6. CREACIÓN Y DESTRUCCIÓN DE OBJETOS

Con el tema anterior se completó el marco básico de los elementos que constituyen el armazón del juego del Pang, así como sus interacciones básicas. En este capítulo se desarrolla la gestión de conjuntos de objetos, principalmente el conjunto de esferas, a través de algunos mecanismos de la POO.

En el código suministrado para el desarrollo de este capítulo se ha incluido el método estático `Interaccion::rebote(Esfera& e1, Esfera& e2)`, cuya finalidad es la de calcular el rebote entre dos esferas simulando el comportamiento físico real. Su complejidad más que en la codificación se encuentra en la física y geometría implicada en el cálculo de las velocidades resultantes tras el choque.

Para demostrar su uso, se ha incluido una llamada a dicha función, de forma que se observa el rebote entre las dos esferas que hasta el momento constituían la escena. Esta llamada está situada en el cuerpo de la función `Mueve()` de la clase `Mundo`:

```
void Mundo::Mueve()
{
    hombre.mueve(0.025f);
    esfera.mueve(0.025f);
    esfera2.mueve(0.025f);
    bonus.mueve(0.025f);
    disparo.mueve(0.025f);

    Interaccion::rebote(hombre,caja);
    Interaccion::rebote(esfera,caja);
    Interaccion::rebote(esfera,plataforma);
    Interaccion::rebote(esfera2,caja);
    Interaccion::rebote(esfera2,plataforma);
    Interaccion::rebote(esfera,esfera2);
}
```

Si se ejecutase el juego ya terminado, se observaría que las esferas son objetos que varían en su cantidad de forma continuada. A veces se tienen dos, tres o cuatro esferas. No importa cuántas tengamos que cada una de ellas rebotará contra las paredes y si son impactadas por un disparo se duplican, y las esferas resultantes vuelven a tener el comportamiento programado para una esfera individual. Se muestra a continuación, el código que haría falta agregar si en vez de dos esferas tuviésemos cuatro:

```
void Mundo::mueve()
{
```

```
    hombre.mueve(0.025f);
    bonus.mueve(0.025f);
    disparo.mueve(0.025f);
    //mover esferas
    esfera.mueve(0.025f);
    esfera2.mueve(0.025f);
    esfera3.mueve(0.025f);
    esfera4.mueve(0.025f);

    //chocar esfera con la caja
    Interaccion::rebote(esfera,caja);
    Interaccion::rebote(esfera2,caja);
    Interaccion::rebote(esfera3,caja);
    Interaccion::rebote(esfera4,caja);

    //chocar esfera con la plataforma
    Interaccion::rebote(esfera,plataforma);
    Interaccion::rebote(esfera2,plataforma);
    Interaccion::rebote(esfera3,plataforma);
    Interaccion::rebote(esfera4,plataforma);

    //choque de esferas entre sí
    Interaccion::rebote(esfera,esfera2);
    Interaccion::rebote(esfera,esfera3);
    Interaccion::rebote(esfera,esfera4);
    Interaccion::rebote(esfera2,esfera3);
    Interaccion::rebote(esfera2,esfera4);
    Interaccion::rebote(esfera3,esfera4);

    Interaccion::rebote(hombre,caja);
}
```

Los comentarios indican la intención del programador, pero el código se va extendiendo de forma progresiva, de una forma que obviamente no es generalizable para cualquier número de esferas. Se observa un patrón en el código que lo que manifiesta es que hay una serie de operaciones que deben aplicarse a cada esfera del conjunto de esferas. ¿Podría crearse algún tipo de objeto que las agrupase, de forma que se pudiera decir: detectar choque de cualquier esfera contra la caja?

6.1. CREANDO LA CLASE LISTAESFERAS

Observamos que hay un elemento especialmente dinámico en cuanto a creación, destrucción, número e iteraciones, dentro del programa. Este es el caso de la `Esfera`. De alguna manera, el programa tiene que trabajar con un número continuamente variable de esferas, ya que estas se duplican y se destruyen continuamente, y mientras tanto hay que estar comprobando si son impactadas por el disparo, si se chocan entre ellas o si chocan contra las paredes.

Por ello parece útil el diseño de una clase que contenga las esferas y que se preocupe de gestionar su aparición y desaparición, así como las operaciones como pintar o mover que se realizan en todas ellas.

A esta clase la denominaremos `ListaEsferas` y deberá suministrar la siguiente funcionalidad:

- Gestionará una lista de esferas, permitiendo agregar, quitar o eliminar esferas creadas externamente.
- Nos dará un acceso elegante a cada una de las esferas.
- Realizara automáticamente operaciones comunes a todo el conjunto de las esferas tales como Pintar, Mover o rebotar contra una pared, o contra la caja.
- Realizará operaciones que supongan la interacción exclusiva de los objetos contenidos entre sí. Es decir, gestionará el rebote entre las esferas.

Nótese que según esta funcionalidad, la clase *contenedora* que estamos diseñando, en un principio no crea ni destruye los objetos, sino que los recibe y los agrupa. Es posible, sin cometer un error de diseño, dar a la clase la responsabilidad de producir o destruir objetos, siempre que estos sean consecuencia directa de una petición desde el exterior. También es importante notar que al contrario que el resto de clases, cuyos conceptos aparecen en singular (`Esfera, Disparo`), la clase `ListaEsferas` parece plural, pero no lo es. El concepto es también singular (podemos tener una lista de esferas)

Aunque se podría plantear el crear una clase que pudiera contener todas las esferas que quisiéramos, por simplificar de momento el código se va a limitar su número a un máximo fijo.

Por tanto, mediante el botón derecho sobre el icono que representa al proyecto accedemos a *Agregar->Clase…* y creamos la clase genérica `ListaEsferas`, acordándonos de guardar los ficheros en la subcarpeta "src":

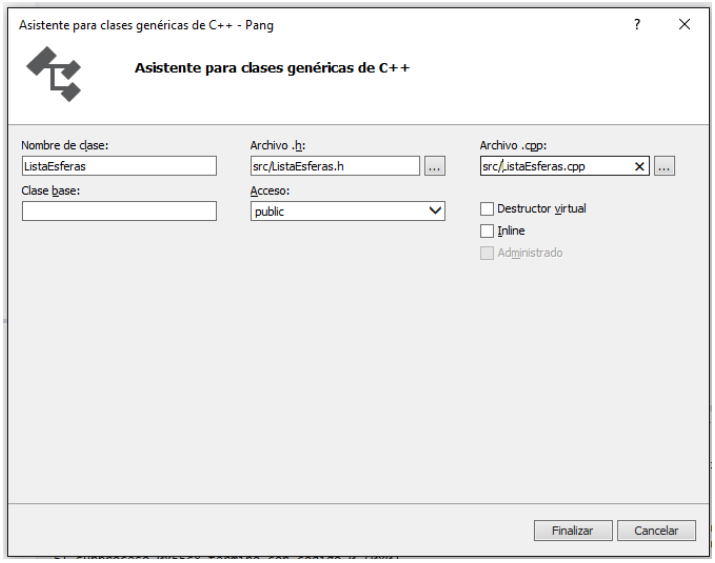

Figura 6-1. Adición de la clase ListaEsferas

Puesto que las esferas serán creadas externamente a la clase, lo que va a hacer esta es mantener una lista con los sitios de la memoria en donde se encuentran cada una de las distintas esferas que debe manejar. Esto se implementará mediante la inclusión de dos atributos **privados**:

```
class ListaEsferas
{
public:
    ListaEsferas();
```

```
    virtual ~ListaEsferas();
private:
    Esfera * lista[MAX_ESFERAS];
    int numero;
};
```

El primero es un vector de punteros a objetos de tipo `Esfera`. Es decir que tendremos la posibilidad de almacenar hasta un número `MAX_ESFERAS` de posiciones de memoria en las que se encuentra una esfera. El segundo atributo, es necesario para llevar cuenta de cuantas esferas llevamos apuntadas. De esta forma es posible recorrer sólo los elementos del vector que tienen direcciones válidas, porque se han rellenado con posiciones de memoria de objetos ya creados.

El valor de `MAX_ESFERAS`, se puede definir al comienzo del fichero de cabecera de la clase, asignando un valor de `100` por ejemplo:

```
#define MAX_ESFERAS 100
```

En el caso de los contenedores, es especialmente importante proteger los atributos que llevan cuenta de los objetos. Esto es así porque si fueran públicos sería posible modificar la cuenta de objetos que se tienen almacenados desde el exterior de la clase con el problema de funcionamiento que esto puede generar.

Conceptualmente, la clase `ListaEsferas` es una agregación de esferas (nótese la diferente representación respecto de la relación de composición), lo que se puede representar en un diagrama UML (en el diagrama de clases de diseño DCD) de nuestra aplicación como:

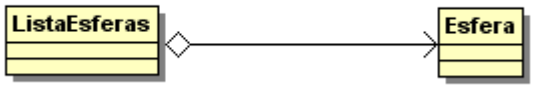

Figura 6-2. La clase ListaEsferas como agregación de Esfera

Una vez preparados los atributos y el armazón de la clase se seguirán los siguientes pasos:

6.1.1. Inicialización de la lista

En primer lugar se asegurará que al principio la información contenida es ninguna, y por tanto, al crear una instancia de la clase `ListaEsferas`, es necesario dar valor cero al atributo `numero`. De igual forma, es conveniente indicar que ningún puntero apunta de momento a nada, puesto que no se ha agregado ninguna esfera al conjunto. Esto se hace poniendo a cero el valor de cada uno de los 100 punteros. Para codificar la inicialización, lógicamente, rellenaremos el código del constructor por defecto, que el compilador ya ha preparado al generar la clase:

```
ListaEsferas::ListaEsferas()
{
    numero=0;
    for(int i=0;i<MAX_ESFERAS;i++)
          lista[i]=0;
}
```

6.1.2. Adición de esferas

Una vez inicializados los atributos de la clase se va a dotar de la función que nos permita añadir esferas al conjunto. Las operaciones que se deben realizar son las siguientes. Es posible que se llegue a la situación de que no quepan más esferas en el contenedor porque se haya alcanzado su capacidad máxima. Es ese caso es conveniente que la función informe de que la esfera no ha podido ser agregada. Según lo descrito el prototipo del método deberá adoptar la siguiente forma:

```
bool agregar (Esfera *e);
```

De forma que si se agrega la esfera, el método retornará el valor `true`, y devolverá `false` en caso contrario. Lógicamente, puesto que lo que se va a almacenar son direcciones, lo que recibe la función será también una dirección.

Una vez comprobado que se puede agregar una esfera, se deben realizar los pasos que se describen a continuación. En primer lugar se almacena la dirección en el último puesto del vector sin rellenar verificando que no se ha superado la capacidad máxima del contenedor. En segundo lugar se indica que el número de esferas apuntadas por el contenedor se ha incrementado en uno. Tanto en C como en C++ esto se suele realizar en una sola sentencia aprovechando el modo de funcionamiento del operador post incremento. Por tanto, el código que se ha de implementar es el siguiente:

```
bool ListaEsferas::agregar (Esfera *e)
{
    if(numero<MAX_ESFERAS)
          lista[numero++]=e;
    else
          return false;
    return true;
}
```

6.1.3. Dibujo y movimiento de las esferas

Ahora la clase `ListaEsferas` ya puede recibir objetos de tipo esfera. Recuérdese que la función principal de esta clase es la de agrupar operaciones, de forma que con una sola instrucción se pueda hacer que todas las esferas contenidas, se pinten o se muevan Estas dos funciones son las que se implementarán, y básicamente consistirá en recorrer las esferas e ir diciéndole a cada una que ejecute su método correspondiente. El código de las mismas es el que se pone a continuación, que como se ve es sencillo.

```
void ListaEsferas::dibuja()
{
    for(int i=0;i<numero;i++)
          lista[i]->dibuja();
}

void ListaEsferas::mueve(float t)
{
    for(int i=0;i<numero;i++)
          lista[i]->mueve(t);
}
```

Puesto que lo que se almacenan son direcciones, para poder acceder a la ejecución de un método de la esfera apuntada es necesario recurrir al operador `->`. El código, lo

único que hace es recorrer las `numero` direcciones de esferas apuntadas, e ir ejecutando sus métodos `dibuja` y `mueve` respectivamente. Como para mover una esfera es necesario pasar el parámetro del intervalo de tiempo, la función mover de `ListaEsferas`, también requerirá de dicho argumento.

6.2. USANDO LA CLASE LISTAESFERAS

Con esto ya tenemos la funcionalidad mínima para poder probar el funcionamiento de lo que se lleva programado. Agregamos un atributo de tipo `ListaEsferas` a la clase `Mundo`, al cual vamos a llamar `esferas`.

Obviamente, el `include` correspondiente a la clase `ListaEsferas` es necesario en el fichero *Mundo.h.*

```
class Mundo
{
    …
private:
    ListaEsferas esferas;
    …
};
```

Agregamos la llamada a las funciones `Dibujar` y `Mueve` de esferas en las funciones respectivas de la clase mundo, de igual forma a como se procedió cuando añadimos el resto de objetos de la escena:

```
void Mundo::dibuja()
{
    …
    esferas.dibuja();
}

void Mundo::mueve()
{
    …
    esferas.mueve(0.025f);
    …
}
```

Evidentemente, si ejecutamos el código, no se observa ningún cambio, puesto que el contenedor está vacío de objetos. Finalmente para probarlo, vamos a agregar unas cuantas esferas al contenedor, y observamos lo que ocurre. Para ello, en la inicialización del mundo, creamos una serie de esferas -en concreto seis- y las introducimos en `esferas`:

```
void Mundo::inicializa()
{
    …

    for(int i=0;i<6;i++)
    {
          Esfera* aux=new Esfera;
          aux->setPos(i,1+i);
```


```
            aux->setVel(i,i);
            aux->setRadio(0.75+i*0.25);
            esferas.agregar(aux);
        }
}
```

6.2.1. Sobrecarga de constructores

En general, cada vez que queremos crear una nueva esfera, será necesario indicar una serie de atributos básicos que definen el objeto. Algunos son claramente auxiliares, como puede ser el color, pero otros casi siempre hemos de indicarlos tras haber creado el objeto.

En el caso anterior se observa claramente como es conveniente especificar el radio, la posición y la velocidad, para diferenciar las distintas esferas que se han creado.

Se podría compactar mucho más el código si permitimos definir estos atributos en el momento de creación del objeto. Esto se puede hacer gracias a la sobrecarga del constructor, que en C++ se permite.

Por ello, a continuación, se va a implementar un nuevo constructor para la clase `Esfera` que espera la inclusión de parámetros como el radio y la posición, y cuyo prototipo sería:

```
Esfera(float rad, float x=0.0f, float y=0.0f,
    float vx=0.0f, float vy=0.0f);
```

En donde indicamos que tanto la posición como la velocidad asumirán por defecto el valor nulo si no se utilizan cuando se invoque al constructor. Al igual que cualquier otra función, los constructores admiten la definición de valores por defecto. Los argumentos asumirán el valor indicado en caso de que el programador no los defina. Por el modo de proceder de este mecanismo, lo normal es poner como primeros argumentos aquellos que tengan más posibilidades de ser definidos explícitamente por el programador, y después por orden decreciente de importancia, los que puedan ser asumidos por defecto.

Esto nos permitiría construir esferas de maneras muy diversas. Los siguientes, son ejemplos de las distintas sentencias válidas:

```
Esfera miesfera;                    //constructor por defecto…sin
argumentos
Esfera miesfera1(8.0F);     //nuevo constructor: radio 8
Esfera miesfera2(3.0F,2,5);             //nuevo constructor: radio 3  y
                                    //posición 2,5
Esfera miesfera3(2.0F,i,i*2,i+4,3);
Esfera *aux=new Esfera(5.0f);     //creación dinámica con el nuevo
                                    //constructor
```

La implementación del constructor quedaría como sigue:

```
Esfera::Esfera(float rad, float x, float y, float vx, float vy)
{
    radio=rad;
    posicion.x=x;
    posicion.y=y;
    velocidad.x=vx;
    velocidad.y=vy;

    rojo=verde=255;
```

```
    azul=100; //color distinto
    aceleracion.y=-9.8;
}
```

Rescribimos ahora el código de creación de las esferas utilizando este nuevo constructor:

```
    for(int i=0;i<6;i++)
    {
          Esfera* aux=new Esfera(0.75+i*0.25,i,1+i,i,i);
          esferas.Agregar(aux);
    }
```

6.3. REBOTES

Cuando ejecutamos el programa, nos damos cuenta que las esferas recién creadas se van de la pantalla. Esto se debe obviamente a que no estamos invocando las funciones de rebote correspondientes para cada una de las esferas de la clase `ListaEsferas`. Vamos a programar en esta sección esta funcionalidad.

6.3.1. Rebote con la caja

Lo primero que deseamos es que las esferas no salgan de la caja que define el área de juego. Es decir tenemos que implementar la funcionalidad de rebote de la lista de esferas y la caja. Para ello podríamos seguir diferentes alternativas. Por ejemplo podríamos decidir implementar un método en la clase `Caja`, que admita un parámetro de la clase `ListaEsferas` por referencia, para poder modificarlo si fuera necesario:

```
class Caja  //UNA POSIBLE OPCION
{
    friend class Interaccion;
public:
    Caja();
    virtual ~Caja();
    void dibuja();
    void rebote(ListaEsferas& lista_esferas);
};
```

No obstante esta alternativa violaría la encapsulación de `ListaEsferas`, al no poder acceder fácilmente a sus miembros privados. También podríamos intentar añadir un método a la clase `Interaccion`, aunque a priori también nos encontraríamos con el mismo problema:

```
class Interaccion  //OTRA POSIBLE OPCION
{
public:
    static bool rebote(ListaEsferas& lista, Caja c);
};
```

Además, esta última opción rompe ligeramente el criterio adoptado cuando se desarrolló la clase `Interaccion`, que es que dicha clase se encarga de simular o calcular las interacciones físicas entre pares de objetos. Lo importante cuando se diseña

es establecer criterios y pautas. El criterio seguido en nuestro caso es el siguiente:

> La clase `ListaEsferas` es la responsable de agrupar un conjunto de esferas y de repetir para cada una de ellas tareas individuales, como que se pinte o que se mueva cada esfera. Parece por tanto lógico que la clase `ListaEsferas` sea la encargada de repetir las acciones de rebotes para cada una de sus esferas.

No queremos decir con ello que este sea el único criterio valido. De hecho se propone en el anexo un diseño más avanzado y arquitectónicamente más correcto. No obstante se mantiene el criterio anterior para los siguientes desarrollos por simplicidad y a título didáctico para el programador novel.

Construimos pues a continuación un método que compruebe el rebote de cada una de las esferas contenidas con una caja que se pasa por parámetro. El código resultante, es muy parecido al del apartado anterior, pero en este caso, hacemos uso de la función de la clase `Interacción` que nos permite calcular el rebote entre una caja y una esfera:

```
#include "Interaccion.h"
…
void ListaEsferas::rebote(Caja caja)
{
    for(int i=0;i<numero;i++)
          Interaccion::rebote(*(lista[i]),caja);
}
```

Obviamente, es necesario invocar la función:

```
void Mundo::mueve()
{
    …
    esferas.rebote(caja);
    …
}
```

Obtendremos el resultado siguiente:

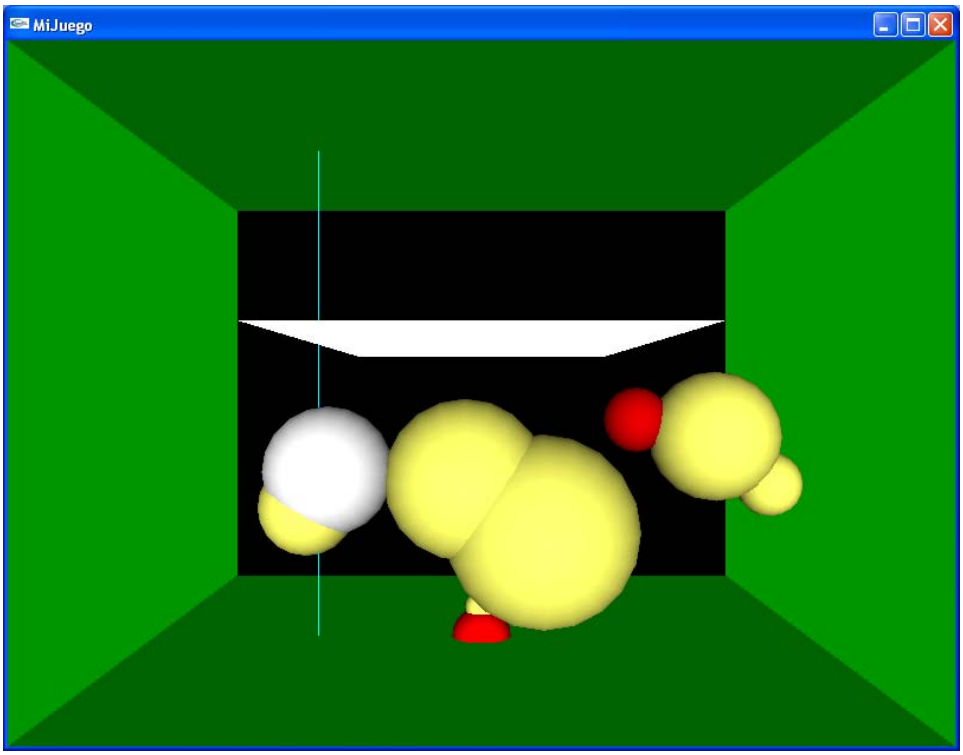

Figura 6-3. La clase ListaEsferas en funcionamiento

De nuevo, aunque la estructura es igual a los casos anteriores se observa una nueva diferencia. La función `Interaccion::rebote(Esfera& e, Caja c)`, espera como argumentos una variable de tipo `Esfera` y otra variable de tipo `Caja`. En cuanto a la caja no hay ninguna dificultad, puesto que se recibe directamente como argumento de la función, sin embargo lo que se ha almacenado en la clase contenedora no son objetos de tipo `Esfera`, sino sus direcciones. Por ello, si se quiere acceder al objeto cuando lo que se tiene es la dirección es necesario hacer uso del operador contenido (*), tal y como se refleja en `*(lista[i])`.

6.3.2. Sobrecarga de los rebotes

Al igual que se ha sobrecargado un constructor, es posible sobrecargar otros métodos. Por ejemplo, en el caso de la lista de esferas, se observa, que interesa que se gestione no sólo el rebote con la caja, sino también el rebote de las esferas con cualquier pared, y de las esferas entre sí. Esto se puede realizar utilizando el mismo identificador para el método, pero cambiando el tipo de argumento que se utilizará.

Ahora procedemos a definir el método `void ListaEsferas::rebote(Pared p)` cuya función es la de hacer que todas las esferas contenidas en la lista, reboten contra la pared que se pasa como argumento:

```
void ListaEsferas::rebote(Pared p)
{
    for(int i=0;i<numero;i++)
          Interaccion::rebote(*(lista[i]),p);
}
```

Para que comience a funcionar, será necesario incluir la llamada a este método en la función `mueve()` de la clase `Mundo`. Aunque aparentemente la llamada a la función es la misma, el compilador diferencia el código que se debe ejecutar en función del tipo de parámetro que se pasa. Esto hace que el código sea muy legible, y facilita la escritura de código por parte del programador.

Más interesante, por ser un poco distinto, es plantearse cómo hacer el código de la función rebotar que gestiona el rebote de las distintas esferas entre sí. Puesto que es la clase contenedora la que contiene toda la información necesaria para este cálculo, deberá ser una función que no recibe ningún argumento.

EJERCICIO: Programar el rebote de las esferas entre sí, mediante un doble bucle for() anidado

Codificar el método de la clase `ListaEsferas`, cuyo prototipo sea el siguiente, y que gestione el rebote entre las esferas contenidas:

```
void ListaEsferas::rebote()
```

De esta forma el código de la función `mueve()` finalmente contenido en `Mundo` y que gestiona las interacciones de las esferas contenidas en el contenedor es el siguiente:

```
   esferas.rebote();
   esferas.rebote(caja);
   esferas.rebote(plataforma);
```

Al ejecutar el código se observa que hay dos conjuntos de esferas que no interaccionan entre sí, aunque sí que lo hacen con el resto de objetos de la escena. En el fondo eso es lo que dice el programa escrito, dado que tratamos de forma independiente a `esfera1` y `esfera2`, y por otro lado a las esferas contenidas en la lista de esferas.

6.4. EL DESTRUCTOR Y EL ENCAPSULAMIENTO

A estas alturas del libro, el lector ya habrá observado que el modo habitual de proceder en POO es tal que las clases suelen tener como parte privada –y por tanto no accesible desde el exterior- prácticamente todos los atributos de la clase, mientras que lo más normal es que los métodos sean públicos. Esta forma de implementación intenta compartimentar el código de forma que podemos asegurar que un objeto funciona por si mismo, incluso si se le piden cosas erróneas. Por ejemplo, la clase `ListaEsferas` tal y como está actualmente diseñada, impide que en una lista de una capacidad determinada, se introduzcan más objetos de los posibles.

Un aspecto importante de este modo de trabajar con objetos cerrados y protegidos es el control de su creación y su destrucción, por medio de los constructores y destructores. Gracias al constructor nos ha sido posible definir que inicialmente un lista de esferas no contiene ninguna esfera. Ahora vamos a diseñar el modo en que podemos destruir las esferas contenidas en la lista.

Hay que destacar que puesto que la clase `ListaEsferas` no ha sido la responsable de la creación de las esferas que contiene, deberá ser la clase creadora, en nuestro programa la clase `Mundo`, la que debe asumir esta responsabilidad, aunque lo haga a través de la petición de una acción a la clase `ListaEsferas`.

Lo vamos a ir haciendo por partes para ver posibles errores que se pueden cometer a la hora de diseñar el proceso de destrucción.

Declarar y definir el método `void ListaEsferas::destruirContenido()` cuya función es la de destruir todas las esferas cuya dirección se contiene, e inicializar la lista, dejándola preparada para volver a contener nuevas direcciones de esferas.

```
void ListaEsferas::destruirContenido()
{
    for(int i=0;i<numero;i++)
          delete lista[i];

    numero=0;
}
```

Al terminar la ejecución de la clase `Mundo`, se ejecutará su destructor. Por tanto igual se debe indicar en este punto que se destruyan las esferas que habiendo sido creadas por esta clase, se almacenaron en la instancia de la clase `ListaEsferas`. Por tanto, editamos el contenido del destructor de `Mundo` y escribimos el siguiente código:

```
Mundo::~Mundo()
{
    esferas.destruirContenido();
}
```

Aparentemente no ha habido ningún cambio significativo en el programa, sin embargo, se ha comenzado a gestionar correctamente la memoria. Buscamos asegurar que la clase `Mundo` destruye, o limpia, todo lo que ha ido creando.

Sin embargo, vamos a ver ahora uno de los posibles errores que se puede cometer con la destrucción de objetos. Para ello, vamos a hacer que `esfera1` y `esfera2` estén contenidos dentro de la lista de esferas, de forma que ya no hay que llamar a sus métodos de pintado, de mover y de interacción de forma independiente.

Por tanto, al inicializar el objeto de la clase `Mundo`, agregaremos:

```
    esferas.agregar (&esfera1);
    esferas.agregar(&esfera2);
```

Por otro lado, vamos limpiando el código de `Mundo`, de forma que `esfera` y `esfera2`, ya no son gestionados de forma independiente. Eliminamos por tanto las llamadas específicas de estos dos objetos en las funciones `Dibuja` y `Mueve` de la clase `Mundo`. Compilamos y ejecutamos. Parece que todo va bien, hasta el momento en que cerramos el programa. En ese momento se produce un error de ejecución.

Si vemos por medio del *Debugger* que es lo que está pasando, se observa que el error es producido precisamente por el destructor que acabamos de escribir. La razón es que estamos intentando destruir a `esfera` y `esfera2`, que son atributos de la clase `Mundo`. Recuérdese la regla:

> Lo que se reserva con `new`, se destruye con `delete`, y lo que se reserva con `new []` se destruye con `delete []`, pero NUNCA SE DEBE LLAMAR A DELETE sobre algo que no ha sido creado con NEW.

Al ser atributos no reservados dinámicamente mediante `new`, el sistema detectará que no es una zona de memoria que admita una operación de `delete` y da un error.

Lo solucionamos eliminando los atributos `esfera` y `esfera2` de la cabecera de la clase `Mundo`, de forma que ya no constituyan un estamento privilegiado en el código del programa. Quedando visualmente igual, y habiendo eliminado los atributos y **todo el código** que usaba estos datos, el método `inicializa` de la clase `Mundo` quedará finalmente:

```
void Mundo::Inicializa()
{
    x_ojo=0;
    y_ojo=7.5;
    z_ojo=30;

    bonus.setPos(5.0f,5.0f);
    disparo.setPos(-5.0f,0.0f);
    plataforma.setPos(-5.0f,9.0f,5.0f,9.0f);

    Esfera *e1=new Esfera(1,2,4,5,15);
    e1->setColor(200,0,0);
    esferas.agregar(e1); //esfera

    Esfera *e2=new Esfera(2,-2,4,-5,15);
    e2->setColor(255,255,255);
    esferas.agregar(e2); //esfera2

    for(int i=0;i<6;i++)
    {
          Esfera* aux=new Esfera(0.75+i*0.25,i,1+i,i,i);
          esferas.agregar(aux);
    }
}
```

Un error solucionable por medio de la encapsulación es el que se produce cuando se intenta agregar una misma esfera dos veces. Si se ordenara la destrucción del contenido de lista esferas, daría error, puesto que se llamaría dos veces a `delete` sobre la misma dirección. En ese caso se producirá un error en la destrucción de nuevo porque se intenta destruir lo ya destruido.

Ejercicio: Implementar la funcionalidad necesaria en ListaEsferas::Agregar() para que no pueda ser agregada la misma esfera (el mismo puntero) mas de una vez.

6.5. AGREGANDO Y ELIMINANDO DINÁMICAMENTE ESFERAS

Durante la evolución del juego, se observa que una esfera grande impactada por un disparo, es eliminada y sustituida por otras dos de menor tamaño, que pasan a formar parte del escenario de juego. En el caso de que la esfera sea pequeña esta es destruida sin más. La agregación de esferas ya la tenemos resuelta, sin embargo, la destrucción y el acceso a las esferas contenidas en la lista no. Vamos a codificar esta funcionalidad:

Permitiremos la eliminación de esferas identificadas por su posición en el vector o directamente por su dirección. Por tanto, codificamos la siguiente función que permite borrar una esfera según su índice:

```
void ListaEsferas::eliminar(int index)
```


```
{
    if((index<0)||(index>=numero))
          return;
    delete lista[index];
    numero--;
    for(int i=index;i<numero;i++)
          lista[i]=lista[i+1];
}
```

Y de la misma forma, otra función que permita borrar una esfera según su dirección de memoria, que se apoya en la función anterior:

```
void ListaEsferas::eliminar(Esfera *e)
{
    for(int i=0;i<numero;i++)
          if(lista[i]==e)
          {
                eliminar(i);
                return;
          }
}
```

Para poder probarlas vamos a hacer que cuando una esfera se choque con el hombre, sea destruida ("comida"). Aunque realmente este no es el comportamiento final del juego, es un paso intermedio útil.

Para ello comenzaremos por codificar un nuevo método de la clase `Interaccion` que nos informe de cuando una esfera colisiona con el hombre (la función devuelve `true` si hay colisión y `false` si no la hay). El método es una cruda aproximación, en la que solo se coge la posición central del hombre, pero realmente no se tiene en cuenta sus dimensiones. Nótese que también ha sido necesario implementar los métodos `GetAltura()` y `GetPos()` de `Hombre`, para acceder a sus datos privados:

```
bool Interaccion::colision(Esfera e, Hombre h)
{
    Vector2D pos=h.getPos(); //la posicion de la base del hombre
    pos.y+=h.getAltura()/2.0f; //posicion del centro

    float distancia=(e.posicion-pos).modulo();
    if(distancia<e.radio)
          return true;
    return false;
}
```

Este método es utilizado por `ListaEsferas` para informarnos de la primera esfera de la lista que choca con la que se pasa como argumento.

```
Esfera * ListaEsferas::colision(Esfera &e)
{
    for(int i=0;i<numero;i++)
    {
          if(Interaccion::colision(e,*(lista[i])))
                return lista[i];
    }
    return 0; //no hay colisión
}
```

Hay que destacar que el valor de retorno de la función es utilizado además para indicar si hay o no colisión. Si se devuelve un cero, (un puntero a NULL en C), indicamos que ninguna de las esferas contenidas choca con la pasada como argumento.

Modificamos el método `Mundo::mueve()` de forma que utilizando dicha función, obtenemos la posible esfera que colisiona con el hombre, y la eliminamos de la clase `ListaEsferas`.

```
void Mundo::mueve()
{
    hombre.mueve(0.025f);
    bonus.mueve(0.025f);
    disparo.mueve(0.025f);

    esferas.mueve(0.025f);
    esferas.rebote();
    esferas.rebote(plataforma);
    esferas.rebote(caja);
    Esfera *aux=esferas.colision(hombre);
    if(aux!=0)//si alguna esfera ha chocado
          esferas.eliminar(aux);

    Interaccion::rebote(hombre,caja);
}
```

Agregamos la funcionalidad de las teclas '1','2','3' y '4' en `Mundo`:

```
void Mundo::tecla(unsigned char key)
{
    switch(key)
    {
          case '1':
                esferas.agregar (new Esfera(0.5f,0,10));
                break;
          case '2':
                esferas.agregar (new Esfera(1.0f,0,10));
                break;
          case '3':
                esferas.agregar (new Esfera(1.5f,0,10));
                break;
          case '4':
                esferas.agregar (new Esfera(2.0f,0,10));
                break;
    }
}
```

Ahora es cuando realmente podremos sorprendernos de lo cómodo que ha sido definir una clase contenedora que se encargue de gestionar con seguridad la evolución de las esferas. Si ejecutamos el programa podremos obtener resultados tan sorprendentes como el de la figura:

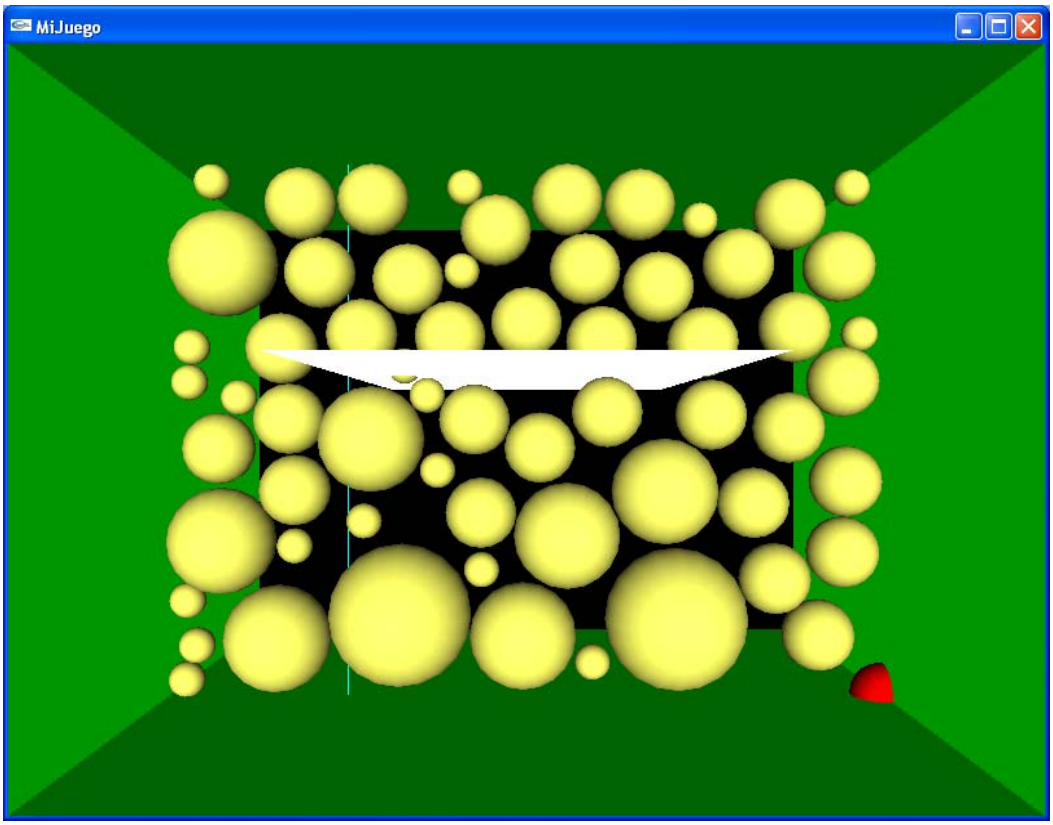

Figura 6-4. La utilidad de la clase ListaEsferas

6.6. ACCESO A LOS ELEMENTOS DE LISTAESFERAS

Para terminar, vamos a agregar dos métodos a la clase, que nos permitirán consultar las esferas contenidas en el contenedor.

Por un lado vamos a sobrecargar el operador `[]` de forma que nos dé la esfera situada en la posición indicada en el interior de los corchetes.

```
Esfera *ListaEsferas::operator [](int i)
{
    if(i>=numero)//si me paso, devuelvo la ultima
          i=numero-1;

    if(i<0) //si el indice es negativo, devuelvo la primera
          i=0;

    return lista[i];
}
```

Y por otro necesitaremos conocer el número de esferas contenidas en la lista, lo cual realizaremos directamente a través de una función `get inline` (en el fichero de cabecera *ListaEsferas.h*):

```
int getNumero(){return numero;}
```

Con ello es posible realizar operaciones específicas para cada esfera. Como ejemplo se puede codificar en la clase `Mundo` que las esferas que superen la altura 11 pasen a color rojo, y las que no a color blanco

```
    for(int i=0;i<esferas.getNumero();i++)
          if((esferas[i]->getPos()).y>11)
                esferas[i]->getColor(255,0,0);
          else
                esferas[i]->getColor(255,255,255);
```

Aparte de haber logrado un bonito cuadro "roji-blanco", y en contra de lo que parece, acabamos de construir una buena explicación del efecto de cómo es posible que el agua se evapore a temperaturas inferiores a 100 ºC. Cuando se pruebe este efecto, eliminar el código anterior, de tal forma que las esferas mantengan su color.

6.7. EJERCICIO PROPUESTO: LISTADISPAROS

Implementar la clase `ListaDisparos`, con la siguiente definición:

```
#define MAX_DISPAROS 10
#include "Disparo.h"
#include "Caja.h"

class ListaDisparos
{
public:
    ListaDisparos();
    virtual ~ListaDisparos();

    bool agregar(Disparo* d);
    void destruirContenido();
    void mueve(float t);
    void dibuja();

    void colision(Pared p);
    void colision(Caja c);

private:
    Disparo * lista[MAX_DISPAROS];
    int numero;

};
```

Para su correcto funcionamiento también se tienen que implementar los métodos:

```
class Interaccion
{
…
    static bool colision(Disparo d, Pared p);
    static bool colision(Disparo d, Caja c);

};
```

La clase `Disparo` debe de ser también completada para permitir acceso a sus variables:

```
class Disparo
{
…
```

```
    void setVel(float vx, float vy);
    float getRadio();
    Vector2D getPos();

};
```

Los disparos se realizarán cuando el usuario pulse el espacio:

```
void Mundo::tecla(unsigned char key)
{
    switch(key)
    {
          case ' ':
                {
                      Disparo* d=new Disparo();
                      Vector2D pos=hombre.getPos();
                      d->setPos(pos.x,pos.y);
                      disparos.agregar(d);
                      break;
                }
…
}
```

Obviamente, el disparo existente debe de ser eliminado y se debe instanciar un objeto de la clase `ListaDisparos` en la clase `Mundo`. El resultado final es el siguiente, en el que los disparos se paran cuando colisionan con la caja o la pared.

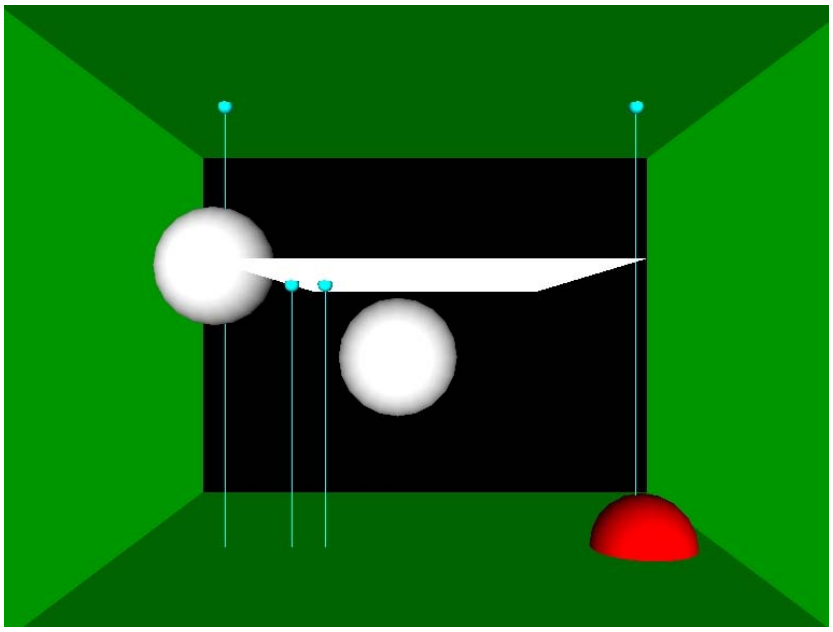

Figura 6-5. La lista de disparos en funcionamiento

6.8. ANEXO: ACERCA DE LA STL

Hemos implementado el vector interno a la clase `ListaEsferas` como un vector estático de una dimensión máxima definida por `MAX_ESFERAS`. C++ tiene una librería estándar, la *Standard Template Library* (STL) que permite (entre otras muchas cosas) el manejo de vectores dinámicos de cualquier tipo de objeto que pueden crecer o decrecer

en su tamaño. Se plantea en este anexo como se implementaría la funcionalidad anterior mediante el uso de las STL. No obstante, seguiremos usando en sucesivos capítulos nuestras clase `ListaEsferas` tal y como ha quedado implementada en los puntos anteriores, si bien gracias a la encapsulación se podría cambiar y el resto del programa no se vería afectado.

Como su propio nombre indica, la STL se construye principalmente haciendo uso de las plantillas (*templates*). Este es un mecanismo avanzado de la programación con C++, que queda fuera del alcance de este libro. De momento baste este ejemplo para ilustrar cómo se utiliza esta librería para implementar contenedores de objetos.

La declaración de un vector de la STL sería de la forma:

```
#include <vector>
class ListaEsferas
{
public:
    ListaEsferas();
    virtual ~ListaEsferas();
private:
    std::vector<Esfera*> lista;
};
```

Literalmente podemos traducir la última línea como “lista es un vector de punteros a Esfera”.

El vector inicialmente está vacío, y no es necesario inicializarlo:

```
ListaEsferas::ListaEsferas()
{
}
```

Para añadir una esfera basta con agregar al final del vector el elemento (push-back):

```
bool ListaEsferas::agregar (Esfera *esf)
{
    lista.push_back(esf);
    return true;
}
```

Es importante darse cuenta, que en condiciones normales (no superemos la memoria física del computador), podremos añadir todas las esferas que queramos, por lo que la función siempre devuelve `true`.

La dimensión del vector se puede conocer con la función `size()`, y el acceso a las componentes del vector se realiza gracias a la sobrecarga de operadores como si de un vector ordinario se tratase. Por tanto, el código del método `Dibuja()` lo podemos escribir como sigue:

```
void ListaEsferas::dibuja()
{
    for(int i=0;i<lista.size();i++)
          lista[i]->dibuja();
}
```

La eliminación de esferas conlleva dos operaciones. Por un lado hay que eliminar de la memoria el objeto, y por otro hay que quitar el puntero de la lista. Para esta última

operación es necesario seguir la sintaxis escrita en la última sentencia para evitar fallos de indexación:

```
void ListaEsferas::eliminarEsfera(int ind)
{
    if((ind<0)||(ind>=numero))
          return;

    delete lista[ind];

    lista.erase(lista.begin()+ind);
}
```

6.9. ANEXO: DISEÑO DE LA GESTIÓN DE INTERACCIONES ENTRE LISTAS DE OBJETOS

A lo largo del capitulo se ha utilizado un criterio y una solución que no necesariamente es la idónea desde el punto de vista arquitectónico de la aplicación. Cuando se añaden las clases que implementan las listas sin incluir en ellas la gestión de las interacciones de los elementos de las mismas, se tiene un diagrama de clases como el de la figura:

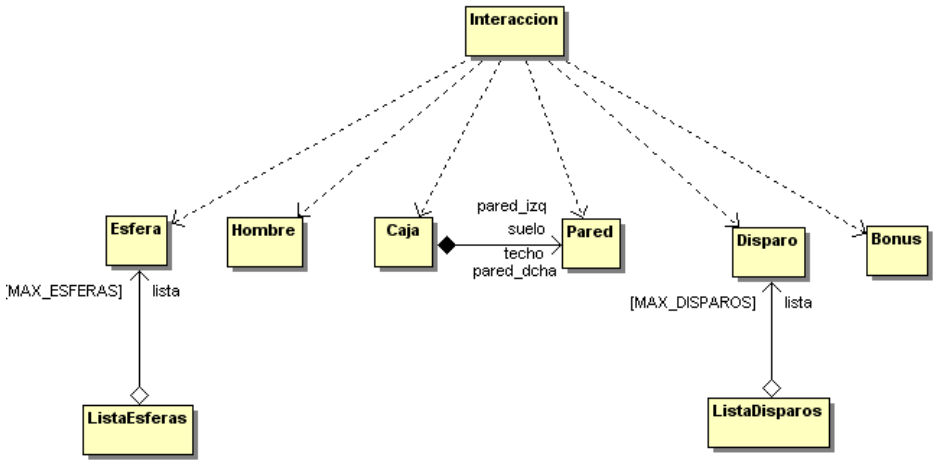

Figura 6-6. Diagrama de clases sin implementar la gestión de interacciones de las listas

La solución adoptada en el capítulo, asigna las responsabilidades de gestionar las interacciones de los conjuntos a las clases contenedoras de los objetos. Así, la clase `ListaEsferas` es la encargada de invocar el rebote de cada una de ellas con una pared dada mediante:

```
void ListaEsferas::rebote(Pared p)
{
    for(int i=0;i<numero;i++)
          Interaccion::rebote(*(lista[i]),p);
}
```

Esto en el diagrama se traduce en una dependencia de la clase `ListaEsferas` a la clase `Pared`. Si se representan en el diagrama todas las dependencias que aparecen en funciones similares de `ListaEsferas` y `ListaDisparos` nos encontramos con el diagrama de la figura:

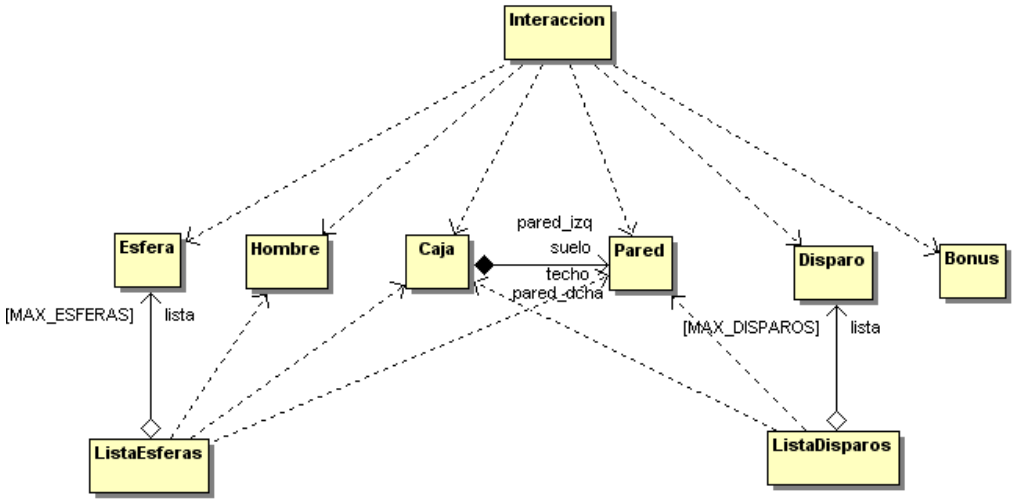

Figura 6-7. Dependencias generadas en la solución adoptada

No obstante, se puede conseguir un diagrama más limpio si utilizamos de nuevo el patrón **Indirección**, y nos llevamos toda la funcionalidad que implica las interacciones de listas de objetos a una nueva clase que podemos llamar `InteraccionListas`. Si la clase `Interaccion` calculaba interacciones entre pares de objetos, la clase `InteraccionListas` hace algo similar, pero gestionando las listas de objetos existentes en nuestro juego.

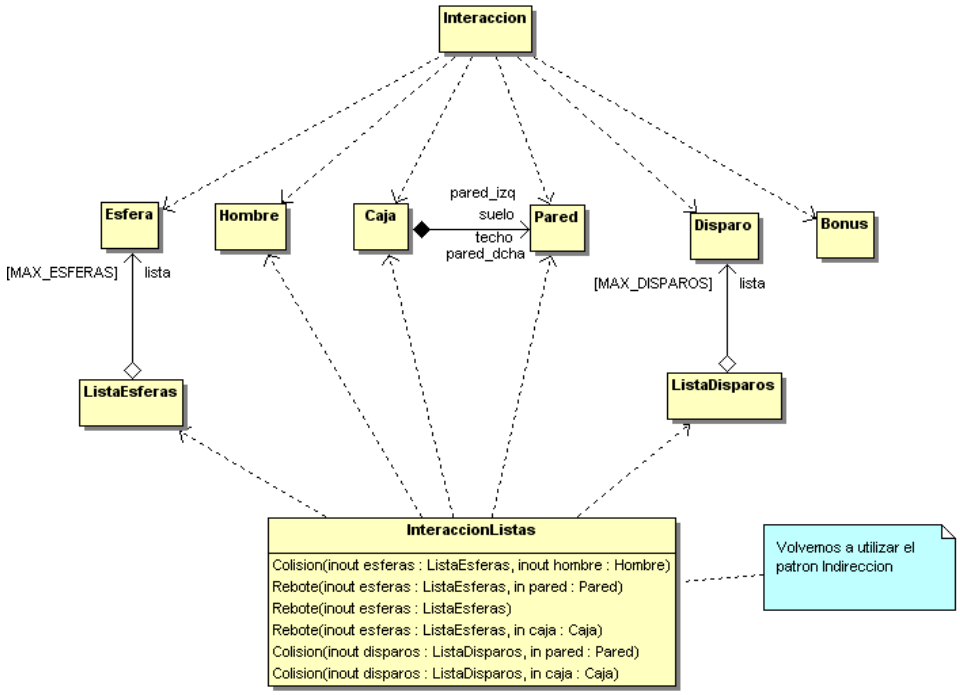

Figura 6-8. Un diseño más estructurado usando el patrón Indirección de nuevo