una colección de objetos derrefenciados se requiere menor intervención del ODBMS y la aplicación "navega" a través de objetos residentes en memoria. Por supuesto, este proceso de navegación está dando lugar a accesos sobre la base de datos si los objetos no están todavía en la memoria del programa, pero esta actividad permanece oculta a la aplicación.
- El software de la base de datos fuerza la transparencia proporcionando una *cache* de objetos para la aplicación. Se trata de una región en la memoria de la aplicación que contiene a los objetos de la base de datos que han sido accedidos por la aplicación. Cuando una aplicación pretende modificar un objeto debe invocar a la función `mark_changed` de la clase `d_Object`.
- La aplicación no tiene que propagar sus modificaciones a la base de datos. Cuando se completa la transacción (*commit*), el ODBMS escribe en la base de datos automáticamente aquellos objetos que lo requieran. Este mecanismo reduce significativamente los costes de desarrollo ya que es posible desarrollar y utilizar librerías que modifican objetos sin necesidad de introducir mecanismos para coordinarse con las aplicaciones que utilizan la librería.

Bases de datos, transacciones y gestión de excepciones

Acceso a la base de datos

- El estándar ODMG proporciona un mecanismo para acceder sólo a una base de datos. La implementación permite el acceso a varias bases de datos, pero el estándar no lo requiere.
- Para que una aplicación pueda acceder a los objetos de una base de datos primero debe "abrir" la base de datos.
- La clase ODMG `d_Database` proporciona la función `open` que es invocada con el nombre de la base de datos que se desea abrir.
- La enumeración `access_status` puede suministrarse opcionalmente para indicar el modo de acceso.

Supongamos que una aplicación mantiene información acerca de los sueldos en una base de datos corporativa denominada "Personal". El siguiente código abriría la base de datos:

```
d_Database db;
db.open("Personal");
```

- El `access_status` por omisión es `d_Database::read_write`.
- Para acceder a la base de datos en exclusiva, se especifica el acceso `d_Database::exclusive`:

```
d_Database db;
db.open("Personal", d_Database::exclusive);
```

Es necesario abrir la base de datos antes de iniciar cualquier transacción. Una vez que la aplicación a terminado las transacciones sobre la base de datos, debe cerrarla mediante una llamada a la función `close`.

Transacciones

La clase ODMG `d_Transaction` proporciona el conjunto típico de operaciones de una transacción y sirven para agrupar una serie de operaciones iniciadas por la aplicación.

Todas las operaciones que se llevan a cabo sobre objetos permanentes deben realizarse dentro de una transacción.

Una transacción se inicia mediante una llamada a `begin`:

```
d_Transaction tx;
tx.begin();
//
// Lleva a cabo las operaciones sobre
// los objetos de la base de datos
//
tx.commit();
```

Una transacción puede finalizarse de varias formas:

- Mediante una la llamada a la función `commit` que provoca que todas las operaciones de creación, modificación y borrado de objetos permanentes que se hayan realizado durante la transacción sean validados en la base de datos. Además, todos los objetos se eliminan de la *cache* de la aplicación y se levantan los *locks* adquiridos por la transacción.
- Llamando a la función `checkpoint`. Los datos se validan en la base de datos pero la aplicación retiene todos los objetos y los *locks*.
- Mediante la llamada a `abort`. Se eliminan todos los *locks* de la transacción pero no se aplica ningún cambio sobre la base de datos. Si una transacción está activa y se invoca al destructor de instancia `d_Transaction`, el proceso se aborta. Esto implica que si la instancia `d_Transaction` es una variable local y se alcanza un `return` antes del `commit`, la transacción se aborta.

Excepciones: gestión de fallos

El estándar ODMG utiliza el mecanismo de gestión de excepciones de C++:

```
#include <odmg.h>

d_database db;

main(int argc, char *argv[])
{
  int ret;
  d_Transaction tx;

  try {
    db.open("Personal");
    tx.begin();
    //
    // Lleva a cabo la transaccion
    //
    tx.commit();
    db.close();
  }

  catch(d_Error &err) {
    cerr << "DB_Error_" << err.get_kind() << "_";
    cerr << err.what() << endl;
    ret = -1;
  }

  catch(exception &e) {
    cerr << "Excepcion_" << e.what() << endl;
    ret = -2;
  }

  catch(...) {
    cerr << "Excepcion_desconocida\n";
  }

  return ret;
}
```

Dominios

- Los dominios son los tipos de datos más primitivos que se pueden almacenar en una base de datos.
- Los objetos y sus relaciones normalmente constan de un conjunto de atributos y los dominios son los tipos de estos atributos. Los atributos son instancias de un tipo de dominio dado.
- Los dominios no son persistentes. Una instancia de un dominio puede convertirse en persistente cuando forma parte de un objeto permanente.

Categorías de dominio

Los dominios se clasifican en categorías basadas en el grado de agregación y encapsulado:

- Tipos literales primitivos.
- Tipos compuestos.
- Tipos de datos abstractos.

Todos los dominios tienen una implementación que es un agregado de componentes.

Por ejemplo, un tipo que representa una fecha se considera atómico aun cuando contiene varios componentes (año, mes y día). De modo similar, una cadena se considera un tipo de dominio aunque se represente mediante un vector de caracteres.

Tipos literales primitivos

El lenguaje C++ proporciona un conjunto de tipos literales primitivos entre los que se encuentran los siguientes:

- `char` (con y sin signo).
- `short` (con y sin signo).
- `int` (con y sin signo).
- `long` (con y sin signo).
- `float` y `double`.

El tipo `puntero`, utilizado para contener la dirección de memoria de una instancia de un tipo particular, no es soportado por el ODMG estándar aunque si por algunos vendedores.

Tipos primitivos en ODMG

ODMG da soporte a un subconjunto de los tipos primitivos de C++:

Tipo	Longitud	Descripción
`d_Char`	8 bits	Carácter ASCII.
`d_Octet`	8 bits	Sin interpretación.
`d_Boolean`	Indefinida	`d_False` y `d_True`.
`d_Short`	16 bits	Entero `Short` con signo.
`d_UShort`	16 bits	Entero `Short` sin signo.
`d_Long`	32 bits	Entero con signo.
`d_ULong`	32 bits	Entero sin signo.
`d_Float`	32 bits	Número de punto flotante en simple precisión (Estándar IEEE 754-1985).
`d_Double`	64 bits	Número de punto flotante en doble precisión (Estándar IEEE 754-1985).

Tipos primitivos en ODMG ...

- Los tipos `int` y `unsigned int` no se incluyen en el ODMG ya que su tamaño depende de la longitud de palabra de máquina. En entornos de varias arquitecturas un `int` no preserva su valor.
- Los tipos primitivos del ODMG tienen su origen en la interface de definición del lenguaje (IDL) de la arquitectura CORBA.
- Algunas implementaciones del ODMG requieren que se utilicen estos tipos en lugar de los C++.

Tipos compuestos

Un tipo compuesto consta de varios miembros accesibles directamente. (`struct` o `class` con miembros públicos).

```
struct Address {
  char    street[40];
  char    city[20];
  char    state[3];
  char    zip_code[6];
};

struct Point {
  int     x;
  int     y;
};

class Rectangle {
  public:
  Point       origin;
  int         width;
  int         height;
  // Operaciones...
};
```

- La clase `Rectangle` un miembro de tipo `Point` que es a su vez un tipo compuesto.
- La clase `Rectangle` se considera un tipo compuesto debido a que sus miembros son públicos.

Tipos de datos abstractos

- Un tipo de datos abstracto representa una abstracción que se implementa mediante una clase.
- El término *abstracto* significa, en este contexto, un tipo de datos definido por el usuario que tiene sentido en el dominio del problema.
- El ADT (*Abstract Data Type*) proporciona las operaciones que mantienen la integridad semántica y la consistencia interna de las instancias.
- Un tipo de datos abstracto se distingue de un tipo compuesto debido a que el mecanismo de encapsulación proporciona una interface.
- La implementación puede ser una agregación compleja de componentes, pero no es accesible.

Tipos de dominio ODMG

El ODMG define un conjunto de clases para representar abstracciones comunes. Estas clases soportan las formas canónicas C++, que incluyen una función de inicialización por omisión (constructor nulo), operador de asignación y destructor.

El ODMG define una clase cadena de caracteres (*string*) denominada `d_String` que soporta sólo el conjunto mínimo de operaciones necesario para almacenar cadenas de caracteres en una base de datos. Es decir, no está desarrollada para realizar complejas operaciones de manipulación de cadenas de caracteres.

El ODMG también soporta las siguientes clases para representar fechas y tiempos:

- `d_Interval`
- `d_Date`
- `d_Time`
- `d_Timestamp`

Las instancias de estos dominios puede ser instancias temporales independientes o bien atributos embebidos en objetos permanentes.

La clase `d_String`

- La clase `d_String` representa una cadena de caracteres de longitud variable.
- Una instancia de `d_String` se puede inicializar con una cadena de caracteres, otra instancia `d_String` o nada (que inicializa la instancia a una cadena nula).

```
d_String  nullString;       // Constructor nulo
d_String  composer("Mozart");
d_String  popArtist("Michael Bolton");
d_String  singer(popArtist);
d_String *favorite;

composer = "Beethoven"; // Asigna con char *
favorite = new d_String("Richard Marx");
popArtist = *favorite;      // Asigna con d_String
```

La clase `d_String` no soporta directamente la interface `iostream` de C++, pero es posible definir los siguientes operadores:

```
istream &operator>>(istream &is, d_String &s)
{
  char buffer[256];
  is.width(256);       // Limita a 256 chars
  is >> buffer;
  s = buffer;
  return is;
}

ostream &operator<<(ostream &os, d_String &s)
{
  os << (const char *)s;
  return os;
}
```

La clase `d_String` ...

- Los operadores de comparación comparan una instancia `d_String` con otra `d_String` o un vector de caracteres de C++.
- Una función miembro obtiene la longitud de la cadena de caracteres y el operador `[]` accede a los caracteres mediante un índice.

Por ejemplo, la siguiente función traduce a mayúsculas una cadena de caracteres:

```
void makeUpperCase(d_String &s)
{
  register unsigned long i;
  register const unsigned long len = s.length();

  for (i = 0; i < len; i++){
    char c = s[i];
    if ( isalpha(c) && islower(c) )
    s[i] = (char) toupper(c);
  }
}
```


La clase `d_String` ...

Las clases permanentes que requieren un atributo de tipo cadena de caracteres pueden utilizar tanto un `d_String` como un vector de caracteres.

- El vector de caracteres tiene longitud fija.
- Una instancia `d_String` contiene una cadena de longitud variable.

```
class Empleado : public Persona
{
  char                genero[2]; // "H","M"
  d_String            direccion;
}
```

- Con objeto de dar soporte a cadenas de caracteres de longitud variable, la base de datos orientada a objetos almacena el contenido de la cadena fuera del objeto.
- Esta aproximación puede dar lugar a un cierto costo adicional en la gestión de este tipo de objetos, aunque en general es mínimo.
- Cuando se emplean cadenas de caracteres, los datos se encuentran directamente embebidos en el objeto, de modo que no es necesario gestionar y acceder a datos secundarios.
- Una cadena de caracteres se gestiona generalmente de forma más rápida, por lo que es una buena alternativa cuando la cadena es pequeña y varía poco en longitud.

La clase `d_Interval`

- La clase `d_Interval` se utiliza para representar un lapso de tiempo.
- El resto de las clases relacionadas con tiempo y fechas también la utilizan para determinadas operaciones aritméticas.
- Una instancia se puede inicializar con valores de días, horas, minutos y segundos.
- El día es la unidad más grande que puede especificarse con `d_Interval`.
- La función `is_zero` puede utilizarse para comprobar si todas las componentes son cero.

La siguiente función inserta los lapsos de tiempo en el *stream* `os`:

```
ostream &operator<<(ostream &os, d_Interval &i)
{
  os << "Intervalo (" << i.day() << " días ";
  os << i.hour() << " horas ";
  os << i.minute() << " minutos ";
  os << i.seconds() << " segundos)";
  return os;
}
```


La clase `d_Interval`

La clase `d_Interval` proporciona un conjunto completo de operaciones aritméticas y funciones de comparación. Por ejemplo, admite componentes no normalizados, normalizando el valor cuando se accede a los componentes.

En las siguientes líneas de código, todas las variables acaban conteniendo el valor 36 horas:

```
d_Interval inter1(1,12);          // 1 dia, 12 horas
d_Interval inter2(inter1);        // como antes
d_Interval inter3(0,36);          // como antes
d_Interval inter4(0,24,12*60);    // como antes
d_Interval inter5(0,12);          // 12 horas
d_Interval inter6(6);             // 6 dias
d_Interval inter7;                // todo ceros
d_Interval inter8(0,-12);         // -12 horas
d_Interval inter9(2,-12);         // 1 dia, 12 horas

if ( inter5 < inter1 )
   inter5 *= inter1;
if ( inter1 == inter2 )
   inter6 = inter6 / 4;
if ( inter1 + inter2 == inter3 * 2 )
   inter7 = inter1 * 2 - inter2;
inter8 = d_Interval(6) - 3 * inter9;
```

Las instancias de `d_Interval` se utilizan frecuentemente cuando se lleva a cabo algún tipo de operación aritmética en instancias de las clases `d_Date` y `d_Timestamp`.

La clase `d_Date`

La clase `d_Date` es una abstracción que consiste en año, mes y día. Si una instancia se inicializa sin argumentos, se le asigna el valor de la fecha actual. Una función estática, `current`, devuelve una instancia `d_Date` con la fecha actual. Los componentes de la fecha se pasan al constructor como se indica:

```
d_Date    today;    // Inicializado a la fecha actual
d_Date    cumple(1999, 12, 14); // Year, Month & Day
```

Supongamos que se intenta inicializar una instancia con valores ilegales, como en:

```
d_Date    badDate(1996, 13, 2);
```

En estos casos se genera la excepción `d_Error_DateInvalid`.

Para evitarlo, el estándar proporciona una función booleana, `is_valid_date`, para verificar la validez de una fecha.

Es posible acceder a todas las componentes de la abstracción:

```
ostream &operator<<(ostream &os, d_Date &d)
{
  os.width(2); os << d.month();
  os << '/';
  os.width(2); os << d.day();
  os << '/';
  os << d.year();
  return os;
}
```

- `Weekday` es una enumeración embebida en la clase `d_Date` que identifica los días de la semana, por ejemplo `Monday` o `Tuesday`.
- a enumeración denominada `Month` se utiliza para identificar los meses del año (`January`, `February`, etc...).

La clase `d_Date` ...

El siguiente código realiza algunas operaciones útiles sobre varias fechas:

```
d_Date    day;    // Dia actual
d_Date    today(d_Date::current()); // Lo mismo

cout << "Hoy es " << today << endl;
cout << ++day << " mañana\n"

day = today + d_Interval(1);
cout << day << " mañana\n";

day = today;
cout << --day << " ayer\n";

day = today;
day -= 7;
cout << day << " mismo día, semana pasada\n";

day += d_Interval(7);
cout << day << " mismo día, semana que viene\n";

day = today;
day.next(d_Date::Tuesday);
cout << day << " próximo martes\n";

day = today;
day.previous(d_Date::Friday);
cout << day << " último viernes\n";

int days = today.days_in_month() - today.day();
cout << days << " días para cobrar!\n";
```

La clase `d_Date` ...

Las siguientes función calcula el número de días hasta Navidad. Si las navidades del año actual ya han pasado, determina el número de días hasta las navidades del año siguiente:

```
int shoppingDaysTillNextChristmas()
{
  d_Date today;            // Fecha actual
  d_Date christmas(today.year(), d_Date::December, 25);

  if ( today > christmas ) {
    christmas = d_Date(today.year()+1,
                d_Date::December, 25);
    return christmas.day_of_year() +
           today.days_in_month() - today.day() + 1;
  }
  return christmas.day_of_year() - today.day_of_year();
}
```

El ODMG 2.0 añade una función que permite substraer dos instancias `d_Date` y que devuelve una instancia `d_Interval`:

```
int shoppingDaysTillNextChristmas()
{
  d_Date today;            // Fecha actual
  d_Date christmas(today.year(), d_Date::December, 25);

  if ( today > christmas ) {
    christmas = d_Date(today.year()+1,
                d_Date::December, 25);
  }
  d_Interval interv = christmas - today;
  return interv.days();
}
```


La clase `d_Date` ...

Para determinar si una fecha se encuentra entre otras dos fechas, es posible utilizar la función `is_between`.

La siguiente función determina si está vigente la garantía de un determinado producto:

```
d_Date addMonths(const d_Date &d, unsigned int months)
{
  register unsigned int i;
  d_Date date(d.year(), d.month(), 1);

  if ( months >= 12 ) {     // más de un año
    i = months / 12;        // obtiene número de años
    date = d_Date(d.year() + i, d.month(), 1);
    months -= i * 12;
  }
  for ( i = 0; i < months; i ++) {
    date += date.days_in_month();
  }

  // Ajustamos ahora el día del mes
  int dayofmonth = date.days_in_month() < d.day() ?
                   date.days_in_month() : d.day();
  date += dayofmonth - 1; // día actual del mes
  return date;
}

int isWarrantyInEffect(d_Date &purchase,
                       unsigned int months)
{
  d_Date warrantyDate( addMonths(purchase, months) );
  return d_Date::current().is_between(purchase,
                                      warrantyDate);
```

```
}
```


La clase `d_Date` ...

Un periodo de tiempo viene determinado por una fecha de inicio y una de fin. La función `overlap` determina si dos periodos de tiempo se solapan. Supongamos que los usuarios de una compañía aérea han programado una serie de viajes que no se solapan. La lista de viajes es una lista encadenada simple ordenada por las fechas de viaje:

```
struct Trip {
  Trip(const d_Date &sd, const d_Date &rd,
       struct Trip *n) : startDate(sd),
                         returnDate(rd), next(n) {}
  d_Date  startDate;
  d_Date  returnDate;
  struct Trip *next;
};
```

La función `newTrip` determina si es posible programar un nuevo viaje para el usuario. Esta función admite como parámetros las fechas de inicio y fin de viaje y un puntero que apunta al primer nodo de la lista (que supondremos que no está vacía).

```
bool newTrip(d_Date sdate, d_Date edate,
             struct Trip *&trips)
{
  register struct Trip **tp;
  for ( tp = &trips; *tp; tp = &((*tp)->next) ) {
    if ( overlaps(sdate, edate,
                  (*tp)->startDate, (*tp)->returnDate)
      return false;
    if ( sdate < (*tp)->startDate )
      break;
  }
  *tp = new Trip(sdate, edate, *tp);
  return true;
```

```
}
```


La clase `d_Time`

La clase `d_Time` se utiliza para representar una hora concreta del día.

- Se gestiona empleando la zona horaria GMT (*Greenwich Mean Time*).
- Los componentes son horas (0–23), minutos (0–60) y segundos (0–59.9999).
- Inicializar una instancia con un valor inválido genera una excepción del tipo `d_Error_TimeInvalid`.
- El constructor nulo genera una instancia con la hora actual. La función `current` también devuelve la hora actual.

Dos ejemplos:

```
d_Time      time1;
d_Time      time2(d_Time::current());

d_Time horaDelBocata(10,30,0.0f);
```

El estándar proporciona varias funciones para acceder a las componentes de una instancia `d_Time`.

```
ostream &operator<<(ostream &os,
                    const d_Time &t)
{
  char fill_char_save = os.fill();

  os.width(2); os.fill(' '); os << t.hour();
  os << ':';
  os.width(2); os.fill('0'); os << t.minute();
  os.fill(fill_char_save);
  return os;
}
```

La clase `d_Time` ...

- Embebido en la clase `d_Time` existe un `enum Time_Zone` con los nombres de las diferentes zonas horarias. La clase mantiene una zona horaria por omisión que se inicializa con el valor de la zona horaria local en uso. La gestión de zona horaria se realiza mediante las funciones:
  - `set_Default_Time_Zone(Time_Zone)`: Modifica el valor de la zona horaria en uso.
  - `set_Default_Time_Zone_to_local()`: Restaura la zona horaria al valor local.
- La clase `d_Time` cuenta con funciones aritméticas que utilizan instancias de la clase `d_Interval`. Además, las instancias de la clase `d_Interval` se pueden sumar y substraer a una instancia `d_Time`. La substracción de dos instancias de la clase `d_Time` da como resultado una instancia de la clase `d_Interval`.
- Los operadores de igualdad, desigualdad y comparación se pueden utilizar para comparar dos instancias `d_Time`. La clase también proporciona varias funciones `overlap` para determinar si dos periodos (especificados mediante dos instancias `d_Time`) solapan.

La clase `d_Timestamp`

La clase `d_Timestamp` proporciona una fecha y una hora y define varias funciones para acceder a cada uno de estos componentes.

- La función estática `current` devuelve una instancia `d_Timestamp` con la fecha y hora actuales.
- Como en el resto de clases orientadas a fechas y tiempos, esta clase proporciona operaciones aritméticas, y varias funciones `overlap` que operan sobre las dos instancias `d_Timestamp` que definen un periodo.

La clase `d_Timestamp` proporciona más ventajas que el mero apareo de instancias `d_Date` y `d_Time`: las instancias de esta clase se tratan como un único valor integrado respecto a las operaciones de actualización. Por ejemplo, consideremos el siguiente código:

```
d_Timestamp   ts(1999, 12, 31, 8);
d_Interval    hr_18(0, 18);

ts += hr_18;
```

El valor final de `ts` es 1 de enero de 2000 a las 2:00 AM.

Restricciones de uso de algunos tipos

Uniones

Una **union** C++ define una unidad de almacenamiento que contiene un valor que puede ser de alguno de los tipos de datos especificados en la definición.

```
union multi_tipo {
  double          float_val;
  d_Ref<amount>   amount;
  unsigned short  smallInt;
};
```

Se utiliza cuando los recursos de memoria son escasos y sólo se necesita uno de los tipos en un momento dado. Para determina el tipo de dato almacenado se emplea un indicador (*type flag*):

```
class MiClase {
  multi_tipo        valor;
  int               tipoValor;
  // Otros atributos...
};

void procesa_MiClase(MiClase *mp)
{
  switch (mp->tipoValor) {
  case 1:
       // Utiliza mp-¿valor.float_val
       break;
  case 2:
       // Utiliza mp-¿valor.amount
       break;
  case 3:
       // Utiliza mp-¿valor.smallInt
       break;
  }
```



```
}
```

Campos de bits

La representación binaria de los campos de bits varía entre arquitecturas de máquina y entornos de compilación. Una base de datos puede dar soporte y garantizar la correcta implementación de los campos de bits en un entorno homogéneo (compilador, sistema operativo y procesador), pero es poco probable que funcione correctamente si alguno de estos elementos cambia.

Punteros y referencias C++

Las referencias C++ embebidas en objetos son también problemáticas. Por ejemplo, las siguiente clase cuenta con un miembro, denominado `ciudad` que hace referencia a una clase `Ciudad`:

```
class Direccion {
  Ciudad &ciudad;
  // Otros atributos
};
```

El estándar ODMG no soporta punteros o referencias C++ como atributos de un objeto almacenado en la base de datos. Una referencia C++ no puede almacenarse en la base de datos ni utilizarse para hacer referencia a un objeto persistente. Sólo puede utilizarse para hacer referencia a objetos transitorios. En las clases persistentes es necesario utilizar instancias de la clase `d_Ref<T>` en lugar de punteros y referencias.

Entidades persistentes

- Una de las actividades principales durante la fase de análisis de un proyecto es descubrir todas las entidades en el dominio del problema que deben ser diseñadas. Se define una clase para cada entidad que debe representarse en la aplicación.
- Una entidad consiste en un conjunto de propiedades que también es necesario determinar en la fase de análisis. Esas propiedades incluyen atributos (de algún tipo de dominio), relaciones (con otras instancias) y operaciones. Se representan en las clases mediante datos y funciones.
- Las instancias de las entidades normalmente se relacionan con otras instancias en el modelo orientado a objetos. Las relaciones están incluidas en la declaración de la clase. En las clases implicadas en las relaciones se embeben objetos especiales, proporcionados por muchas de las implementaciones de bases de datos orientadas a objetos. Discutiremos estos objetos más adelante.
- Las operaciones también están asociadas con las clases y se implementan como funciones miembro. Deben proporcionar una interface que encapsula la abstracción y que lleva a cabo todas las modificaciones sobre las instancias. Encapsular la implementación permite definir exactamente la semántica y minimiza la cantidad de software que necesita "conocer" los detalles de la clase.
- Es frecuente que una entidad tenga algún tipo de relación `IS-A` con otras entidades. Esas relaciones se representan en C++ mediante relaciones de herencia entre clases. Una relación `IS-A` significa que existe una relación "es un tipo de" entre clases (por ejemplo, un coche es un tipo de vehículo).

Clases persistentes

En una base de datos C++, la declaración C++ de las clases actúa como esquema de definición de la base de datos. Esta es una de las mayores ventajas de una base de datos orientada a objetos.

La clase base `d_Object`

Cuando una aplicación utiliza la interface del ODMG, todas las clases cuyas instancias deben almacenarse en la base de datos deben derivar, directa o indirectamente, de la clase base `d_Object`.

```
class Persona : public d_Object {
public:
    d_String        apellido;
    d_String        nombre;
    // Otros atributos de persona
};
```

La relaciones de herencia también se pueden definir entre clases persistentes:

```
class Empleado : public Persona {
public:
    // Propiedades de un empleado
};
```

Ya que la clase `Persona` deriva de `d_Object`, `Empleado` también tiene capacidad de persistencia.

Instancias

- El hecho de que la clase `d_Object` proporcione persistencia no implica automáticamente que todas las instancias de cualquier clase que derive de ella sean persistentes.
- Las instancias pueden definirse como transitorias y es la llamada al operador `new` específico lo que determina esta propiedad.
- Cada instancia persistente tiene un identificador de objeto (*Object IDentifier*) que se emplea para referenciar al objeto. El identificador está asociado a la instancia hasta que ésta es borrada y es inmutable: la aplicación no tiene acceso al OID.
- Sólo los atributos pueden modificarse y esto no afecta a la identidad del objeto. En este punto existe una gran diferencia con los sistemas relacionales, en donde el identificador se almacena en una columna de la tupla.
- La clase `d_Ref<T>` se emplea para representar referencias a objetos.
- Cuando se almacena una instancia `d_Ref<T>` en la base de datos, ésta contiene el identificador del objeto referenciado.

Propiedades

Las propiedades de una clase incluyen a los atributos y a las funciones. Cada propiedad tiene nombre y tipo (dominio). Se pueden utilizar los especificadores de control de acceso de C++ `public`, `private` y `protected` para controlar el acceso a las propiedades de la clase. La interface pública de una clase debe consistir en el conjunto de operaciones que representan la abstracción que se está diseñando.

Atributos

- Los atributos se emplean para describir las características de la entidad que se está diseñando. Los atributos no tienen identidad (OID) y no se pueden referenciar en la base de datos de forma independiente.
- Un objeto establece el "contexto" en el cual residen sus atributos. Los atributes proporcionan los valores que definen el estado del objeto que los contiene.
- Es una buena política de diseño de objetos declarar privados los atributos de un objeto. Las operaciones son las que controlan todas las modificaciones y accesos a los atributos: son las responsables de mantener la integridad semántica del objeto.

Existen dos categorías de atributos que tienen un tratamiento diferente:

- **Atributos estáticos:** En C++ el atributo estático de una clase es una única copia que es compartida por todas las instancias de la clase.
  - El atributo estático tiene ámbito de clase y no está contenido en cada instancia.
  - Estos miembros no se almacenan en la base de datos, sino que residen en la sección de datos de cada proceso de aplicación que incluye la clase.

Atributos ...

- **Atributos transitorios:** Algunas bases de datos orientadas a objetos permiten declarar atributos transitorios: existen en el objeto mientras éste está en la memoria de la aplicación, pero no se almacenan en la base de datos. Estos atributos transitorios son útiles para representar datos directamente asociados al objeto durante la transacción pero no es necesario almacenarlos en la base de datos. Los datos pueden ser necesarios cuando se está llevando a cabo algún tipo de cálculo en una transacción pero cuyo valor no es válido en el contexto de otra transacción. La sintaxis varía entre los diferentes productos de ODBMSs.
  - Algunos sistemas tratan todos los punteros como atributos transitorios.
  - Otros sistemas requieren preceder la declaración del atributo con la palabra clave `transient`.

    El estándar ODMG no contempla esta capacidad.

  Supongamos que se está utilizando una clase denominada `Formulario` de la interface de usuario para mostrar los atributos del objeto en una ventana. Puede ser conveniente que una instancia de `Persona` haga referencia al objeto `Formulario` al que está asociado directamente:

```
class Persona : public d_Object {
  // Otros atributos de persona
  transient Formulario *_form;
};
```

  La utilidad de captura de esquemas que analiza la declaración C++ de la clase encontrará los atributos transitorios y tendrá en cuenta que no deben ser almacenados.

Operaciones

- Las operaciones públicas deben proporcionar la interface que representa a la abstracción que se está diseñando. Son responsables de mantener la integridad semántica de una instancia, decidiendo que modificaciones son válidas.
- Estas operaciones se pueden considerar las "minitransacciones" definidas para un objeto.
- Cada transacción (identificada en la etapa de análisis) que opera sobre una instancia debe definirse como una función miembro.
- También pueden emplearse para implementar "atributos derivados". Por ejemplo, la clase `Persona` puede definir una operación `edad`:

```
d_UShort Persona::edad() const
{
  d_Date today = d_Date::current();
  d_UShort years = today.year() - _birth.year();
  if ( today.day_of_year() < _birth.day_of_year() )
     --years;
  return years;
}
```

- Si tanto los atributos almacenados como los derivados se acceden vía operaciones no existe diferencia sintáctica en la forma de acceso. Esta aproximación, conocida como diseño de datos funcional (*functional data modeling*), proporciona una gran flexibilidad ya que permite modificar la implementación sin afectar a las aplicaciones.
- Si todos los accesos se implementan mediante operaciones es posible añadir funcionalidad a una operación que se ejecuta cada vez que se accede al atributo, proporcionando operaciones de tipo *trigger*. Los atributos derivados y los *triggers* se utilizan frecuentemente en otras tecnologías de bases de datos.

Creación de instancias

- Una instancia se crea mediante una llamada al operador `new`.
- Este operador se define en el estándar ODMG como miembro de la clase `d_Object`, sin embargo algunas implementaciones añaden operadores adicionales para sacar partido de las características específicas del ODBMS concreto.
- El operador `new` específico utilizado para crear la instancia determina si ésta será transitoria o persistente.
- Una vez creada, una instancia transitoria no puede convertirse en persistente ni a la inversa.

Creación de instancias transitorias

Dos vías para crear una instancia transitoria mediante la interface del ODMG:

- **Automática:** Todas las variables locales de una clase con capacidad de persistencia son transitorias. Por ejemplo, la siguiente función declara una instancia de la clase `Persona` que se comporta como una variable local:

```
void fichaje()
{
  Persona candidato;
  ...
}
```

  La variable local `candidato` no se almacena en la base de datos.

Creación de instancias transitorias ...

- **Mediante el operador `new`:** El operador `new` admite dos formas para crear instancias transitorias. Por ejemplo, el siguiente código crea dos instancias transitorias de la clase `Empleado` derivada de la clase `Persona`:

```
Empleado *emp1, *emp2;
emp1 = new Empleado();
emp2 = new(d_Database::transient_memory,
           "Empleado") Empleado();
```

  - El primer operador `new` emplea la sintáxis estándar sin parámetros adicionales. Crea una instancia transitoria como cualquier otro operador `new` de C++.
  - El segundo operador `new` utiliza un argumento que es un puntero a un objeto `d_Database`. Este puntero se utiliza para indicar en qué base de datos debe emplazarse el nuevo objeto. El dato estático público de la clase `d_database`, `transient_memory`, se utiliza para indicar que el nuevo objeto es transitorio.
  - Si el primer argumento del operador `new` hubiese sido un puntero a una base de datos abierta, la instancia creada sería persistente y se colocaría en esa base de datos.

  Algunas implementaciones necesitan conocer el tipo de instancia que se está creando. Desgraciadamente, esta información no está disponible de forma automática en el operador `new`, por lo que debe incluirse en la llamada.

Creación de instancias persistentes

Una instancia persistente se puede crear utilizando el siguiente operador `new`:

```
d_database db;      // Apunta a una db abierta
Empleado *emp3 = new(&db, "Empleado") Empleado();
```

- La instancia de `d_database` debe referirse a una base de datos abierta.
- El operador `new` es la misma función empleada en el ejemplo anterior, pero en este caso la instancia `Empleado` creada es persistente y se almacena en la base de datos apuntada por `db`.

Agrupación de instancias

Cuando se accede frecuentemente a un conjunto de objetos relacionados puede resultar útil agruparlos físicamente en la base de datos.

Es posible indicar algunas opciones de agrupamiento cuando se crea la instancia. El agrupamiento (*clustering*) es una técnica de optimización que las aplicaciones pueden utilizar para:

- Reducir el número de transferencias a disco.
- Reducir la cantidad de memoria *cache* del cliente y del servidor requerida para acceder al objeto.
- En algunas arquitecturas, pueden incluso reducir el coste de las comunicaciones cliente–servidor.
- En aquellas implementaciones que proyectan directamente los bloques de la base de datos en páginas de memoria del cliente, el agrupamiento reduce la cantidad de memoria que requiere el cliente.

Agrupación de instancias ...

El ODMG proporciona una interface para especificar que una instancia debe agruparse:

```
Empleado *emp4 = new(emp3, "Empleado") Empleado();
```

- El nuevo objeto `emp4` se agrupará "cerca" del objeto referenciado por `emp3` y en la misma base de datos.
- El grado de "proximidad" que puede especificarse varía entre las diferentes implementaciones: puede ser la misma página, el mismo segmento o simplemente la misma base de datos. Algunos proveedores proporcionan capacidades de agrupamiento específicas, pero no son portables entre diferentes implementaciones del ODMG
- Algunas implementaciones son capaces de agrupar objetos de forma automática; la aplicación no necesita especificar el agrupamiento cuando se crea el objeto. El agrupamiento automático se puede fundamentar en la clase de objeto creado o en las relaciones con otros objetos de la base de datos.

Inicialización de instancias

En una base de datos orientada a objetos se emplean dos formas de inicializar un objeto:

- Cuando un objeto se crea por primera vez, es necesario inicializar cada atributo. En C++ son los constructores los que se encargan de llevar a cabo esta tarea sobre todos los objetos creados por la aplicación.
- La segunda forma de inicialización se produce cuando un proceso de aplicación accede por primera vez a un objeto de la base de datos creado con anterioridad por otro proceso de aplicación. Este proceso, en el que el objeto se instancia en la cache de un nuevo proceso, se conoce con el nombre de "activación" y se verá más adelante.

Los constructores:

- Sirven para el mismo propósito en una base de datos C++ que en otros entornos orientados a objetos.
- Una aplicación puede definir tantos constructores como sea necesario para una clase.
- Aquellos atributos para los que no se especifique un valor inicial deben inicializarse con un valor nulo, por lo que es recomendable definir un constructor nulo para cualquier tipo definido por el usuario.

Inicialización de instancias ...

Como ejemplo, veamos cuales pueden ser los constructores para las clases `Persona` y `Empleado`.

```
Persona::Persona(const char *fname, const char *lname,
                 d_Date bd, const struct Address &addr,
                 const char *phone) :
                 _f_name(fname), _l_name(lname),
                 _birth(bd), _address(addr)
{
  strncpy(_phone, phone, 10);
  _phone[10] = 0;
  _form = new Form();
}

Persona::~Persona()
{
  delete _form;
}

Empleado::Empleado(const char *fname, const char *lname,
                   d_Date bd, const struct Address &addr,
                   const char *hphone, const char *ophone,
                   const char *email, d_Float sal) :
                   Persona(fname, lname, bd, addr, hphone),
                   _o_phone(ophone), _email(email),
                   _salary(sal)
{ }
```


Borrado de instancias

- En una base de datos orientada a objetos, un objeto no es borrado de forma explícita por una aplicación hasta que no es borrado de la base de datos.
- El borrado de un objeto de la cache del cliente es un proceso denominado "desactivación" y se tratará en la siguiente sección.
- Cuando se borra un objeto se invoca a la función destructor si ésta se ha definido. Esta función lleva a cabo las operaciones de "limpieza" necesarias para mantener la integridad de los datos, incluyendo operaciones tales como borrar los objetos secundarios gestionados por el objeto.
- Este proceso se conoce con el nombre de propagación del borrado (*delete propagation*).

El estándar ODMG proporciona dos mecanismos de borrado. Los objetos pueden borrarse usando:

- El operador C++ `delete`.
- El operador `delete_object` definido en la clase `d_Ref<T>`.

Los objetos `Empleado` a los que apuntan `emp3` y `emp4` pueden eliminarse de la base de datos mediante ambos mecanismos:

```
d_Ref<Empleado> empRef(emp3);

delete emp4;
empRef.delete_object();
```

Borrado de instancias ...

- Si la aplicación cuenta con un puntero al objeto que debe borrarse es posible utilizar el operador `delete`. El puntero debe estar inicializado con la dirección de memoria del objeto en la cache de la aplicación: el objeto debe estar ya en la cache.
- Si la aplicación sólo cuenta con una referencia `d_Ref<T>` al objeto es posible utilizar la función `delete_object`. Cuando se invoca, el objeto puede estar o no en la cache del sistema. Para ejecutar el destructor, la instancia debe estar activada en la cache.

El proceso:

- Una vez que se ejecuta el destructor, el objeto se elimina de la cache de la aplicación.
- Si la transacción se valida con éxito, el objeto es entonces eliminado de la base de datos.
- Si la transacción aborta, el objeto permanecerá en la base de datos así como cualquier otro objeto borrado por el destructor.
- Las instancias transitorias no se ven afectadas, normalmente, por las actividades de transacción.

Características:

El destructor definido en `d_Object` se define como una función virtual, de modo que se puede utilizar un puntero a la clase base para borrar una instancia de la clase derivada.

Por ejemplo: una instancia de `d_Ref<Persona>` puede hacer referencia a una instancia `Empleado`. La llamada a `delete_object` generará la llamada correcta al destructor de la clase `Empleado`.

Borrado de instancias ...

Consideraciones de rendimiento:

- Si la clase no cuenta con un destructor y la instancia no se encuentra en la cache de la aplicación, no es necesario activar el objeto cuando se borra.
- De modo similar, algunas implementaciones permiten borrar un objeto sin invocar al destructor.
- Ambas técnicas pueden mejorar el rendimiento: en ocasiones es aceptable obviar el proceso de destrucción y por tanto reducir el coste de procesado que requiere activar el objeto en la cache.
- Sin embargo, es responsabilidad de la aplicación determinar si es necesario o no realizar algún tipo de proceso especial.

Deficiencias:

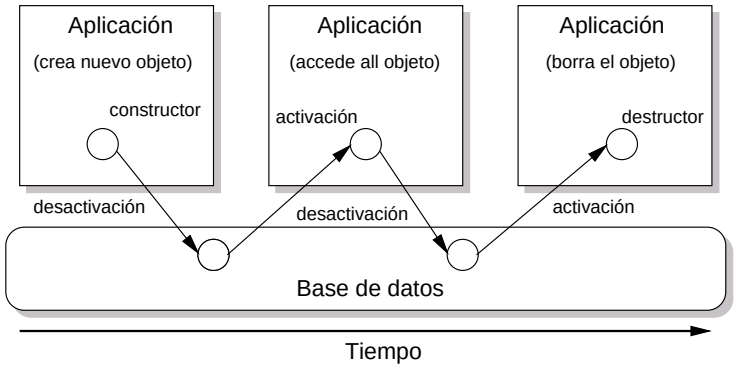
- La llamada a los operadores `delete` y `dele_object` elimina la instancia tanto de la cache de la aplicación como de la base de datos.
- En algunas circunstancias, puede ser conveniente liberar espacio en la memoria cache eliminando algunos objetos de la cache pero no de la base de datos (especialmente en aquellas aplicaciones que operan sobre un conjunto de objetos tan grande que no cabe en la cache).
- Desgraciadamente, ODMG no proporciona ningún mecanismo para eliminar objetos de la cache de la aplicación sin eliminarlos de la base de datos.
- Algunas aplicaciones ODBMS proporcionan soluciones propietarias. Una aplicación puede marcar el objeto como *unpinned*, permitiendo que el software de la base de datos lo pase al área de intercambio (*swap*).

Activación y desactivación

A medida que los objetos son accedidos desde la base de datos por una transacción son movidos hacia y desde la cache de la aplicación. Estos movimientos se denominan *activación* y *desactivación*.

- Cuando un objeto es activado, puede ser necesario llevar a cabo algún proceso de inicialización de los datos transitorios.
- Del mismo modo, cuando un objeto abandona la cache, puede ser necesario realizar algún tipo de operación de finalización.
- En general, será necesario asignar memoria dinámica cuando el objeto entra en la cache y liberarla cuando la abandona.
- La desactivación tendrá lugar cuando la transacción se valida o cuando el objeto es pasado al área de intercambio (*swapped out*).

En la figura se ilustra el proceso de activación y desactivación de un objeto.



Las funciones `d_activate` *y* `d_deactivate`

Las funciones virtuales `d_activate` y `d_deactivate` de la clase `d_Object` son invocadas por el software de la base de datos cuando tienen lugar los eventos de activación y desactivación. Cualquier funcionalidad que la aplicación deba realizar sobre los objetos debe emplazarse en esas funciones.

- La aplicación puede requerir algún tipo de proceso cuando se activa un objeto, tales como acceder a objetos relacionados o refrescar la interface gráfica de usuario gestionada por la aplicación. Este proceso puede llevarse a cabo en `d_activate`.
- Las funciones `d_activate` y `d_deactivate` juegan un papel similar a los constructores y destructores de una clase C++.
- Estas funciones gestionan cualquier inicialización de datos transitorios o cualquier otro tipo de procesado requerido por la aplicación.
- Es importante dejar claro que los atributos persistentes no necesitan ser inicializados, ya que sus valores se almacenan en la base de datos. Sin embargo, puede ser necesario inicializar los parámetros transitorios.

Las funciones `d_activate` *y* `d_deactivate` *...*

- Supongamos que el objeto mostrado en la figura anterior es una instancia de la clase `Empleado` derivada de la clase `Persona`.
- Supongamos también que hemos definido las funciones virtuales `d_activate` y `d_deactivate` para dicha clase.

El orden de ejecución del constructor, destructor, `d_activate` y `d_deactivate` es como sigue:

```
Aplicación 1 comienza
   Persona::Persona
   Empleado::Empleado          La aplicación crea Empleado
   Empleado::d_deactivate      Invocado en tiempo de commit
Aplicación 2
   Empleado::d_activate        Se accede al objeto Empleado
   Empleado::d_deactivate      Invocado en tiempo de commit
Aplicación 3
   Empleado::d_activate        Se accede al objeto Empleado
   Empleado::~Empleado         Se borra el objeto
   Persona::~Persona
```

Es interesante notar que:

- No se llama a las funciones `d_activate` y `d_deactivate` de la clase `Persona`.
- El software de la base de datos no llama a cada una de las funciones `d_activate` y `d_deactivate` en la jerarquía de herencia.
- Se recomienda que las aplicaciones emulen el comportamiento de los constructores y destructores estándar cuando implementan `d_activate` y `d_deactivate`.
- Por tanto, cada función `d_activate` de la clase derivada debe llamar primero a la función `d_activate` de la clase base.

Las funciones `d_activate` *y* `d_deactivate` *...*

Por ejemplo:

```
void Empleado::d_activate()
{
  Persona::d_activate();
  // activación específica de Empleado
}

void Empleado::d_deactivate()
{
  // desactivación específica de Empleado
  Persona::d_deactivate();
}
```

Exactamente el mismo procedimiento debe llevarse a cabo con las funciones `d_deactivate`.

Nota:

- La función `d_activate` no se invoca sobre los objetos cuando se llama a `checkpoint` sobre el objeto `d_Transaction`.
- Cuando se lleva a cabo un *checkpoint*, el estado de todos los objetos en la cache de la aplicación se valida en la base de datos, pero éstos permanecen en la cache para futuras manipulaciones.

Modificación de instancias

- Cuando un objeto es activado en la cache de la aplicación, el software de la base de datos le asigna implícitamente un *lock*, generalmente de sólo lectura que permite el acceso compartido.
- Si la aplicación pretende modificar el objeto, debe invocar a la función `mark_modified` antes de la alteración.
- El estándar ODMG no establece qué sucede cuando la aplicación modifica el objeto antes de invocar a la función `mark_modified`.
- El comportamiento varía entre implementaciones: si no se invoca primero a la función no hay garantía de que las modificaciones realizadas se escriban en tiempo de *commit*.
- Algunas implementaciones utilizan la llamada a `mark_modified` para indicar que se debe activar un *lock* de escritura. En este caso la llamada puede considerarse una forma explícita de obtener el *lock*.

Los sistemas con memoria virtual permiten controlar el acceso de lectura/escritura a las páginas de memoria. Algunas implementaciones emplean esta capacidad para controlar el acceso:

- Cuando el objeto es accedido por primera vez, la página se proyecta en la memoria de la aplicación con acceso de sólo lectura. Si la aplicación intenta modificar el objeto se produce un *trap* a nivel del sistema.
- El software de la base de datos puede capturar ese *trap* y actuar activando el *lock* de escritura para el objeto y cambiando el acceso a la página de memoria a lectura-escritura.

Los ODBMSs que utilizan esta técnica no requieren que la aplicación llame a `mark_modified` para activar el *lock* apropiadamente. Sin embargo esto no es portable.

Modificación de instancias ...

Si la técnica de encapsulado se ha utilizado correctamente, todos los atributos son `private` y sólo pueden modificarse a través de las operaciones de la clase. Esta aproximación presenta dos ventajas:

- Aquellas operaciones que no modifiquen el estado del objeto, pueden declararlo `const`: el compilador puede forzar la naturaleza de sólo lectura de la función.
- Las operaciones que modifican el estado del objeto deben llamar a la función `mark_modified` antes de alterar el atributo.

Una aplicación coherente de esta técnica permite que las aplicaciones que utilizan la clase no necesiten llamar directamente a la función `mark_modified`.

```
void Persona::newAddress(struct Address &addr)
{
  mark_modifed();
  _address = addr;
}
```

- Cuando se invoca a `mark_modified`, la implementación de la base de datos asegura que si la transacción se valida con éxito, los valores de las instancias modificadas se almacenan correctamente.
- La implementación normalmente mantiene un *flag*, denominado *dirty bit*, para marcar qué ha sido modificado (la granularidad del ODBMS puede ser mayor que una instancia individual).
- La base de datos conoce qué instancias han sido modificadas y es capaz de escribir en la base de datos los objetos modificados.
- La aplicación no necesita propagar las modificaciones en la base de datos.

Modificación de instancias ...

Esta técnica supone un gran abaratamiento de los costes de desarrollo. Las aplicaciones que utilizan otras tecnologías de bases de datos deben propagar explícitamente el estado de todos los objetos modificados en la base de datos:

- La aplicación debe saber qué objetos han sido modificados.
- Debe contar con el software apropiado para que cada clase escriba sus instancias en la base de datos.

Comparación con el modelo relacional

1. En una base de datos relacional, una entidad en el modelo se materializa en una tabla, que se define mediante la sentencia `create table` de SQL. Los atributos de la entidad se representan mediante columnas en la tabla y el nombre y tipo de cada columna se define en la sentencia `create table`. Las instancias de la entidad son las filas de la tabla.
2. Las relaciones de herencia entre entidades no pueden declararse entre tablas. Las aplicaciones pueden diseñar y gestionar relaciones de herencia por encima del modelo relacional, pero la base de datos desconoce que éstas existen.
3. La base de datos relacional no proporciona un interface para proyectar las clases entre la base de datos y la representación C++. Todos los accesos a la base de datos, tanto en lectura como en escritura, se realizan en términos de columnas.
4. En una base de datos orientada a objetos, la aplicación accede necesariamente a todos los atributos de una clase, ya que no es posible acceder a sólo un subconjunto de los atributos. En la base de datos relacional lo fácil y natural es acceder a sólo un subconjunto de los atributos.
5. Una base de datos relacional desconoce las instancias utilizadas por la aplicación y no proporciona un mecanismo de cache para la aplicación. La aplicación es responsable de determinar qué objetos es necesario escribir en la base de datos. El desarrollador de la aplicación debe tomar un gran número de decisiones acerca de como gestionar la propagación de las modificaciones en la base de datos y escribir el software correspondiente. En una base de datos orientada a objetos este tipo de cosas se gestionan automáticamente reduciendo substancialmente el costo de desarrollo.

Comparación con el modelo relacional ...

6. Las bases de datos relacionales no soportan encapsulado a nivel de aplicación, ya que desconocen el modelo de objetos utilizado por la aplicación. Mediante SQL, todos los atributos (columnas) son directamente accesibles. Si se requiere control de acceso, es necesario utilizar *views* para limitar el número de columnas accesibles. Las *views* proporcionan un cierto grado de ocultación de información en una base de datos relacional, pero existen algunas restricciones importantes: en general no es posible actualizar las columnas de una *view*.
7. Las columnas de una fila son modificadas directamente mediante el comando `update` de SQL. Algunas bases de datos relacionales permiten especificar restricciones para mantener la integridad semántica, que son comprobadas en el momento de la validación para asegurar que los cambios dejan a la entidad en un estado lógico consistente. La aplicación primero hace los cambios y después se validan en la etapa de *commit*. Esta aproximación difiere substancialmente de la utilizada en una base de datos orientada a objetos, en donde la aplicación define explícitamente qué operaciones son válidas. Sólo el diseñador e implementador de la clase puede alterar los atributos directamente. Esta estructura conduce a un gran nivel de integridad de los datos.

Identificación de objetos

- La técnica primaria para identificar un objeto en una base datos orientada a objetos es el identificador de objeto (OID).
- Otra posibilidad consiste en asignarle un nombre, consistente en una cadena de caracteres ASCII, con un valor determinado por la aplicación. La aplicación puede entonces acceder al objeto mediante el nombre.
- Uno o más atributos de la clase pueden utilizarse como clave para localizar el objeto. Una aplicación puede especificar que quiere acceder al conjunto de todos los objetos que tienen atributos cuyos valores cumplen una determinada condición (*query*).
- Muchos objetos reproducen cosas que existen en el mundo real, algunas de las cuales pueden contar con un identificador propio y diferente del OID. En una base de datos orientada a objetos es posible definir nombres o claves y establecer una relación con el identificador real del objeto en la base de datos.

Identificador de objeto

- El mecanismo fundamental para identificar un objeto es vía su identificador de objeto (OID).
- La base de datos asigna un OID único a cada instancia cuando ésta se crea por primera vez.
- El OID no es un atributo de la instancia y su valor no puede ser cambiado por una aplicación.
- Es la implementación de la base de datos la responsable de generar OIDs únicos.
- Sin embargo, los OIDs son únicos sólo en un ámbito dado (normalmente la base de datos).
- Ya que el ODMG proporciona acceso a una única base de datos, el OID debe ser único dentro de la base de datos. Determinadas implementaciones amplían este ámbito.

En ODMG:

- Se utiliza una instancia de la clase `d_Ref<T>` para referenciar una instancia de la clase `T`, que debe derivar de la clase base `d_Object`.
- Esta instancia puede también referenciar ha instancias de cualquier clase pública derivada de `T`.
- Cuando las instancias de `d_Ref<T>` se almacenan en la base de datos, contienen el OID del objeto referenciado.
- Cuando una instancia `d_Ref<T>` está en memoria, su valor puede ser un OID o una dirección de memoria de la cache que contiene el identificador.
- Las aplicaciones no pueden acceder a la representación real del identificador, que varía significativamente entre proveedores.

Identificador de objeto ...

El siguiente código muestra cómo declarar una referencia a dos instancias de `Persona` y `Empleado`:

```
d_Ref<Persona>    persona;
d_Ref<Empleado>   emp;
```

Una instancia de la clase `d_Ref_Any` puede contener una referencia a una instancia de cualquier clase derivada de `d_Object`:

```
d_Ref_Any    apersona;
```

- Con un `d_Ref_Any` sólo es posible llevar a cabo un subconjunto de las operaciones que se pueden realizar con un `d_Ref<T>`.
- Es posible convertir una referencia `d_Ref_Any` a `d_Ref<T>`.
- Cuando la conversión finaliza es necesario realizar comprobaciones de tipo en tiempo de ejecución para asegurar que el objeto es una instancia de la clase `T` o derivada de `T`.

Ambas clases proporcionan "punteros inteligentes" (*smart pointers*) cuya semántica es similar a la de los punteros, ya que es posible utilizar los operadores de derreferencia `->` y `*`.

- Un referencia `d_Ref<T>` admite herencia y polimorfismo del mismo modo que un puntero C++.
- Supongamos que una clase `D` deriva de la clase `B`. Un `d_Ref<B>` puede referenciar a una instancia de cualquier clase derivada de `B`, incluyendo a `D`.
- La instancia se puede referenciar independientemente de su tipo real, de modo que es posible utilizar referencias polimórficas en la base de datos.

Identificador de objeto ...

El siguiente código crea una instancia de `Empleado`:

```
d_Database db; // Apunta a una db abierta

d_Ref<Persona> persona =
    new(&db, "Empleado") Empleado("Juan", "López");
```

La variable `persona` contiene una referencia a una instancia de `Empleado`:

- Como cuando se utilizan punteros, las aplicación sólo puede acceder a los atributos de `Persona`.
- Sin embargo, es posible llamar a las funciones virtuales empleando la variable `persona` y se invocará la función correcta de la clase `Empleado`.

La derreferencia es una de las operaciones fundamentales en una base de datos orientada a objetos. Para realizar esta operación se utilizan los operadores `->` y `*` definidos en `d_Ref<T>`:

```
persona->_l_name();     // usado más frecuentemente
(*persona)._f_name();
```

- El operador `->` devuelve la dirección del objeto en la cache de la aplicación.
- El operador `*` devuelve una referencia C++ al objeto.
- Si el objeto referenciado ya se encuentra en la aplicación, estos operadores simplemente devuelven un puntero al objeto.
- Si el objeto no se encuentra en memoria, se recupera de la base de datos y se coloca en memoria.

Inicialización y asignación

Hay varias formas de inicializar una instancia de `d_Ref<T>` o `d_Ref_Any`. El constructor nulo inicializa la referencia a un valor nulo. Las siguientes declaraciones de `d_Ref<T>` son válidas:

```
d_Ref<Persona>      persona;
Persona             *p = 0;
d_Ref<Empleado>     empleado;
Empleado            *e = 0;
d_Ref_Any           any;

d_Ref<Persona>      p1(persona);
d_Ref<Persona>      p2(p);
d_Ref<Persona>      p3(empleado);
d_Ref<Persona>      p4(e);
d_Ref<Persona>      p5(any);  // Requiere comprobación
                              // de tipo en tiempo de
                              // ejecución
```

Algunas inicializaciones y asignaciones no son válidas debido a las relaciones de herencia entre clases. Por ejemplo:

```
d_Ref<Departamento>     depto;
Departamento            *d = 0;

// Las clases Departamento y Empleado
// no están relacionadas
depto = empleado;       // ilegal
empleado = d;           // ilegal
// Empleado deriva de Persona, una referencia a
// una clase derivada no puede apuntar a un
// objeto de la clase base
empleado = persona;     // ilegal
empleado = p;           // ilegal
```

Si tiene lugar una inicialización o asignación inválida se produce una excepción `d_Error_TypeInvalid`.

Referencias nulas

- Una referencia puede contener un valor nulo. Las clases `d_Ref<T>` y `d_Ref_Any` cuentan con un constructor nulo que inicializa las referencias a un valor nulo.
- Adicionalmente, la función `clear` se puede emplear para poner un valor nulo en la referencia.
- Existen varias formas de determinar cuando una referencia contiene un valor nulo.
  - La función booleana `is_null` devuelve *true* (no cero) si la referencia es nula y *false* (cero) en caso contrario.
  - El operador `!` devuelve *true* si la referencia es nula.

  El siguiente ejemplo muestra el uso de estas funciones:

```
d_Ref<Empleado>    emp;

if ( emp.is_null() ) ...
if ( !emp ) ...
if ( emp == 0 ) ...
```


Operaciones de derreferencia

Las operaciones de derreferencia se utilizan para acceder a un objeto a través de una referencia.

- La conversión de una referencia de la base de datos a un puntero se denomina *swizzling*.
- Las funciones `->` y `ptr` llevan a cabo el proceso de *swizzling* y devuelven un puntero a la instancia referenciada.
- El operador `*` también lleva a cabo el *swizzling* y devuelve una referencia C++ a la instancia.
- Estos operadores permiten que la clase `d_Ref<T>` cuente con una interface con sintáxis y semántica muy similar a la de un puntero.

```
void referral(const Persona &);
d_Ref<Persona>    persona;
d_Ref<Empleado>   emp;
Persona           *p;

persona->l_name();
referral(*persona);   // referencia C++ al objeto
referral(*emp);
p = persona.ptr();    // dirección en la cache
p = emp.ptr();
```

- Para acceder de forma explícita a la dirección de un objeto en la cache, es necesario invocar la función `ptr`.
- `->` se utiliza para acceder a un miembro del objeto y `*` cuando se requiere una referencia.
- Es conveniente comprobar primero si la instancia es nula.
- Derreferenciar un `d_Ref<T>` nulo genera una excepción `d_Error_RefNull`.

Copia de referencias

La operación de copia de constructores `d_Ref<T>` y `d_Ref_Any` lleva a cabo una "copia superficial" similar a la de los punteros C++.

```
class Foo : public d_Object {
  int           x;
  d_Ref<Bar>    bar;
  // Otros atributos
public:
  Foo(const d_Ref<Bar> &);
  // Supongamos que no existe el
  // constructor copy
};
```

- Si la aplicación no define un operador de copia, el compilador genera una que inicializa miembro a miembro. Cuando se inicializa una instancia `Foo` con otra instancia `Foo`, ambas hacen referencia a la misma instancia `Bar`:

```
d_Database db; // db abierta

d_Ref<Bar>      b = new(&db, "Bar") Bar();
d_Ref<Foo>      foo = new(&db, "Foo") Foo(b);
d_Ref<Foo>      newFoo = new(&db, "Foo") Foo(*foo);
```

- Para aplicar la operación al objeto referenciado por `d_Ref<T>`, la aplicación debe especificar la funcionalidad explícitamente:

```
Foo::Foo(const Foo &f) : x(f.x)
{
  bar = new(&db, "Bar") Bar(*f.bar);
  // Otras inicializaciones
}
```

Mediante el constructor de copia para `Foo`, las dos instancias harán referencia a diferentes instancias de `Bar`. Esta técnica se conoce como "inicialización profunda".

Igualdad y desigualdad

Los operadores de igualdad y desigualdad (`==` y `!=`) están definidos para ambas clase `d_Ref<T>` y `d_Ref_Any`.

- El operador `==` definido para `d_Ref<T>` puede utilizarse para comparar una referencia de la base de datos con una referencia nula:

```
d_Ref<Empleado>  nullEmp;

if ( emp == nullEmp ) ...
while ( emp != nullEmp ) { ... }
```

- La operación de igualdad es superficial, sólo se comparan los identificadores de objetos y no los objetos referenciados. Esto se denomina "identidad".
- La definición del operador `==` para las clases `Foo` y `Bar` del ejemplo anterior se pueden hacer:
  1. Basándose en el valor de `Bar` (ambos `bar` refieren a la misma instancia):

```
int operator==(const Foo &f, const Foo &g)
{
  // Igualdad superficial (identidad)
  return f.x == g.x && f.bar == g.bar;
}
```

  2. En función de los valores referenciados por `bar`:

```
int operator==(const Foo &f, const Foo &g)
{
  // Igualdad profunda
  return f.x == g.x && *f.bar == *g.bar;
}
```

El desarrollador de la aplicación es quien debe determinar cual es la operación apropiada.

Denominación de objetos

- Es posible asociar un nombre a un objeto de la base de datos que consiste en una cadena de caracteres ASCII con un valor determinado por la aplicación.
- El nombre debe ser único en la base de datos y proporciona al usuario una forma de identificar el objeto.
- No es obligatorio que todas las instancias de una clase cuenten con un nombre; es la aplicación quien decide qué objetos deben nombrarse.
- El nombre puede ser un valor arbitrario, pero es recomendable establecer una política de asignación de nombres que se ajuste a las necesidades de la aplicación.
- Sólo un pequeño grupo de objetos de la base de datos tienen nombre. Una vez que se accede al objeto inicial mediante nombre, se recorre la estructura de referencias para acceder a los objetos adicionales.
- El objeto con nombre se denomina "objeto raíz" (*root object*) o *entry point object* por su papel como punto de acceso a los objetos relacionados.

La especificación ODMG presenta algunas limitaciones en la asignación de nombres:

- El ODMG proporciona un único espacio de nombres por base de datos. Si un nombre ya en uso y se intenta asignar a otro objeto se produce una excepción `d_Error_NameNotUnique`.
- En el ODMG-93 1.2, un objeto sólo puede tener un nombre. Si se pone nombre a un objeto que ya lo tiene, el nuevo nombre reemplaza al anterior. En ODMG 2.0 puede tener múltiples nombres.

Denominación de objetos ...

- La función `set_object_name` de `d_Database` asocia un nombre con un objeto.
- Para borrar el nombre (o los nombres) de un objeto, basta con invocar la función `set_object_name` con el argumento `0`.
- En ODMG 2.0 se añade la función `rename_object` para renombrar un objeto, pasando como argumentos el antiguo nombre y el nuevo.
- Para eliminar uno de los varios nombres que puede tener un objeto en ODMG 2.0 basta con invocar a la función `rename_object` utilizando `0` como segundo argumento.

El lenguaje de consulta orientado a objetos, OQL (*Object Query Language*) permite la utilización de nombres para acceder a los objetos: con el fin de facilitar el uso de nombre en OQL, el nombre debe utilizar sólo caracteres alfanuméricos.

Convertir los espacios en caracteres "_" es una técnica para manejar nombres compuestos.

```
d_Ref<Departamento>
createDBDepto(const char *name, const char *fax)
{
  d_Ref<Departamento> depto;
  const char *oname = createDBname(name);
  depto = new(&db, "Departamento")
              Departamento(name, fax);
  db.set_object_name(depto, oname);
  delete oname;
  return depto;
}
```

Denominación de objetos ...

- La clase `d_Database` cuenta con la función `lookup_object` para obtener un `d_Ref_Any` a partir de una cadena ASCII .
- Si no existe ningún objeto con el nombre especificado, la función retorna un `d_Ref_Any` nulo.

El siguiente código se utiliza para acceder a una instancia `Departamento` a través de su nombre `dname`:

```
const char *oname = createDBname(dname);
d_Ref<Departamento> depto = db.lookup_object(oname);
delete oname;
```

- Si en la base de datos existe un objeto con el nombre `oname` pero el objeto no es de la clase `Departamento` o de una clase derivada de ésta, se produce una excepción `d_Error_TypeInvalid`.
- Por tanto, el sistema lleva a cabo comprobaciones de tipo en tiempo de ejecución.

Colecciones

- Las colecciones se utilizan para definir un conjunto de valores de algún tipo.
- Son indispensables en el diseño de objetos y desarrollan muchos papeles en la tecnología de bases de datos orientadas a objetos.
- Son parte constituyente de los fundamentos sobre los que se define el modelo de objetos de la aplicación y se utilizan para representar las relaciones entre los objetos de la base de datos.
- Constituyen también el valor retornado como resultado de una consulta a la base de datos.
- Existen varios tipos de colecciones, cada una con sus características propias.
- Las colecciones difieren en el tratamiento de los valores duplicados, orden de los elementos, acceso y capacidades de iteración.
- Las colecciones más comunes son los *sets*, *bags*, *lists*, *arrays* y *dictionaries*.

Colecciones ...

Son varias las operaciones definidas que operan sobre colecciones:

- Insertar y eliminar elementos son las más comunes.
- Algunas colecciones proporcionan operaciones para determinar si un valor específico es un elemento de la colección.
- Cada instancia colección cuenta con un atributo de cardinalidad, que es el número de elementos que contiene.

Los objetos *iterators* se emplean para iterar sobre los elementos de una colección.

- Dependiendo del tipo de colección, el iterador puede soportar acceso a los elementos en orden directo, inverso o aleatorio.
- Cuando el iterador se asocia a una colección referencia al elemento actual o se encuentra en el estado "fin de iteración".
- Ya que las colecciones y los iteradores operan sobre tipos definidos por el usuario, emplean la capacidad `template` de C++.

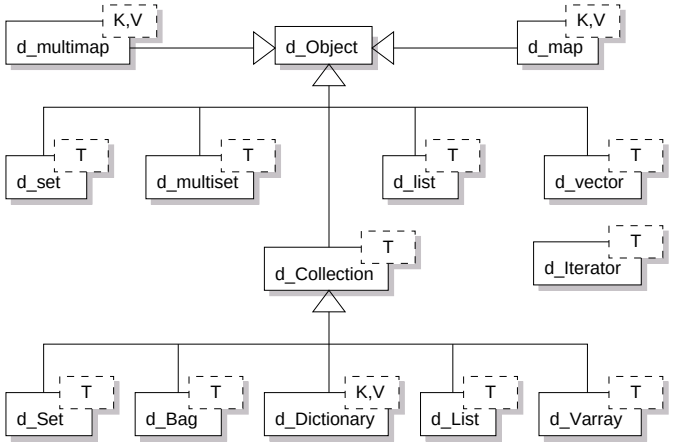
Colecciones e iteradores en ODMG

Todas las clases de colección del ODMG derivan de la clase base `d_Collection<T>`. Las operaciones soportadas por todas las colecciones se definen en dicha clase base.

En la tabla se muestran algunas características de las colecciones ODMG:

Colección	Ordenada	Permite duplicados
`d_Set<T>`	No	No
`d_Dictionary<K,V>`	No	No
`d_Bag<T>`	No	Si
`d_List<T>`	Si	Si
`d_Varray<T>`	Si	Si

En la figura se representan las relaciones de herencia entre las colecciones. La clase iterador `d_Iterator<T>` se define en la interface ODMG y es utilizada por todas las clases de colección.



Colecciones e iteradores en ODMG ...

- La clase base `d_Collection<T>` deriva de `d_Object` de modo que una colección puede ser un objeto independiente y persistente en la base de datos.
- Una colección persistente tiene un identificador de objeto y puede referenciarse en la base de datos.
- Puede también embeberse como un atributo de una clase persistente, pero en estas circunstancias funciona como un dominio, carece de OID y no puede ser referenciada directamente.

El ODMG 2.0 da soporte a un subconjunto de las colecciones del STL (*Standar Template Library*). Estas colecciones derivan también de la clase `d_Object` y pueden ser persistentes. Esto es aplicable a las siguientes colecciones:

- `d_set<T>`.
- `d_multiset<T>` (similar a un *bag*).
- `d_vector<T>` (similar a un *array*).
- `d_list<T>`.
- `d_map<K,V>`.
- `d_multimap<K,V>`.

Sets

- La clase plantilla o patrón `d_Set<T>` del ODMG representa a un conjunto de elementos del tipo `T`.
- Un *set* es una colección desordenada de elementos sin duplicación.
- Cuando se inserta un valor en un *set*, la implementación debe determinar si el valor es ya un elemento del *set* (algo en lo que es muy eficiente).
- El *set* debe también ser capaz de determinar si dos valores son iguales.
- Esto último implica que el operador igualdad debe estar definido para el elemento y ser accesible por la implementación de *set*.

Bags

Un *bag* es una colección desordenada que permite la repetición de valores.

Lists

Una *list* es una colección ordenada de valores en los que se permite la duplicación.

- Cada elemento de la lista tiene un índice que determina su posición relativa.
- Proporciona operaciones para insertar nuevos elementos en posiciones determinadas de la lista.
- Está disponible una operación de indización para acceder al elemento en la i-sima posición.

Arrays

El ODMG define la clase `d_Varray<T>` que implementa vectores de longitud variable.

- También proporciona una operación de indización para acceder al i-simo elemento, aunque generalmente de forma mucho más eficiente que una *list*
- Sin embargo, el *list* es más eficiente en la inserción y eliminación de elementos).

Dictionaries

Otro tipo de colección que las aplicaciones encuentran generalmente útil es el *dictionary* (también llamado vector asociativo).

- La clase `d_Dictionary<K,V>` se introdujo en el ODMG 2.0.
- Un diccionario establece una relación entre una clave de un determinado tipo y un valor de otro tipo.
- Debido a que la implementación puede soportar o no la repetición de claves, los diccionarios se construyen normalmente empleando técnicas de *hashing* o estructuras de datos que sacan partido de la ordenación de las claves.
- Por este motivo, el tipo de la clave debe proporcionar una función de *hash* o una función para comparar los valores de sus instancias.
- Algunos diccionarios mantienen sus claves ordenadas, permitiendo la iteración ordenada.
- El tipo de diccionario que comúnmente se encuentra en los modelos orientados a objetos cuenta con una clave de tipo cadena de caracteres que se emplea para localizar un valor de cualquier otro tipo (generalmente la referencia a un objeto).

Tipo de los elementos de una colección

Los elementos de una colección pueden ser de cualquier tipo, aunque dependiendo de la implementación pueden existir algunas limitaciones.

Cada tipo `T` usado como elemento de una colección debe soportar las siguientes operaciones:

```
class T {
public:
                T();
                T(const T &);
                ~T();
          T     &operator=(const T &);
  friend int operator==(const T &, const T &);
};
```

Es importante que la implementación de estas funciones sea lo más eficiente posible, ya que tienen un impacto directo en el rendimiento de la colección que contiene el tipo.

Operaciones comunes sobre colecciones

Gestión de los elementos de la colección

- La función `cardinality` determina el número de elementos de una colección.
- La función `contains_element` determina si un valor se encuentra en la colección. La clase `T` debe proporcionar un operador `==`.
- La función `insert_element` inserta un valor en la colección.
  - Si la colección es un `d_Set<T>` que ya contiene ese valor, no se añade el elemento.
  - Como las clases `d_Bag<T>` y `d_Set<T>` son colecciones sin orden, el nuevo elemento se inserta en cualquier posición.
  - En una colección de la clase `d_List<T>` el nuevo elemento se inserta al final de la lista.
  - De modo similar, en una colección de la clase `d_Varray<T>`, el tamaño del vector se incrementa en una unidad y el nuevo elemento se inserta en la última posición.
- La función `remove_element` elimina un elemento con el valor especificado.
  - Las clases `d_Bag<T>`, `d_List<T>` y `d_Varray<T>` admiten duplicación: el elemento que se borra es el primero al que se accede en la iteración.
  - Cuando se elimina un elemento del tipo `T` se invoca al destructor correspondiente.
  - Si el elemento es de tipo `d_Ref<T>`, sólo se invoca al destructor de esta clase y no al destructor de la clase `T`. Es la aplicación quien debe eliminar el objeto `T` si es necesario.
  - La función `remove_all` elimina todos los elementos o todos aquellos elementos con un determinado valor.

Gestión de los elementos de la colección ...

Notas:

- Cuando se modifica una colección, se invoca automáticamente a `mark_modifed`.
- Esto es válido para las operaciones `insert_element` y `remove_element`.
- Esto puede afectar a la concurrencia.

Copia y asignación de colecciones

- El constructor nulo inicializa la colección con cero elementos. El destructor elimina el objeto colección e invoca al destructor de cada elemento.
- El constructor copia inicializa una nueva colección utilizando la colección que se pasa como argumento.
  - Cada elemento de tipo `T` se copia a partir de la colección existente invocando al constructor copia de dicha clase.
  - Es importante destacar que si el elemento es de tipo `d_Ref<T>`, ambas colecciones se referirán a la misma instancia `T`.
  - Si la colección no está ordenada, la iteración sobre las colecciones puede acceder a los elementos en orden diferente.
  - En las colecciones ordenadas se preserva el orden.

Notas:

- El proceso de copia y asignación puede ser un proceso costoso, especialmente si colecciones son grandes.
- El rendimiento depende también de la eficiencia de las operaciones de inicialización, copia y destrucción (ya que se invocan sobre cada elemento).

Comparación de colecciones

- Los operadores de igualdad y desigualdad (`==` y `!=`) se definen en la clase base `d_Collection<T>`.
- Su semántica depende del tipo de colección que se está comparando.
- Ambas colecciones deben tener el mismo tipo de elementos, pero el tipo de colección puede ser diferente. Por ejemplo, es posible comparar la igualdad de dos colecciones `d_List<T>` y `d_Set<T>`.
- Dos colecciones no pueden ser iguales si tienen cardinalidad diferente: esta es la primera comprobación que se realiza.
- Cuando una colección ordenada (`d_List` o `d_Varray`) se compara con una colección desordenada (`d_Set` o `d_Bag`), ambas colecciones se tratan como desordenadas.
- Sin embargo, cuando se comparan colecciones ordenadas, el orden tiene importancia.

Comparación de colecciones ...

Algunas particularidades:

- Dos instancias `d_Set<T>`, `A` y `B`, son iguales si tienen la misma cardinalidad y si cada elemento de `A` es igual a un elemento de `B`.
- Las colecciones del tipo `d_Bag<T>`, `d_List<T>` y `d_Varray<T>` permiten duplicación de elementos. Cuando una instancia `B` de alguna de estas clases se compara con una instancia `A` de `d_Set<T>`, si `B` contiene algún duplicado entonces ambas colecciones son diferentes.
- Dos instancias `A` y `B` del tipo `d_Bag<T>` son iguales si tienen la misma cardinalidad y si `B` tienen el mismo número de ocurrencias para cada elemento único de `A`.
- Cuando una instancia `A` del tipo `d_Bag<T>` se compara con una colección del tipo `d_List<T>` o `d_Varray<T>`, `A` y `B` deben contar con el mismo número de ocurrencias para cada valor único de `A`. Ya que `A` carece de orden, el orden de los elementos de `B` no afecta a la operación.
- Cuando se comparan instancias `A` y `B` del tipo `d_List<T>` o `d_Varray<T>`, ambas instancias son iguales sólo si tienen la misma cardinalidad y si los elementos en cada posición de `A` son iguales a `B`.

Iteración

- La clase `d_Iterator<T>` del ODMG se utiliza para iterar sobre todos los tipos de colección.
- A partir de la versión 1.2 del ODMG-93, la clase `d_Iterator<T>` soporta la especificación STL para un iterador bidireccional constante. Un iterador constante no permite la modificación directa de los elementos de la colección.
- Se soportan los dos operadores incremento (`++`) y decremento (`--`), tanto en su forma prefijo como sufijo para permitir la iteración hacia adelante y hacia atrás (*forward* y *backward*).
- Sin embargo, las clases `d_Set<T>` y `d_Bag<T>` no permiten la iteración *backwards*, generando una interrupción del tipo `d_Error_IteratorNotBackward`.
- Son posibles varios estilos de iteración

Iteración ...

Ejemplo A:

```
void Departamento::printProject(ostream &os)
{
  d_Ref<Proyecto>                proj;
  d_Iterator<d_Ref<Proyecto>>    pi;

  for ( pi = proyectos.create_iterator();
        pi.not_done();
        pi.advance() ) {
    proj = pi.get_element();
    os << proj->name << endl;
  }
}
```

- La función `create_iterator` devuelve un iterador posicionado en el primer elemento.
- La función `advance` posiciona el iterador en el siguiente elemento.
- La función `not_done` devuelve *true* si la iteración no es completa. La función `get_element` accede al elemento actual de la iteración.

Iteración ...

Ejemplo B:

```
void Departamento::printProject(ostream &os)
{
  d_Ref<Proyecto>              proj;
  d_Iterator<d_Ref<Proyecto>>  pi;

  pi = projectos.create_iterator();
  while ( pi.next(proj) ) {
    os << proj->name << endl;
  }
}
```

- La función `next` de `d_Iterator<T>` derreferencia el iterador, avanza y comprueba el final de la iteración en una única llamada.
- Devuelve, vía un parámetro de referencia, el valor del elemento que era actual antes de la llamada a `next`.

Notas:

- Las funciones `get_element` y `next` no deben utilizarse simultáneamente.
- Tras la llamada a `next`, el iterador se posiciona en el siguiente elemento.
- Dentro del bucle `while`, el argumento `proj` se utiliza para referenciar el elemento actual de la iteración. Si se efectúa una llamada a `get_element` o al operador `*` dentro del bucle, el elemento que se devuelve es el siguiente al devuelto por la función `next`.
- Cuando la iteración alcanza al último elemento, de produce una excepción `d_Error_IteratorExhausted`.

Iteración ...

Ejemplo C:

Las siguientes implementaciones de la función `printProject` utilizan el estilo de iteración de la STL.

```
void Departamento::printProject(ostream &os)
{
  d_Iterator<d_Ref<Proyecto>>  pi;
  d_Iterator<d_Ref<Proyecto>>  pe;

  for ( pi = projectos.begin(), pe = projectos.end();
        pi != pe;
        ++pi ) {
    os << (*pi)->name << endl;
  }
}
```

- La función `begin` definida en `d_Collection<T>` devuelve un iterador posicionado en el primer elemento de la colección.
- La función `end` devuelve un iterador posicionado detrás del último elemento.
- Las funciones `begin` y `end` se añadieron a la clase `d_Collection<T>` por compatibilidad con la STL.
- El operador de derreferencia `*` se redefine en `d_Iterator<T>` para que acceda al elemento actual de la iteración.

Iteración ...

Ejemplo D:

El último estilo de iteración utiliza el algoritmo STL `for_each`.

- Este algoritmo itera desde un iterador posicionado en el primer elemento hasta que se alcanza un segundo iterador que indica el fin de iteración.
- Cada elemento obtenido durante la iteración es pasado como argumento a la función especificada en la llamada `for_each`.

```
void printName(d_Ref<Proyecto> proj)
{
  cout << proj->name << endl;
}

void Departamento::printProjects(ostream &os)
{
  for_each(proyectos.begin(),
           proyectos.end(),
           printName)
}
```


Operaciones específicas del tipo de colección

Subsets* y *Supersets

- Un conjunto `A` es un subconjunto de `B` si cada elemento de `A` está también en `B`.
- El conjunto `A` puede ser un subconjunto de `B` y simultáneamente igual a `B` (`A = B`). En ese caso, se dice que `B` es un superconjunto (*superset*) de `A`.
- Si `A` no es igual a `B`, entonces se dice que `A` es un subconjunto propio (*proper subset*) de `B` y que éste es un superconjunto propio (*proper superset*) de `A`.

La clase `d_Set<T>` proporciona operaciones para determinar si un conjunto es un subconjunto o superconjunto (propios) de otro conjunto dado:

- `d_Boolean is_subset_of(const d_Set<T> &) const`
- `d_Boolean is_proper_subset_of(const d_Set<T> &) const`
- `d_Boolean is_superset_of(const d_Set<T> &) const`
- `d_Boolean is_proper_superset_of(const d_Set<T> &) const`

Operaciones de *set*

Las clases `d_Set<T>` y `d_Bag<T>` soportan las siguientes operaciones de conjuntos:

- Unión `+`. La unión de los conjuntos `A` y `B` es el conjunto de todos los valores que están bien en `A`, bien en `B`. Como los *sets* no admiten duplicación, la unión de los conjuntos contiene sólo un elemento de cada valor de `A` o `B`. En el caso de *bags*, el número de elementos de la unión es la suma de los elementos de `A` y `B`.
- Intersección `*`. La intersección de dos conjuntos `A` y `B` es un conjunto que contiene los valores de están simultáneamente en `A` y en `B`. Si los conjuntos no tienen elementos comunes, se dice que son disjuntos. Si `A` y `B` son *bags*, el número de elementos de un determinado valor `v` en el conjunto intersección es el mínimo del número de elementos `v` en `A` o `B`. Por ejemplo, si `A` y `B` son *bags* de enteros y el número 7 se repite dos veces en `A` y cinco en `B`, el conjunto intersección sólo tendrá dos elementos con el número 7.
- Diferencia `-`. La diferencia entre los conjuntos `A` y `B` (`A-B`) es el conjunto de los valores de `A` que no están en `B`. Esta operación también se denomina "complemento de `A` en relación a `B`". En el caso de *bags*, para cada valor `v` en `A`, el número de valores de `v` en `A-B` es el número de valores de `v` en `A` menos el número de valores de `v` en `B`. Por ejemplo, sea `A = { 1,2,2,2,3,3 }` y `B = { 1,2,2,3,3,3,4 }`, el conjunto `A-B` es `{ 2 }`.

Las operaciones descritas se aplican entre clases del mismo tipo. Si un aplicación necesita llevar a cabo estas operaciones entre una instancia de la clase `d_Set<T>` y otra de la clase `d_Bag<T>`, la instancia `d_Set<T>` debe convertirse previamente en un *bag*.

Operaciones `d_List<T>` y `d_Varray<T>`

- Las clases `d_List<T>` y `d_Varray<T>` soportan acceso indexado a sus elementos. El valor del primer índice es 0, siguiendo el convenio usado en C y en C++.
- Es posible establecer el tamaño de una instancia unidimensional de `d_Varray<T>` en su inicialización o pasando el nuevo tamaño a la función `resize`.
- El tamaño actual de un vector se puede obtener mediante una llamada a la función `cardinality`.
- Existen funciones para recuperar, reemplazar y eliminar elementos en una posición determinada. Si el índice está fuera de rango, se produce una excepción `d_Error_PositionOutOfRange`.
- La función `insert_element`, heredada de la clase base `d_Collection`, inserta el nuevo elemento al final de la colección.
- La colección `d_List<T>` proporciona funciones para insertar un elemento en la primera o última posición de la lista y antes o después de un determinado elemento.
- La colección `d_List<T>` también proporciona funciones para combinar los elementos de dos listas. La definición de estas funciones varía dependiendo de si se modifica la lista original o de si se crea una nueva lista.
  - Las funciones `concat` y `+` crean una nueva lista que contiene los elementos del operando izquierdo seguidos por los del operando derecho.
  - Las funciones `append` y `+=` añaden los elementos del operando al final de la lista original. Esta funciones son más eficientes, ya que el resultado se aplica sobre el objeto que las invoca.

Colecciones con nombre y *extents*

Es frecuente que una aplicación tenga que acceder a todas las instancias de una determinada clase. Existen dos métodos fundamentales para este propósito.

- La especificación ODMG-93 1.2 requiere la utilización de una colección con nombre que contiene una referencia a cada instancia. El nombre de la colección es arbitrario, pero la aplicación es responsable de mantener la colección: cada vez que se crea o destruye una instancia es necesario modificar la colección.
- El conjunto de todas las instancias persistentes de una clase se denomina su *extent* (literalmente extensión). Cuando una transacción crea o borra instancias, los cambios se reflejan automáticamente en el *extent* asociado. Los *extents* son soportados desde la versión ODMG 2.0.

Colecciones con nombre

El mantenimiento de una colección con nombre puede gestionarse mediante los constructores y destructores de la clase.

Colecciones con nombre ...

Una colección denominada `Deptos` se utiliza para contener una referencia a cada instancia de `Departamento`:

```
class Departamento : public d_Object {
public:
  static d_Ref<d_Set<d_Ref<Departamento>>> deptos();
  Departamento (const char *n);
  ~Departamento();
  // Otros atributos públicos

private:
  static const char * const name_deptos;
  static d_Ref<d_Set<d_Ref<Departamento>>> _deptos;
  // Otros atributos privados
};
```

- El miembro estático `_deptos` es una referencia a una colección denominada `Deptos`, que contiene la referencia a cada instancia de `Departamento`.
- Cuando la base de datos se crea, es necesario llamar una vez a una función `create_deptos` para crear la colección con nombre.
- La función `deptos` se puede utilizar entonces para acceder a esta colección.

```
const char * const Departamento::name_deptos = "Deptos";
d_Ref<d_Set<d_Ref<Departamento>>> _deptos;

d_Ref<d_Set<d_Ref<Departamento>>> Departamento::deptos()
{
  if (_deptos.is_null())  // Todavía no accedido
    _deptos = db.lookup_object(name_deptos);

  if (_deptos.is_null())  { // Todavía no creado
     _deptos = new(&db) d_Set<d_Ref<Departamento>>;
     db.set_object_name(_deptos, name_deptos);
  }
  return _deptos;
}
```

Colecciones con nombre ...

Cada constructor de `Departamento` inserta una referencia a su instancia en la colección con nombre mientras que el destructor elimina la referencia de la colección.

```
Departamento::Departamento(const char *n) : name(n)
{
  deptos()->insert_element(this);
  // Otras inicializaciones
}

Departamento::~Departamento()
{
  deptos()->remove_element(this);
  // Código del destructor
}
```

El código anterior supone que todas las instancias son persistentes, algo que no es usual ya que una aplicación normalmente cuenta con una mezcla de instancias persistentes y transitorias de la misma clase.

Extents

- Conceptualmente, un *extent* para la clase `T` es similar a un `d_Set< d_Ref<T> >`.
- Es una colección desordenada que contiene una referencia para cada instancia `T`.
- Una instancia de `d_Extent<T>` no es persistente, proporciona una interface para acceder a las instancias persistentes de `T` a través de iteradores, que pueden actuar sobre toda las instancias o sobre el subconjunto que satisface determinadas condiciones de consulta expresadas en OQL (*Object Query Language*).

Una aplicación que tenga que acceder a todas las instancias de la clase `Persona` y sus subclases debe declarar la siguiente variable:

```
d_Database *dbp;   // Database abierta

d_Extent<Persona>  Personas(dbp);
```

El constructor de `d_Extent` cuenta con un segundo argumento de tipo `d_Boolean` que indica si se deben incluir las instancias de las subclases. El valor por defecto es `d_True`. Si la aplicación no quiere que se incluyan las instancias de las subclases, la declaración debe ser:

```
d_Extent<Persona>    Personas(dbp, d_False);
```

- El *extent* es mantenido de forma automática por la base de datos, por lo que las operaciones para insertar y borrar un elemento no están accesibles para las aplicaciones.
- Siempre que se crean o destruyen instancias, la base de datos actualiza el *extent* de forma adecuada.
- Ya que la aplicación no manipula directamente el *extent*, la concurrencia es mayor que la obtenida con las colecciones con nombre.

Relaciones

- Las relaciones describen las asociaciones entre objetos y pueden diseñarse directamente en una base de datos orientada a objetos.
- Son varios los tipos de relaciones que se pueden definir, con la ventaja adicional de que algunos de estos tipos mantienen automáticamente la integridad referencial.

Definición y propósito de las relaciones

La porción más significativa del software de una aplicación de base de datos está generalmente dedicada al recorrido y mantenimiento de las relaciones entre entidades.

- En una base de datos orientada a objetos, una relación se representa mediante un objeto con un conjunto de operaciones asociadas.
- Las asociaciones entre instancias se establecen, modifican y eliminan invocando operaciones que actúan sobre objetos de tipo relación.
- Estas operaciones son las que se encargan de mantener la integridad referencial entre ambos extremos de la relación.

Relaciones unidireccionales y bidireccionales

Las relaciones pueden ser unidireccionales o bidireccionales:

- En una relación unidireccional, una instancia `A` se relaciona con una instancia `B` pero no al contrario. La relación sólo puede recorrerse desde `A` hasta el extremo `B`.
- Las relaciones bidireccionales son recíprocas; cada una hace referencia a la otra. Si existe una relación bidireccional entre las clases `A` y `B`, entonces la clase `A` hace referencia a `B` y `B` hace referencia a `A`.

Consideremos la representación de una estructura de tipo árbol. La clase `NodoArbol<T>` representa un nodo del árbol:

```
template <class T>
class NodoArbol : public d_Object {
  T                              valor;
  d_Set<d_Ref<NodoArbol<T>>>     nodoHijo;
};
```

Cada nodo del árbol cuenta con un conjunto de referencias a sus hijos. Este ejemplo ilustra una relación unidireccional. Si el recorrido se inicia siempre en la raíz del árbol la aplicación puede utilizar recursión para recorrer el árbol completo.

Para convertir la relación en bidireccional es necesario añadir una referencia al nodo padre:

```
template <class T>
class NodoArbol : public d_Object {
  T                              valor;
  d_Ref<NodoArbol<T>>            nodoPadre;
  d_Set<d_Ref<NodoArbol<T>>>     nodoHijo;
};
```

con lo que una aplicación podría recorrer el árbol en ambos sentidos.

Relaciones recursivas

Algunas relaciones tienen carácter recursivo:

- En una estructura de tipo árbol, por ejemplo, cada nodo tiene una referencia a su padre y referencias a sus hijos (ramas).
- La recursión permite especificar el recorrido de forma concisa.

La clase `Persona` proporciona un ejemplo de ello:

```
class Persona : public d_Object {
public:
    d_Ref<Persona>          padre;
    d_Ref<Persona>          madre;
    d_set<d_Ref<Persona>>   hijos;
};
```

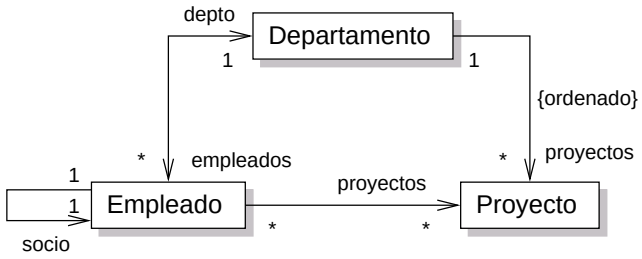
Para acceder al abuelo de un individuo, podríamos utilizar la siguiente expresión:

```
d_Ref<Persona>   persona;
// Inicialización de la instancia persona
d_Ref<Persona>   abuelo = persona->padre->padre;
```

- El atributo `padre` es recorrido dos veces para acceder al objeto `abuelo`.
- Esta expresión muestra como se especifica el recorrido a través de varias instancias interrelacionadas para acceder al objeto destino.

Interface a las relaciones

Una relación se representa mediante un objeto y sus operaciones para recorrer las relaciones.



- Cada departamento cuenta con un conjunto de empleados y un conjunto de proyectos.
- Los miembros del departamento trabajan en múltiples proyectos y los proyectos son dirigidos (generalmente) por varios empleados.
- Cada empleado puede tener, como máximo, un "socio" (colega o compañero), que se representa por una relación de la clase `Empleado` consigo misma (relación reflexiva).
- Estas relaciones se pueden representar tanto por mecanismos unidireccionales como bidireccionales. La clase `Departamento` tendrá un conjunto de referencias a objetos `Proyecto`, pero cada `Proyecto` puede no tener una relación hacia `Departamento`. En este caso la relación es unidireccional.
- Cuando se diseña el esquema de datos, es necesario determinar si el recorrido debe realizarse en las dos direcciones.

Relaciones unidireccionales en ODMG

El siguiente código proporciona la declaración de las relaciones unidireccionales para el modelo de objetos anterior.

Cuando las relaciones son unidireccionales, no se requiere un objeto especial para representar la relación; es suficiente utilizar referencias y colecciones:

```
class Departamento : public d_Object {
public:
  d_Set<d_Ref<Empleado>>   empleados;
  d_List<d_Ref<Proyecto>>  proyectos;
};

class Empleado : public d_Object {
public:
  d_Ref<Empleado>          socio;
  d_Ref<Departamento>      depto;
  d_Set<d_Ref<Proyecto>>   proyectos;
};

class Proyecto : public d_Object {
public:
  d_Date      fechaFin() const;
};
```

- La clase `d_Ref<T>` representa las relaciones uno-a-uno.
- Las clases `d_List< d_Ref<T> >` y `d_Set< d_Ref<T> >` representan las relaciones uno-a-muchos desordenadas y ordenadas respectivamente.
- Estas clases, comentadas en los apartados anteriores, se utilizan para representar las relaciones unidireccionales.

Relaciones unidireccionales en ODMG ...

- Obsérvese que la relación entre `Departamento` y `Empleado` presenta recorridos a ambos extremos, pero la relación establecida entre ambas clases sólo se realiza en la dirección `Departamento` hacia `Proyecto`.
- Cada objeto en el modelo que debe ser referenciado por otros objetos debe ser un objeto persistente independiente (no embebido en otro objeto). Por lo tanto, la colección `empleados` de `Departamento` debe contener referencias a instancias `d_Ref<Empleado>` en lugar de contener directamente objetos de la clase `Empleado`.

  Si la declaración de `empleados` hubiese sido:

  ```
  d_Set<Empleado>          empleados;
  ```

  los objetos `Empleado` serían estructuras simples embebidas en la colección. No se trataría de objetos independientes con identidad referenciable por otros objetos de la base de datos.

Objetos relación

El ODMG define clases *template* para representar las relaciones binarias bidireccionales que son mantenidas automáticamente por el software de la base de datos.

- La clase `d_Rel_Ref<T,MT>` representa una relación con una cardinalidad uno-a-uno.
- La clase `d_Rel_Set<T,MT>` representa una relación desordenada y la clase `d_Rel_List<T,MT>` una relación ordenada. La aplicación es responsable de mantener el orden basándose en la posición relativa de los elementos en la lista.

Las clases se muestran en la tabla adjunta con sus correspondientes clases base. Todas las capacidades de la clase base (tales como inserción, borrado e iteración sobre los elementos) están disponibles en la clase relación, aunque en esto caso las operaciones mantienen la integridad referencial.

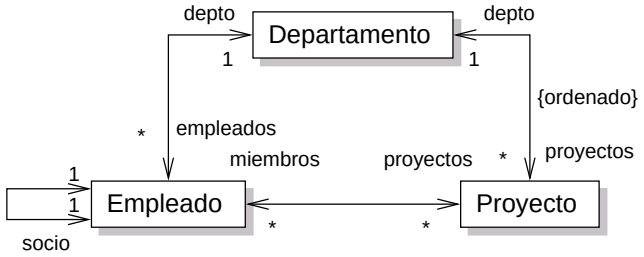
Clase relación	Clase base
`d_Rel_Ref<T, const char *MT>`	`d_Ref<T>`
`d_Rel_Set<T, const char *MT>`	`d_Set< d_Ref<T> >`
`d_Rel_List<T, const char *MT>`	`d_List< d_Ref<T> >`

Estas clase relación se utilizan por parejas; la clase en cada extremo de la relación bidireccional debe contener un miembro que es una instancia de una de esas clases relación. Los dos argumentos del *template* representan:

- `T`: La clase en el otro extemo de la relación.
- `MT`: El nombre de una variable cadena de caracteres que identifica al miembro en `T` que representa el recorrido inverso de la relación.

Objetos relación ...

La mejor forma de explicar cómo se utilizan estas clases es redefinir las relaciones en el ejemplo previo:




```
extern const char _socio[];      // Empleado ⇔ Empleado
extern const char _depto[];      // Empleado ⇔ Departamento
extern const char _emps[];       // Empleado ⇔ Departamento
extern const char _p_d[];        // Departamento ⇔ Proyecto
extern const char _d_p[];        // Departamento ⇔ Proyecto
extern const char _miembros[];   // Empleado ⇔ Proyecto
extern cosnt char _proys[];      // Empleado ⇔ Proyecto

class Departamento : public d_Object {
public:
  d_Rel_Set<Empleado, _depto>   empleados;
  d_Rel_List<Proyecto, _p_d>    proyectos;
};

class Empleado : public d_Object {
public:
  d_Rel_Ref<Empleado, _socio>    socio;
  d_Rel_Ref<Departamento, _emps> depto;
  d_Rel_Set<Proyecto, _miembros> proyectos;
};

class Proyecto : public d_Object {
public:
  d_Rel_Ref<Departamento, _d_p>  depto;
  d_Rel_Set<Empleado, _proys>    miembros;
};
```

Objetos relación ...

- La clase `Departamento` cuenta con el miembro `empleados` que hace referencia a instancias de `Empleado` y la clase `Empleado` tiene una cuenta con el miembro `depto` que hace referencia a instancias de `Departamento`.
- En la declaración de `empleados`, el segundo argumento del *template*, `_depto`, es una cadena de caracteres que contiene el valor `"depto"`. Esta cadena hace referencia al atributo `depto` de `Empleado`, que es el miembro del recorrido inverso.
- De modo similar, en la declaración del miembro `depto` de `Empleado` el segundo argumento del *template*, `_emps`, es una cadena que contiene el valor `"empleados"`.
- La implementación de la base de datos orientada a objetos utiliza la especificación del miembro de la relación inversa para asociar los dos extremos, de modo que puede realizar las operaciones de mantenimiento de las relaciones. Si un extremo de la relación se modifica, puede actualizar automáticamente el otro extremo.

Las variables se inicializan en un fichero adicional:

```
extern const char _socio[] = "socio";        // en Empleado
extern const char _depto[] = "depto";        // en Empleado
extern const char _emps[]  = "empleados";    // en Departamento
extern const char _p_d[]   = "depto";        // en Proyecto
extern const char _d_p[]   = "proyectos";    // en Departamento
extern const char _miembros[] = "miembros"   // en Proyectos
extern const char _proys[] = "proyectos";    // en Empleado
```


Mantenimiento de la integridad referencial

- Los objetos de tipo relación comentados anteriormente llevan a cabo las operaciones de mantenimiento de la integridad referencial.
- Esto reduce significativamente la probabilidad de introducir errores de programación que pueden dar lugar a la pérdida de la integridad referencial.
- Cuando se realiza una modificación en un extremo de la relación, se deben activar determinados *locks* en los dos extremos que se han de modificar.
- Muchas implementaciones también requieren que los objetos se encuentren activos en la cache de la aplicación cuando se realizan las modificaciones.
- Para reducir la actividad de *locking*, en ocasiones es preferible utilizar relaciones unidireccionales y aumentar el grado de concurrencia.

Gestión de relaciones uno-a-uno

Utilizaremos como ejemplo la relación reflexiva `socio` definida en la clase `Empleado`. Supongamos que las siguientes variables de tipo `Empleado` existen en la base de datos:

```
d_Ref<Empleado>    jennifer;
d_Ref<Empleado>    jeremy;
d_Ref<Empleado>    brandon;
```

Las operaciones que podemos realizar son:

- **Establecer una relación:** Supongamos que debemos establecer la relación socios entre `jennifer` y `jeremy`, podemos usar:

  ```
  jennifer->socio = jeremy;
  ```

  o bien:

  ```
  jeremy->socio = jennifer;
  ```

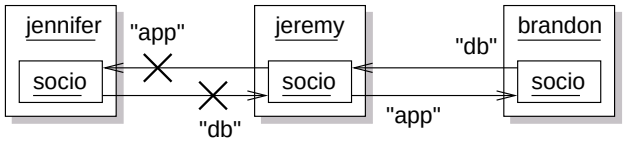
  Sólo es necesario especificar un extremo de la relación, el otro extremo es actualizado automáticamente por el ODBMS.

- **Modificar una relación:** Supongamos ahora que `jeremy` y `brandon` deben se socios. La aplicación deberá ejecutar la siguiente línea de código:

  ```
  jeremy->socio = brandon;
  ```

  Esta línea tiene como resultado las asociaciones mostradas en la figura. El atributo `socio` de `jennifer` es ahora nulo, por lo que se han efectuado de forma automática las dos siguientes operaciones:

  ```
  jennifer->socio = clear();
  brandon->socio = jeremy;
  ```



Gestión de relaciones uno-a-uno ...

- **Eliminar una relación:** El siguiente código pone una referencia nula en la relación `asociado` entre `jennifer` y `jeremy`:

  ```
  jennifer->socio.clear();
  ```

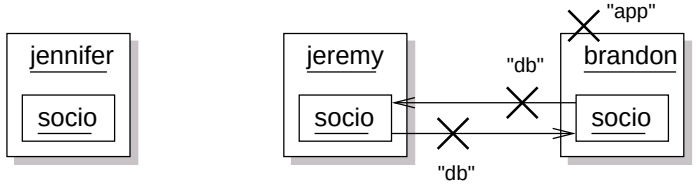
- **Eliminar un objeto:** Supongamos que las relaciones son las mostradas en la figura anterior y se elimina al empleado `brandon` de la base de datos. Cualquier relación bidireccional en la que esté implicado el usuario `brandon` debe actualizarse de modo que no haya referencias al objeto borrado. La aplicación puede ejecutar cualquiera de las dos operaciones siguientes:

  ```
  brandon.delete_object();
  ```

  o bien:

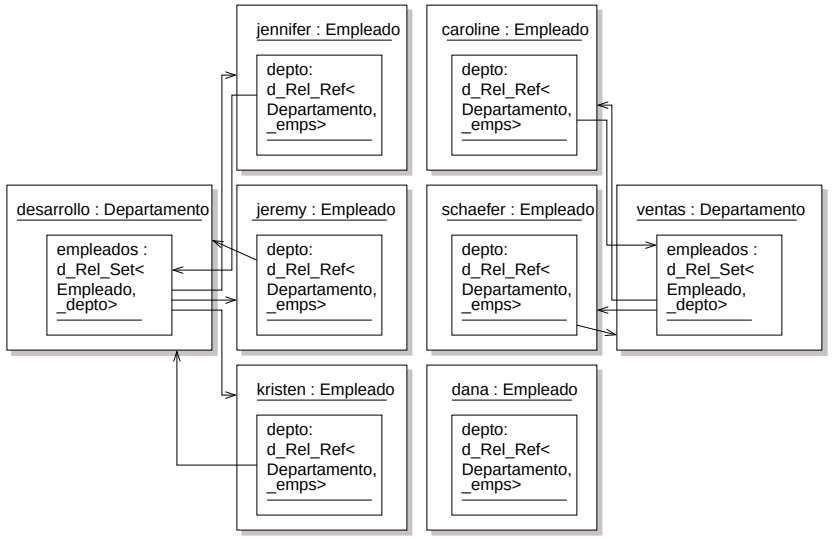
  ```
  Empleado *b = brandon.ptr();
  delete b;
  ```

  para eliminar al `brandon`. El resultado es el mostrado en la siguiente figura, la referencia `socio` en `jeremy` se actualiza automáticamente a null.



Gestión de relaciones uno-a-muchos

- En este caso, uno de los extremos de la asociación es una colección de referencias en lugar de una única referencia.
- La inserción y borrado de los elementos de la colección se lleva a cabo explícitamente por la aplicación o implícitamente por el software de la base de datos.



Supongamos los objetos e interrelaciones mostradas en la figura y supongamos además, que definimos las siguientes variable adicionales:

```
d_Ref<Departamento>    desarrollo, ventas;
d_Ref<Empleado>        dana, schaefer, caroline, kristen;
```


Gestión de relaciones uno-a-muchos ...

Analicemos con detalle cada una de las operaciones que se pueden llevar a cabo:

- **Establecer una relación:** Supongamos que `kristen` es asignada al departamento de ventas:

  ```
  kristen->depto = sales;
  ```

  Tras la ejecución, la referencia `depto` de `kristen` apunta a `ventas` y además `kristen` ha sido añadida implícitamente al conjunto de empleados en la instancia `ventas`. Se podría haber obtenido el mismo resultado añadiendo directamente a `kristen` al departamento `ventas`:

  ```
  ventas->empleados.insert_element(kristen);
  ```

- **Modificar una relación:** Supongamos que es necesario reasignar a `dana` al departamento de ventas

  ```
  dana->depto = ventas;
  ```

  El ODBMS realiza automáticamente los siguientes pasos:

  ```
  desarrollo->empleados.remove_element(dana);
  ventas->empleados.insert_element(dana);
  ```

- **Eliminar una relación:** Para eliminar a `caroline` del departamento de `ventas` se ejecutaría el siguiente código:

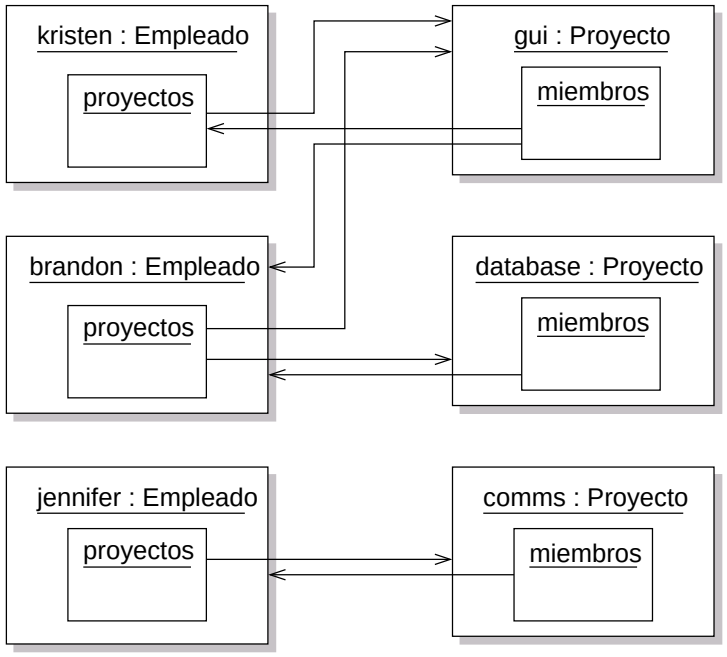
  ```
  ventas->empleados.remove_element(caroline);
  ```

  La referencia `depto` de `caroline` se actualiza automáticamente al valor null.

- **Eliminar un objeto:** Cuando se elimina un objeto, todas las relaciones en las que está implicado deben actualizarse para borrar cualquier referencia.

Gestión de relaciones muchos-a-muchos

Utilizaremos la relación entre `Empleado` y `Proyecto` para ilustrar cómo se manipulan las relaciones muchos-a-muchos. Supondremos que se dan las relaciones mostradas en la figura:



Los objetos de la clase `Proyecto` que utilizaremos se declaran del siguiente modo:

```
d_Ref<Proyecto>  gui, database, comms;
```


Gestión de relaciones muchos-a-muchos ...

Podemos llevar a cabo las siguientes operaciones:

- **Establecer una relación:** Añadimos al empleado `jennifer` al proyecto `database`:

  ```
  database->miembros.insert_element(jennifer);
  ```

  Teniendo en cuenta que es el ODBMS quien mantiene la integridad referencial, el mismo resultado se podría haber obtenido mediante:

  ```
  jennifer->proyectos.insert_element(database);
  ```

- **Modificar una relación:** La inserción de elementos en un conjunto del tipo `d_Rel_List` es un proceso algo más sofisticado, ya que permite actualizar en posiciones concretas de la lista.
- **Eliminar una relación:** Supongamos que eliminamos al usuario `brandon` del proyecto `database` ejecutando la siguiente sentencia:

  ```
  database->miembros.remove_element(brandon);
  ```

  La referencia a `brandon` se elimina explícitamente de la colección `miembros` del proyecto `database`. Implícitamente, se elimina la referencia al proyecto `database` de la colección `proyectos` de `brandon`.
- **Eliminar objeto:** El efecto de eliminar un objeto en una relación muchos-a-muchos es similar al de eliminarlo en una relación uno-a-muchos.
  Supongamos que se completa el desarrollo de la interface de usuario, y debemos eliminar el objeto referenciado por `gui`:

  ```
  gui.delete_object();
  ```

  Ya que tanto `kriten` como `brandon` contienen referencias a `gui`, sus atributos `proyectos` se actualizan automáticamente.

Recorrido y expresiones de recorrido

- El recorrido de las relaciones permite a una aplicación navegar entre los objetos relacionados en la base de datos.
- La optimización de las referencias y las colecciones es uno de los puntos en los que mayor esfuerzo se ha realizado en las bases de datos orientadas a objetos.
- Por este motivo, las bases de datos orientadas a objetos llevan a cabo estas actividades de forma mucho más eficiente que las bases de datos relacionales (en las que se deben establecer las asociaciones dinámicamente en tiempo de ejecución mediante *joins*).

Una expresión que navega a través de varios objetos se denomina "expresión de recorrido". Supongamos que dado un empleado, deseamos imprimir información acerca de los proyectos en los que está trabajando su "socio":

```
extern ostream &operator<<(ostream &, const Proyecto &);

d_Ref<Empleado>       emp;
d_Iterator<Proyecto> proys;
d_Ref<Proyecto>       proy;

proys = emp->socio->proyectos.create_iterator();
while ( proys.next(proy) ) {
  cout << *proy << endl;
}
```

En donde hemos supuesto que se ha definido un operador de impresión para la clase `Proyectos`.

Recorrido y expresiones de recorrido ...

Alternativamente, el código anterior se podría haber escrito utilizando la sintáxis STL. Supongamos que se ha definido la función:

```
void printProyecto(d_Ref<Proyecto> proy)
{
  cout << *proy << endl;
}
```

El siguiente código STL llevaría a cabo la misma función:

```
d_Ref<Empleado> o_socio = emp->socio;
for_each(o_socio->proyectos.begin(),
         o_socio->proyectos.end(),
         printProyecto);
```

Objetos compuestos

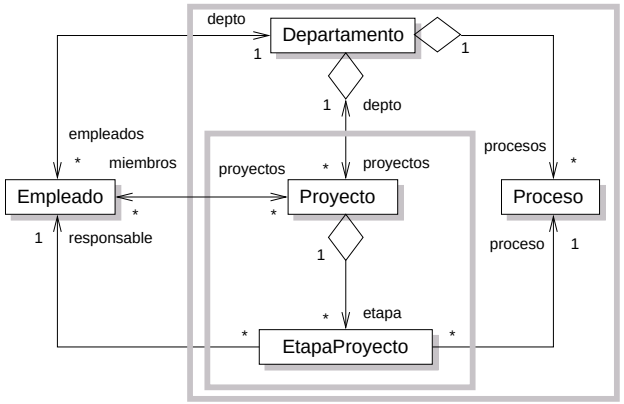
- Los modelos de objetos contienen en ocasiones objetos cuyas características no se pueden representar mediante atributos simples embebidos en una instancia. Esos componentes se representan de forma más adecuada mediante objetos separados.
- Por supuesto, si el número de objetos requeridos es fijo se pueden embeber en el objeto matriz, pero este no es generalmente el caso.
- Estos objetos independientes se consideran componentes del objeto aun cuando no son atributos embebidos.
- El objeto que los soporta se denomina "objeto compuesto" y se dice que está formado por "subobjetos". En ocasiones también se emplea eel nombre de "objetos compuestos" para describirlos.
- Otro de los motivos por el que los subobjetos se representan mejor mediante objetos separados está en la posibilidad de establecer relaciones entre ellos.
- El objeto compuesto es el contexto en el que existen los subobjetos y sus relaciones.
- El "valor" del objeto compuesto se basa en sus atributos junto con los valores de sus subobjetos y sus atributos y relaciones.

Ejemplos de objetos compuestos

En el ejemplo de departamentos, empleados y proyectos utilizado hasta ahora, cualquiera de esos objetos puede ser un objeto compuesto.

- La clase `Departamento` contiene un conjunto de proyectos y a su vez un conjunto de procesos de desarrollo que deben completarse para cada proyecto.
- La clase `Departamento` también cuenta con un conjunto de objetos `Empleado`. Es más lógico suponer que los empleados son miembros de la compañía como un todo. Si el departamento se disuelve, los empleados son reasignados.
- Las instancias `Proyecto` asociadas a un departamento pueden considerarse subobjetos de su instancia `Departamento`.
- La clase `Proyecto` puede ser a su vez un objeto compuesto, cuyos subobjetos representan las etapas requeridas para completar el proyecto. Cada etapa puede tener un "dueño" y una fecha de conclusión.

En la figura se muestran las relaciones entre las clases descritas:



Ejemplos de objetos compuestos ...

Si ambos, `Departamento` y `Proyecto` son objetos compuestos, las relaciones entre estas clases pueden representarse mediante las siguientes declaraciones de clase:

```
extern const char _depto[];      // Empleado ⇔ Departamento
extern const char _emps[];       // Empleado ⇔ Departamento
extern const char _p_d[];        // Departamento ⇔ Proyecto
extern const char _d_p[];        // Departamento ⇔ Proyecto
extern const char _miembros[];   // Empleado ⇔ Proyecto
extern cosnt char _proys[];      // Empleado ⇔ Proyecto

class Departamento : public d_Object {
  d_Rel_List<Proyecto,_p_d>     proyectos;
  d_Rel_Set<Empleados,_depto>   empleados;
  d_List<d_Ref<Proceso>>        procesos;
  // Otros miembros
};

class Empleado : public d_Object {
  d_Rel_Ref<Departamento, _emps>    depto;
  d_Rel_Set<Proyecto, _miembros>    proyectos;
};

extern const char _depto[]     = "depto";      // en Empleado
extern const char _emps[]      = "empleados"; // en Departamento
extern const char _p_d[]       = "depto";      // en Proyecto
extern const char _d_p[]       = "proyectos"; // en Departamento
extern const char _miembros[] = "miembros"    // en Proyectos
extern const char _proys[]     = "proyectos"; // en Empleado
```


Ejemplos de objetos compuestos ...

- Cada departamento cuenta con un conjunto de las etapas que se deben realizar para cada proyecto.
- Este conjunto puede ser diferente para cada departamento, de modo que cada uno de ellos cuenta con su propia lista.
- La clase `Proceso` contiene información general asociada a un proceso que no es específica de un proyecto particular.

```
class Proceso : public d_Object {
  d_String      nombre;
  d_String      descripcion;
  // etc...
};
```

La aplicación no necesita recorrer desde una instancia `Proceso` a cada una de las instancias `EtapaProyecto` asociada con ella, por lo que no se ha previsto la expresión de recorrido (*traversal path*).

Ejemplos de objetos compuestos ...

- La clase `EtapaProyecto` está constituida por miembros denominados `proceso` que hacen referencia a una instancia `Proceso`. Cada proyecto debe mantener la información acerca de sus etapas.
- Cada instancia `Proyecto` cuenta con instancias `EtapaProyecto` para cada instancia `Proceso` referenciada en el atributo `procesos` de la clase `Departamento`.

```
class Proyecto : public d_Object {
  d_Rel_Ref<Departamento, _d_p>      depto;
  d_Rel_Set<Empleado, _proys>        miembros;
  d_List<d_Ref<EtapaProyecto>>       etapa;
  // Otros atributos
};

class EtapaProyecto : public d_Object {
  d_Ref<Proceso>      proceso;
  d_Date              fechaPrevista;
  d_Date              fechaConclusion;
  d_Ref<Empleado>     responsable;
  // Otros miembros
};
```

- La clase `Proyecto` se considera un objeto compuesto ya que gestiona un conjunto de instancias `EtapaProyecto`.
- La aplicación accede a las instancias de `EtapaProyecto` sólo a través del miembro `etapa` de `Proyecto`.

Subobjetos

- Como se ha dicho, un subobjeto actúa como componente de un objeto compuesto.
- Un objeto embebido como miembro de otra clase no se considera un subobjeto.
- Un subobjeto puede ser un objeto simple u otro objeto compuesto, por lo que es posible construir objetos compuestos organizados en una jerarquía de complejidad arbitraria.
- Un objeto se considera un subobjeto si su existencia depende su objeto compuesto base; es decir, no puede existir sin su objeto compuesto.
- Cuando una aplicación elimina un objeto compuesto, se eliminan tanto el objeto base como todos sus subobjetos.
- El destructor del objeto compuesto es responsable de propagar la operación de borrado a todos los subobjetos.

Referencias a objetos que no son subobjetos

- Un objeto que contiene referencias (`d_Ref<T>`) o una colección de referencias a otros objetos de tipo `T` no es necesariamente un objeto compuesto.
- De modo similar, un objeto compuesto puede contener además referencias a objetos que no son sus subobjetos. Estos objetos referenciados no se consideran componentes del objeto compuesto y no están afectados por las operaciones de borrado del objeto compuesto.

Referencias a objetos que no son subobjetos ...

La aplicación que hemos estado usando presenta varios ejemplos de esto:

- La clase `EtapaProyecto` contiene un miembro `proceso` que es una referencia a `Proceso` y un miembro `responsable` que referencia a una instancia de `Empleado`. Ninguno de estos miembros se pueden considerar subobjetos de la instancia `EtapaProyecto`.
- De modo similar, la clase `Departamento` contiene una colección de referencias a `Empleado` denominada `empleados`. En este modelo, las instancias de `Empleado` no son subobjetos de `Departamento`, aún cuando `Departamento` es un objeto compuesto que cuenta con subobjetos para la gestión de los proyectos y los procesos.

Objeto compuesto base

- Las operaciones que son específicas del objeto compuesto se definen normalmente dentro de la clase del objeto compuesto base. Su implementación requiere generalmente la manipulación de subobjetos.
- El objeto compuesto base debe aparecer como una entidad atómica.
- Uno de los objetivos en la definición de un objeto compuesto es obtener un alto grado de atomicidad aún cuando el objeto está compuesto por una agregación de otros objetos.

Operaciones

Las operaciones que se aplican sobre un objeto compuesto generalmente implican operar sobre sus subobjetos.

- Este fenómeno se denomina "propagación de operaciones".
- Más aún, algunas operaciones sobre objetos compuestos deben necesariamente propagarse a todos sus subobjetos (por ejemplo, el borrado).
- La semántica de las operaciones comunes deben definirse en la clase base.
- El diseñador de la clase debe decidir si estas operaciones deben ser superficiales o en profundidad. Las operaciones canónicas de C++ (copia, borrado y asignación) suelen ser operaciones en profundidad.
- Un subobjeto es independiente del objeto compuesto base que lo contiene. Por tanto, cuando una operación de copia se aplica al objeto compuesto base, la operación debe crear una copia de cada subobjeto.

  Supongamos dos instancias `A` y `B` de un objeto compuesto base. Una asignación de `B` a `A` debe borrar todos los objetos actuales de `A` y realizar una copia de todos los subobjetos de `B`.

  Normalmente se requieren operaciones en profundidad para llevar a cabo esta tarea.
- Por definición, el borrado de un objeto compuesto debe propagarse a todos sus subobjetos, aunque es importante recordar que un objeto compuesto puede contener referencias a objetos que no deben ser borrados. Es responsabilidad de la aplicación gestionar adecuadamente esta situación.

Operaciones ...

En la aplicación que nos sirve de ejemplo, las clases `Departamento` y `Proyecto` son objetos compuestos.

Sus destructores proporcionan las operaciones adecuadas para manejar sus subobjetos:

```
Proyecto::~Proyecto()
{
  d_Iterator<d_Ref<EtapaProyecto>> iter;
  d_Ref<EtapaProyecto> ep;

  iter = EtapaProyecto.create_iterator();
  while ( iter.next(ep) )
     ep.delete_object();
}

Departamento::~Departament()
{
  d_Iterator<d_Ref<Proceso>>   iter_proceso;
  d_Iterator<d_Ref<Proyecto>>  iter_proyecto;
  d_Ref<Proceso>               proc;
  d_Ref<Proyecto>              proy;

  iter_proceso = procesos.create_iterator();
  while ( iter_proceso.next(proc) )
     proc.delete_object();

  iter_proyecto = proyectos.create_iterator();
  while ( iter_proyecto.next(proy) )
     proy.delete_object();
}
```


Operaciones ...

Consideraciones de *locking*

- Algunas bases de datos permiten la propagación de bloqueos, aunque no está soportado en el estándar ODMG.
- El diseñador debe determinar cuándo el bloqueo de un objeto compuesto implica también el bloqueo de sus subobjetos.
- La base de datos determina automáticamente que cuando un determinado objeto adquiere un bloqueo, debe propagar esta acción a todos los objetos relacionados.
- Un objeto bloqueado indirectamente puede a su vez propagar el bloqueo a otros objetos, alcanzando a un gran número de objetos y limitando la concurrencia.
- Otra posibilidad consiste en establecer una política de bloqueo cuando se accede a un objeto compuesto.

  Por ejemplo, podría imponerse el requisito de que una aplicación debe bloquear el objeto compuesto antes de acceder a todos sus subobjetos (bloqueando implícitamente todos los subobjetos).