Tema 2:

Introducción a la programación orientada a objetos

***Objetivos**: en este tema se presentan los conceptos básicos que hay detrás de la programación orientada a objetos, así como el camino que han seguido los lenguajes de programación desde sus orígenes hasta llegar a abrazar este paradigma, el dominante en la actualidad.*

Índice

1 La abstracción en los lenguajes de programación

La abstracción es una herramienta que nos permite comprender, analizar y resolver problemas complejos del mundo real de un modo eficaz. La abstracción se fundamenta en eliminar todo aquello que no es relevante para un problema permitiendo centrarnos sobre aquella información que sí lo es; el ser o no relevante depende fuertemente del contexto.

Aunque la abstracción es una herramienta general aplicable a cualquier campo, a nosotros nos interesa la abstracción que proporciona un lenguaje de programación. Todos los lenguajes de programación proveen abstracciones. Muchos autores defienden que la complejidad de los problemas que un lenguaje de programación permite resolver depende del tipo y de la calidad de las abstracciones que el lenguaje permite crear.

El lenguaje ensamblador proporciona una pequeña abstracción respecto a la máquina sobre la que se ejecuta el programa. Los lenguajes de programación imperativos (Fortran, BASIC, Pascal, C, etc.) son abstracciones del lenguaje ensamblador. Si bien estos lenguajes son una gran aportación respecto al ensamblador, no son suficiente, ya que siguen requiriendo que el programador piense en **términos de la estructura del ordenador** más que en **términos del problema que tienen que resolver**. El programador se ve obligado a establecer una asociación entre el **modelo de la máquina** (que, por ejemplo, comprende variables enteras, reales, cadena de caracteres, instrucciones condicionales, bucles, etc.) y el **modelo del problema que pretende resolver** (que, por ejemplo, puede comprender cuentas bancarias con sus saldos y sus respectivos propietarios, fórmulas que indican cómo y qué intereses se deben de abonar en cada cuenta, comisiones que se deben de retirar, condiciones de cobro de hipotecas y/o letras del coche, etc.). El esfuerzo que requiere al programador realizar una correspondencia entre ambos mundos hace que los programas sean difíciles de escribir y de mantener y ha sido el detonante para que aparezcan una gran cantidad de metodologías de programación en la industria del software.

Una forma de aumentar el nivel de abstracción de los lenguajes de programación sería diseñar un lenguaje de muy alto nivel que incorporase múltiples estructuras de datos

generales y un conjunto de operaciones para manejar dichas estructuras. Estos lenguajes tratan de modelar el problema a resolver y no la máquina en la que se ejecutará el programa. Así, por ejemplo, en Lisp y APL se modela una visión particular del mundo: para el primero, todos los problemas se traducen en listas; para el segundo, en algoritmos. PROLOG, quizás el lenguaje de más difusión que sigue esta aproximación, modela todos los problemas como cadenas de decisiones. Todos estos lenguajes son una buena aproximación para resolver un tipo particular de problemas: aquéllos para los cuales aportan unas estructuras de datos y operaciones que componen una abstracción adecuada del problema. Sin embargo, cuando abandonamos el dominio para el cual han sido creados (por ejemplo, Lisp -> creación de redes, APL -> problemas matemáticos y compiladores , Prolog -> bases de datos o teoría de juegos) se convierten en terriblemente torpes y engorrosos, ya que no proporcionan las abstracciones adecuadas.

Intentar construir un lenguaje que modele la mayor parte de las abstracciones del mundo real que sus usuarios pudiesen demandar es un problema inabordable. La gran cantidad de estructuras de datos y de operaciones entre ellas que habría que contemplar convierten al problema del diseño de este lenguaje en intratable.

Llegados a este punto, la mejor solución al problema que se nos plantea es obvia: ya que no podemos proporcionar todas las abstracciones que cualquier usuario puede requerir para resolver cualquier problema, proporcionaremos la mejor herramienta posible (lenguaje de programación) para que el usuario construya las abstracciones que le permitan modelar el problema de un modo lo más adecuado posible. Ésta es la aproximación seguida por lenguajes de programación orientada a objetos: tratan de proporcionar herramientas generales (que no limiten al programador a un determinado dominio) para representar los elementos que componen el problema. Estos elementos que componen el problema se denominan "objetos". Por tanto, la idea es que el lenguaje se adapte a cada problema particular creando las abstracciones más adecuadas para él.

No obstante, esta aproximación no es incompatible con la anterior: los lenguajes de programación orientados a objetos ofrecen en sus librerías un conjunto cada vez más amplio de estructuras de datos ya implementadas y listas para ser empleadas por el programador. Sin embargo, la clave de su potencia es que permiten al programador definir nuevas estructuras extendiendo las ya existentes y/o modificándolas, o bien partiendo

desde cero cuando las librerías del lenguaje no le proporcionan ninguna herramienta útil en la que apoyarse.

1.1 Evolución de la abstracción en los lenguajes de programación

A principios de la historia de la programación la mayoría de los programas se desarrollaban en ensamblador. Como ya vimos en la asignatura de programación, por aquel entonces había grandes máquinas que ejecutaban pequeños programas; el software tenía un peso específico muy pequeño. A medida que los programas se hicieron más complejos los programadores encontraban más dificultades para recordar toda la información que necesitaban para desarrollar sus programas y el lenguaje ensamblador se manifestó como una herramienta insuficiente para abordar aquellos problemas.

La aparición de los lenguajes de alto nivel resolvió muchas de las dificultades del ensamblador, al mismo tiempo que aumentó las expectativas de los usuarios de sistemas informáticos, que cada vez demandaban una funcionalidad más sofisticada de los programas. Fue entonces cuando los programas empezaron ser inabordables por un único programador y empezó a ser habitual que fuesen abordados por un equipo de desarrolladores. Sin embargo, los programas eran sistemas muy complejos y con un alto grado de interconexión y los programadores necesitaban conocer una gran cantidad de detalles del trabajo de sus compañeros, lo cual les restaba tiempo para realizar su propio trabajo.

Como el propósito de este apartado es mostrar las limitaciones de la programación estructurada, vamos a emplear un lenguaje de programación que sólo soporta programación estructurada (C) en los ejemplos. En esta sección iremos refinando varias veces una misma implementación de un problema hasta llegar a la "mejor solución posible" dentro de un lenguaje de programación estructurado. Después apuntaremos las limitaciones e inconvenientes de esa solución, esto es, apuntaremos las causas que justificaron el nacimiento de la programación orientada a objetos.

A continuación, mostramos un programa que calcula la suma de dos vectores de tres dimensiones y el resultado de la suma es multiplicado vectorialmente por otro vector. La implementación se ha realizado de un modo similar a como se programaba cuando

surgieron los lenguajes de alto nivel; esto es, sin realizar uso de funciones ni módulos. Todo el programa es un único bloque de código.

```
/*
*ejemplo2_1.c
*/
#include <stdio.h>
#include <malloc.h>
typedef struct vector3D {
    int x,y,z;
} *vector;

main () {
    vector v1 = (vector)calloc(sizeof(struct vector3D),1);
    vector v2 = (vector)calloc(sizeof(struct vector3D),1);
    vector v3 = (vector)calloc(sizeof(struct vector3D),1);
    vector vs = (vector)calloc(sizeof(struct vector3D),1);
    vector vp = (vector)calloc(sizeof(struct vector3D),1);

    v1->x = 1; v1->y = 0; v1->z = 0;
    v2->x = 0; v2->y = 1; v2->z = 0;
    v3->x = 0; v3->y = 0; v3->z = 1 ;

    //Suma
    vs->x = v1->x + v2->x;
    vs->y = v1->y + v2->y;
    vs->z = v1->z + v2->z;
    printf("La suma es % d,%d,%d\n\n", vs->x, vs->y, vs->z);

    //Producto
    vp->x = vs->y * v3->z - vs->z * v3->y;
    vp->y = vs->x * v3->z - vs->z * v3->x;
    vp->z = vs->x * v3->y - vs->y * v3->x;
    printf("El producto es %d,%d,%d\n\n", vp->x, vp->y, vp->z);

    free (v1);
    free (v2);
    free (v3);
    free (vs);
    free (vp);
}
```

La siguiente evolución en el mundo de la programación fue el nacimiento de la programación procedimental; ésta se basa en un mecanismo de abstracción en el cual un bloque de código se ve como una caja negra que puede recibir o no unos parámetros y como resultado de su operación puede devolver o no un dato. Este mecanismo de abstracción permitió que un desarrollador sólo necesitase saber qué hacía el código de sus

compañeros y no cómo lo hacía. Surgió entonces un concepto clave: el de la ocultación de información. Era deseable que cada procedimiento no pudiese emplear información de los demás procedimientos para minimizar las dependencias entre ellos y obtener así una auténtica funcionalidad de caja negra. Sin embargo, los lenguajes de programación de la época sólo resolvían parcialmente este problema; a menudo las distintas funciones tenían que compartir datos, y por tanto unos programadores necesitaban conocer detalles del funcionamiento interno de otras funciones.

A continuación mostramos el mismo problema del ejemplo anterior resuelto empleando funciones:

```
/*
*ejemplo2_2.c
*/
#include <stdio.h>
#include <malloc.h>

typedef struct vector3D {
    int x,y,z;
} *vector;

vector productoVectorial(vector, vector);
vector sumaVectorial(vector, vector);
void imprimeVector (vector) ;

main () {
    vector v1 = (vector)calloc(sizeof(struct vector3D),1);
    vector v2 = (vector)calloc(sizeof(struct vector3D),1);
    vector v3 = (vector)calloc(sizeof(struct vector3D),1);
    vector vs, vp;

    v1->x = 1; v1->y = 0; v1->z = 0;
    v2->x = 0; v2->y = 1; v2->z = 0;
    v3->x = 0; v3->y = 0; v3->z = 1 ;

    //Suma
    vs = sumaVectorial(v1, v2);
    printf("La suma es ");
    imprimeVector(vs);

    //Producto
    vp = productoVectorial(vs,v3);
    printf("El producto es ");
    imprimeVector(vp);

    free (v1);
    free (v2);
    free (v3);
```

```
    free (vs);
    free (vp);
}

vector sumaVectorial(vector v1, vector v2) {
    vector vs = (vector)calloc(sizeof(struct vector3D),1);
    vs->x = v1->x + v2->x;
    vs->y = v1->y + v2->y;
    vs->z = v1->z + v2->z;
    return vs;
}

vector productoVectorial(vector v1, vector v2) {
    vector vp = (vector)calloc(sizeof(struct vector3D),1);
    vp->x = v1->y * v2->z - v1->z * v2->y;
    vp->y = v1->x * v2->z - v1->z * v2->x;
    vp->z = v1->x * v2->y - v1->y * v2->x;
    return vp;
}

void imprimeVector (vector v1) {
    printf("(%d Ux, %d Uy, %d Uz)\n\n", v1->x, v1->y, v1->z);
}
```

Para incrementar la independencia entre el código de diversos programadores y favorecer la reutilización surgió la programación modular. Esta es una técnica mejorada para crear espacios de nombres para las variables (de tal modo que los nombres de las variables que empleó un programador no colisionen con los nombres de las variables de los demás). Además, con la programación modular se incorporó el soporte para definir la parte pública y la parte privada de cada unidad de código, de tal modo que dentro de un módulo pudiese haber partes que constituyen "detalles de implementación" y que resultan invisibles a otros programadores que usen ese módulo, y partes públicas que se exportan para ser usadas por otros programadores en otros módulos.

Mostramos una nueva implementación de nuestro problema siguiendo una aproximación modular:

```
/*
*ejemplo2_3.c
*/

#include <stdio.h>
#include <malloc.h>
#include "vector.h"

main () {
```

```
    vector v1 = (vector)calloc(sizeof(struct vector3D),1);
    vector v2 = (vector)calloc(sizeof(struct vector3D),1);
    vector v3 = (vector)calloc(sizeof(struct vector3D),1);
    vector vs, vp;

    v1->x = 1; v1->y = 0; v1->z = 0;
    v2->x = 0; v2->y = 1; v2->z = 0;
    v3->x = 0; v3->y = 0; v3->z = 1 ;

    //Suma
    vs = sumaVectorial(v1, v2);
    printf("La suma es ");
    imprimeVector(vs);

    //Producto
    vp = productoVectorial(vs,v3);
    printf("El producto es ");
    imprimeVector(vp);

    free (v1);
    free (v2);
    free (v3);
    free (vs);
    free (vp);
}


/*
*ejemplo2_3.c
*vector.h
*/

typedef struct vector3D {
    int x,y,z;
} *vector;

extern vector productoVectorial(vector, vector);
extern vector sumaVectorial(vector, vector);
extern void imprimeVector (vector);


/*
*ejemplo2_3.c
*vector.c
*/

#include "vector.h"
#include <stdlib.h>
#include <stdio.h>

vector sumaVectorial(vector v1, vector v2) {
    vector vs = (vector)calloc(sizeof(struct vector3D),1);
    vs->x = v1->x + v2->x;
```

```
    vs->y = v1->y + v2->y;
    vs->z = v1->z + v2->z;
    return vs;
}

vector productoVectorial(vector v1, vector v2) {
    vector vp = (vector)calloc(sizeof(struct vector3D),1);
    vp->x = v1->y * v2->z - v1->z * v2->y;
    vp->y = v1->x * v2->z - v1->z * v2->x;
    vp->z = v1->x * v2->y - v1->y * v2->x;
    return vp;
}

void imprimeVector (vector v1) {
    printf("(%d Ux, %d Uy, %d Uz)\n\n", v1->x, v1->y, v1->z);
}
```

En este paradigma se hizo habitual que los datos de un cierto tipo (por ejemplo, los datos del tipo "cuentas bancarias") fuesen gestionados por un módulo administrador de dicho tipo de datos. Este módulo proporcionaba un conjunto de operaciones para manipularlos, a la vez que proporcionaba mecanismos para ocultar los datos y una abstracción fácil de manejar de ellos (por ejemplo, para transferir dinero de una cuenta bancaria a otra bastaría con invocar una función transferir (cuenta_origen, cuenta_destino) del módulo CuentaBancaria, sin necesidad de conocer en detalle las operaciones que hay que realizar para transferir dinero entre dos cuentas). Sin embargo, la programación modular no proporcionaba soporte para algunas operaciones muy comunes sobre estos tipos de datos, como puede ser la creación de ejemplares de dicho tipo (crear cuentas bancarias). El programador del módulo gestor del tipo de datos tenía la responsabilidad de resolver este problema, problema al que, por otro lado, se tenían que enfrentar todos los programadores que querían desarrollar un módulo gestor de cualquier tipo de datos.

Surgió entonces el concepto de tipo de datos abstractos. Un tipo de dato abstracto es definido por un programador, pero puede ser manipulado de un modo similar a los datos definidos por el lenguaje de programación. Al igual que éstos, un tipo de dato abstracto puede tener un rango de valores lícitos y un conjunto de operaciones que se pueden ejecutar sobre ellos. Para construir un tipo de dato abstracto es necesario:

- Exportar una definición del tipo.
- Proporcionar las operaciones que pueden usarse para manipular instancias.

- Impedir que los datos internos del tipo sean accesibles a través de otro mecanismo distinto de las operaciones provistas con tal fin.
- Proporcionar mecanismos para crear múltiples instancias del tipo.

En un tipo de dato abstracto lo importante es cuáles son sus valores posibles y cuál es la semántica de las operaciones que se pueden realizar sobre él (su especificación). El cómo se representan los datos o el cómo se implementan las operaciones se ignora y, lo más importante, no se necesita en absoluto para acceder a su funcionalidad. Por tanto, un programador que vaya a emplear el tipo de dato abstracto puede centrarse en qué hace y no en cómo lo hace. Además, si por cualquier motivo se tiene que cambiar la forma en la que se realiza una operación o la representación interna de los datos esto no afectará a otros programadores mientras no varíe la semántica de las operaciones posibles del tipo de datos.

Mostramos una nueva versión de nuestro ejemplo en la cual se ha construido un tipo de dato abstracto para representar a los vectores:

```
/*
*ejemplo2_3.c
*/
#include <stdlib.h>
#include <stdio.h>
#include "TADvector.h"

main () {
    vector v1 = crearVector (1, 0, 0);
    vector v2 = crearVector (0, 1, 0);
    vector v3 = crearVector (0, 0, 1);
    vector vs, vp;

    //Suma
    vs = sumaVectorial(v1, v2);
    printf("La suma es ");
    imprimeVector(vs);

    //Producto
    vp = productoVectorial(vs,v3);
    printf("El producto es ");
    imprimeVector(vp);

    liberarTodosVectores();
}


/*
```

```
*ejemplo2_3.c
*TADvector.h
*/

typedef unsigned int vector;

extern vector crearVector (int x, int y, int z);
extern void liberarVector (vector v1);
extern vector sumaVectorial(vector v1, vector v2);
extern vector productoVectorial(vector v1, vector v2);
extern void imprimeVector (vector v1);
extern void liberarTodosVectores();
```

```
/*
*ejemplo2_3.c
*TADvector.c
*/
#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>
#include "TADvector.h"
#define TAM_MAX 100

typedef struct vector3D {
    int x,y,z;
} *vector3DInterno;

static vector3DInterno vectores[TAM_MAX] ; //contiene los vectores
static int usados[TAM_MAX]= {0}; //contiene 1 si el vector está
usado 0 en caso contrario

vector crearVector(int x, int y, int z) {
    int i;
    for (i = 0; i < TAM_MAX; i++) {
        if (!usados [i]) {
            vectores[i] = (vector3DInterno)calloc(sizeof(struct
vector3D),1);
            vectores[i] ->x = x;
            vectores[i] ->y = y;
            vectores[i] ->z = z;
            usados [i]  = 1;
            break;
        }
    }
    return i;
}

void liberarVector (vector v1) {
    free (vectores[v1]);
    usados[v1]=0;
}
```

```
void liberarTodosVectores() {
    int i;
    for (i = 0; i < TAM_MAX; i++) {
        if (usados [i]) {
            liberarVector (i);
            usados[i]=0;
        }
    }
}

vector sumaVectorial(vector v1, vector v2) {
    vector vs = crearVector (vectores[v1]->x + vectores[v2]->x,
                             vectores[v1]->y + vectores[v2]->y,
                             vectores[v1]->z + vectores[v2]->z);
    return vs;
}

vector productoVectorial(vector v1, vector v2) {
    vector vp = crearVector (vectores[v1]->y * vectores[v2]->z -
vectores[v1]->z * vectores[v2]->y,
                             vectores[v1]->x * vectores[v2]->z -
vectores[v1]->z * vectores[v2]->x,
                             vectores[v1]->x * vectores[v2]->y -
vectores[v1]->y * vectores[v2]->x);
    return vp;
}

void imprimeVector (vector v1) {
    printf("(%d Ux, %d Uy, %d Uz)\n\n", vectores[v1]->x,
vectores[v1]->y,
           vectores[v1]->z);
}
```

A pesar de que los tipos de datos abstractos supusieron un gran avance en ingeniería de software, son insuficientes. Una vez definido un tipo de dato no es posible adaptarlo a nuevas aplicaciones y modificar su definición. El problema está en que la abstracción que proporcionan no permite distinguir entre las propiedades generales de todas las instancias del tipo y las propiedades particulares de una instancia concreta. Así, por ejemplo, podemos crear un tipo de dato que represente una cuenta bancaria. Tanto una cuenta corriente como una cuenta a plazo fijo son cuentas bancarias y comparten muchas propiedades y operaciones. Dado el tipo de datos "cuenta bancaria" sería deseable poder derivar los tipos de datos "cuenta corriente" y "cuenta a plazo fijo" basándose en el tipo de dato anterior, y no empezando desde cero.

Supongamos, por ejemplo, que queremos realizar un programa que opere con vectores en un espacio de dos dimensiones; esto es, con vectores confinados al plano XY. Estos

vectores son un caso particular de los vectores anteriores para los cuales la componente z siempre debe valer 0. Sin embargo, empleando tipos de datos abstractos no se dispone de un mecanismo que permita crear un tipo de dato que represente vectores de dos dimensiones reutilizando en la medida de lo posible la implementación realizada para los vectores del espacio de tres dimensiones.

Aquí es donde surge la programación orientada a objetos. La programación orientada a objetos proporciona mecanismos para crear distintos tipos de datos que comparten un código común (herencia), a la vez que permite adaptar el código compartido para que se ajuste a las necesidades específicas de cada tipo de datos concreto (polimorfismo). Otra aportación de la programación orientada a objetos es la idea del paso de mensajes entre las distintas instancias de los tipos de datos. Los objetos interactúan entre ellos mediante solicitudes que se realizan enviándose mensajes, y no mediante una invocación de una función. En el fondo, la ejecución de una interacción se lleva a cabo de un modo muy similar. Sin embargo, ahora se concede más importancia al valor devuelto por la operación y no a la propia operación.

2 Programación orientada a objetos

Alan Kay, uno de los diseñadores del primer lenguaje orientado a objetos que consiguió una aceptación considerable, Smalltalk, resume en cinco las características que debe tener una programación totalmente orientada a objetos:

1. **Todo es un objeto**. Debemos de pensar que todo es un objeto, una "variable" que almacena datos y que es capaz de responder a una serie de peticiones de otros objetos que le piden que realice un conjunto de operaciones sobre sí misma. En teoría, cualquier objeto que componga tu problema en el mundo real puede ser representado con un objeto en tu programa.
2. **Un programa es un conjunto de objetos que interactúan entre ellos pasándose mensajes**. Para hacer cualquier petición a un objeto se le envía un mensaje.
3. **Cada objeto tiene su propia memoria y puede estar formado por otros objetos**. Es posible crear un nuevo objeto "empaquetando" otros objetos ya existentes. Esto permite crear programas muy complejos cuya complejidad queda escondida detrás de la simplicidad de un conjunto de objetos.

4. **Todos los objetos tienen un tipo**. Cada objeto se considera una instancia de una clase, siendo la clase el tipo al cual pertenece el objeto. Lo más importante de una clase es ¿qué tipos de mensaje se puede enviar a los objetos de esa clase?
5. **Todos los objetos de un tipo particular pueden recibir los mismos mensajes**. Dado que un perro es un mamífero, un perro debe poder responder a cualquier mensaje al que un mamífero pueda responder. Esto permite escribir código en el que se envíe mensajes a los mamíferos y luego emplearlo para cualquier objeto que sea un mamífero.

Estudiemos más en detalle cada uno de estos puntos.

2.1 Todo es un objeto

En torno a este punto no existe un acuerdo común inapelable en la bibliografía. ¿Es necesario que para que un programa se considere un programa orientado a objetos que todo dentro de él sea un objeto? Y, para que un lenguaje de programación sea un lenguaje orientado a objetos, ¿todos sus tipos de datos deben ser objetos? Algunos puristas responderían que sí a ambas preguntas; y efectivamente esta es la aproximación seguida por algunos lenguajes de programación como Smalltalk y Eiffel, lenguajes de programación que, por otro lado, no son precisamente los que gozan de más popularidad y difusión.

Uno de los problemas de que todo sea un objeto es la eficiencia del lenguaje de programación: considerar que entidades tan simples como los datos reales, enteros y binarios (boolean) sean objetos puede ocasionar que operaciones tan básicas como la suma de dos números enteros tenga un costo computacional excesivo. Por este motivo muchos lenguajes de programación como C++, C# y Java (que, sin duda, se encuentran entre los más populares hoy en día) contemplan tipos de datos que no son objetos.

Para algunos, los lenguajes de programación orientados a objetos "puros" son sólo los primeros; mientras que los segundos son "híbridos". Evitaremos entrar en más discusiones de carácter filosófico, y tomaremos como cierta una afirmación un poco más relajada que la realizada por Alan: "casi todo es un objeto". En cualquier caso, nadie pone en duda que los objetos son el pilar básico de la programación orientada a objetos.

Un objeto representa una entidad física del mundo real (perro, avión, cajero automático, etc.), conceptual (cuenta bancaria, hipoteca, reacción química, etc.) o de software (una cadena de caracteres, una lista enlazada, una caché, etc.). Según la definición oficial de la OMG, el consorcio internacional dedicado a la estandarización de las tecnologías orientadas a objetos, un objeto es "Una entidad delimitada precisamente y con identidad, que encapsula estado y comportamiento. El estado se representa mediante sus atributos y relaciones, y el comportamiento mediante sus operaciones, métodos y máquinas de estados. Un objeto es una instancia de una clase". En esta definición hay tres conceptos clave:

- **Identidad**: es aquella propiedad de un objeto que lo distingue de los demás. El propio nombre del objeto, que siempre debe ser único, determina su identidad.
- **Estado**: comprende todas las características del objeto y dependiendo de su estado un objeto puede responder a una misma petición de diferente modo.
- **Comportamiento**: es el modo en el que actúa y se relaciona; depende del estado en que se encuentra.

Podemos considerar que un objeto es una entidad que posee un determinado estado y manifiesta cierto comportamiento.

2.2 Un programa es un conjunto de objetos que interactúan entre ellos

La interacción entre los objetos se basa en el paso de mensajes entre ellos. Un mensaje puede considerase como una orden que se envía a la un objeto para indicarle que realice alguna acción. Cuando el objeto recibe un mensaje se ejecuta el código correspondiente con una **función miembro** o **método**. Las funciones miembro son similares a las funciones de C, pero se diferencian de éstas en que están asociadas a un determinado objeto y, por tanto, una misma función miembro invocada sobre dos objetos diferentes de un mismo tipo (clase) puede realizar operaciones diferentes dependiendo del estado del objeto. Las operaciones que se pueden realizar sobre un objeto, según Booch, se pueden clasificar en cinco grupos básicos:

- **Operación de modificación**: la operación modifica el estado del objeto.
- **Operación de construcción**: la operación crea un objeto e inicializa su estado.

- **Operación de destrucción**: la operación destruye un objeto o su estado.
- **Operación de selección**: la operación accede al estado del objeto sin modificarlo.
- **Operación de iteración**: la operación permite acceder a los atributos del objeto en un orden preestablecido.

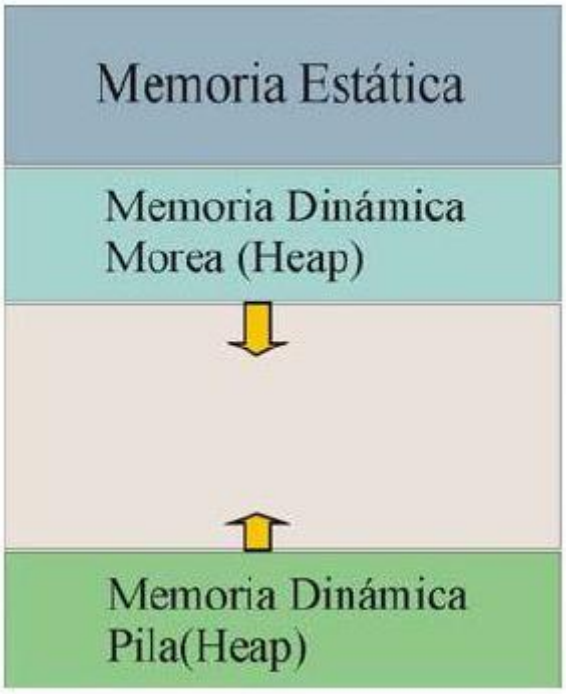
El paso de un mensaje implica a dos partes: al emisor y al receptor. Cuando el objeto emisor envía un mensaje a un objeto receptor tiene que especificar lo siguiente:

- El objeto receptor del mensaje.
- El nombre del mensaje.
- Argumentos o parámetros del mensaje, si éste lo requiere.

En primer lugar, el objeto emisor construye el mensaje agrupando los parámetros que necesita enviar al objeto receptor. El receptor comprueba si el mensaje se encuentra entre aquellos que entiende, y si es así lo acepta y responde al mensaje invocando la función miembro asociada.

2.3 Cada objeto tiene su propia memoria hecha de otros objetos

Dado que hemos relajado los requerimientos que hacemos a un lenguaje de programación orientado a objetos y permitimos que en él haya tipos de datos no objetos, es posible que un objeto esté construido tanto por objetos como por tipos de datos no objetos.



La memoria de un objeto se reserva cuando se realiza una operación de construcción que crea una instancia de una determinada clase, y se libera cuando se realiza una operación de destrucción sobre dicha instancia. El cómo se controla el ciclo de vida de los objetos depende de cada lenguaje de programación concreto. En el caso particular que nos atañe, Java, se ha optado por la aproximación que incrementa la sencillez de uso del lenguaje. Todos los

objetos en Java se crean en el heap, o área de almacenamiento dinámica. Sin embargo, a diferencia de lo que sucedía en C, el entorno de ejecución de Java nos proporciona un "recogedor de basura" para el heap; esto es, un gestor que automáticamente descubre cuando los objetos dejan de ser útiles y los destruye. Esto simplifica la tarea al programador, a la vez que disminuye la eficiencia de ejecución del programa.

Cada objeto puede contener dentro de él otros objetos y/o tipos de datos no objeto. Un objeto internamente puede considerarse formado por dos componentes básicos: atributos y funciones miembro. Los atributos describen el estado del objeto y tienen dos partes: un nombre y un valor. Cada uno de estos atributos puede ser un tipo primitivo u otro objeto. En cuanto a las funciones miembro, son las que describen el comportamiento asociado al objeto y las que permiten que el objeto preste servicios a otros objetos

En el siguiente apartado, veremos qué posibles relaciones puede guardar un objeto con aquellos que lo componen.

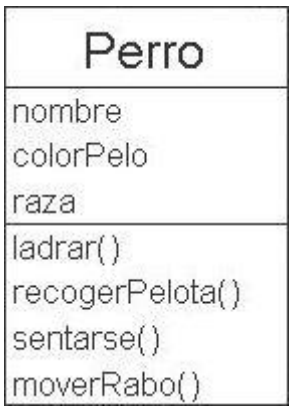
2.4 Todos los objetos tienen un tipo (clase)

Una clase es una descripción de un conjunto de objetos que manifiestan los mismos atributos, operaciones, relaciones y la misma semántica. Una clase puede verse desde tres perspectivas diferentes pero, a la vez, complementarias:

- Como un conjunto de objetos que comparten una estructura y un comportamiento.
- Como una plantilla que permite crear objetos.
- Como la definición de la estructura y del comportamiento de una clase.

La primera perspectiva hace énfasis en la clasificación y en la abstracción. Una clase es una abstracción software de un conjunto de objetos (que se pueden corresponder con objetos del mundo real o no) que, por compartir una serie de atributos y comportamiento, clasificamos bajo una misma etiqueta. La segunda perspectiva toma un enfoque un tanto utilitarista: la clase es la "herramienta" que los lenguajes de programación emplean para construir objetos. La tercera hace énfasis en que la definición de una clase es una estructura común que se puede reutilizar (crear objetos con ella) cuántas veces se quiera. A la acción de crear un objeto de una clase se la denomina **instanciar**; y a los objetos creados **instancias** de la clase.

Una clase define atributos, que servirán para representar el estado de sus instancias, y operaciones, que establecen el comportamiento de las instancias. Un atributo es una abstracción de una característica común para todas las instancias de la clase. Cuando se llega a la fase de implementación a los atributos a menudo se les denomina variables instancia o datos miembros, y a las implementaciones de las operaciones funciones miembro.

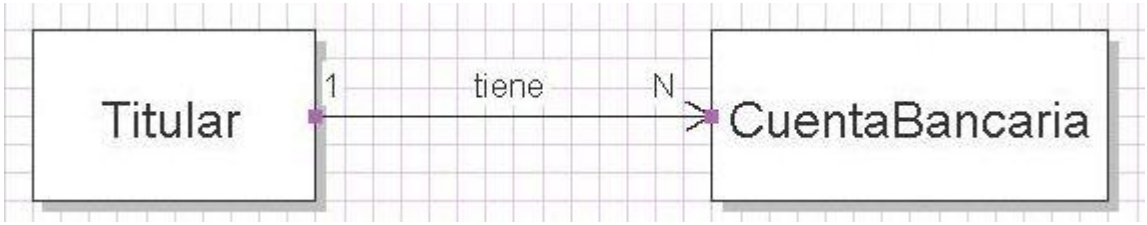


El Lenguaje Unificado de Modelado (UML, Unified Modelling Language) es el lenguaje de modelado de software más empleado actualmente; aún cuando todavía no es un estándar oficial, está apoyado en gran manera por la OMG. Este lenguaje es de carácter gráfico y es al ingeniero del software lo mismo que los planos de un edificio son a un arquitecto. En él una clase se representa mediante un rectángulo donde el nombre de la clase va en la parte superior, a continuación suelen ir los atributos (propiedades) de la clase y finalmente las funciones miembro o métodos (comportamiento). En la imagen mostramos cómo se podría representar la clase "Perro" en UML

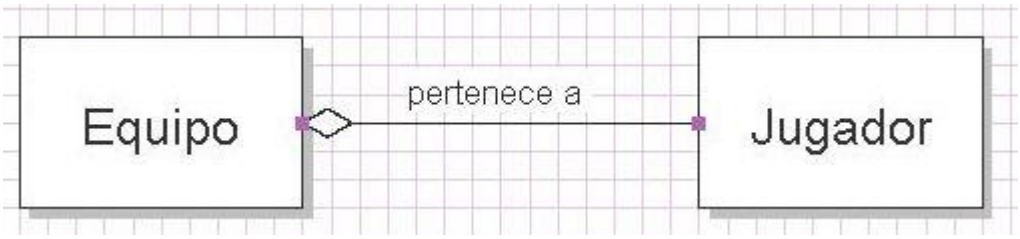
2.4.1 Relaciones entre clases

La relación más simple entre dos clases es la de asociación; es el tipo más general de relación y el de menor contenido semántico. En muchas ocasiones este tipo de relación se emplea sólo en las etapas más tempranas del análisis orientado a objetos y a medida que se avanza en el diseño se sustituye por otras relaciones más específicas. Las relaciones de asociación entre dos objetos pueden tener distintas multiplicidades: puede ser "uno a uno": una cuenta bancaria tiene un saldo, "uno es a varios": una persona puede tener una o más cuentas bancarias.

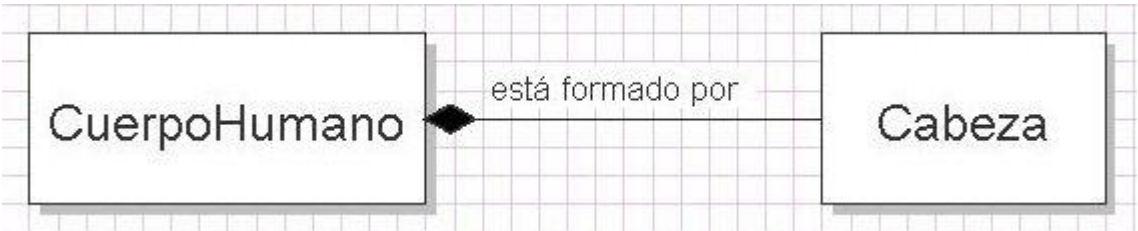
En UML esta relación se representa mediante una flecha de línea continua y cuya punta está abierta. La multiplicidad de la relación se suele indicar en cada uno de los extremos de la flecha, como se muestra el dibujo, y el tipo de relación se indica textualmente sobre la flecha:



La agregación es una especialización de las asociaciones que representa una relación del tipo "todo-parte". Los objetos forman parte del objeto todo, pero la vinculación entre ellos y el todo no es absoluta: se pueden crear y destruir de modo independiente objetos de tipo todo y de tipo parte. Un ejemplo de esta relación sería un equipo de fútbol y sus jugadores. Los jugadores forman parte del equipo de fútbol, sin embargo son independientes de éste y, periódicamente, es normal que algunos se vayan y vengan otros nuevos. Es más, en el hipotético caso de que todos los jugadores de un equipo de fútbol abandonarán el equipo al mismo tiempo el equipo de fútbol seguiría existiendo y podría buscar una nueva plantilla. En UML la relación de relación se representa mediante una línea en uno de cuyos extremos (el correspondiente con el todo) hay un rombo vacío:



La composición es otra especialización de las asociaciones que modela también una relación del tipo "todo-parte", pero con un vínculo absoluto y permanente. La composición implica que los ciclos de vida de los objetos parte y el objeto todo se hallan fuertemente relacionados en el tiempo. Cuando se crea un objeto todo se crea cada uno de los objetos parte y cuando se destruye el objeto todo se destruyen los objetos parte. Un objeto parte no puede asignarse a otro objeto todo diferente al cual fue creado. Esta es la relación que existe entre el objeto "cuerpo humano" y sus diversas partes: cabeza, tronco, órganos vitales, etcétera. En UML esta relación se representa de un modo similar a la agregación, pero con un rombo cuya punta está rellena:



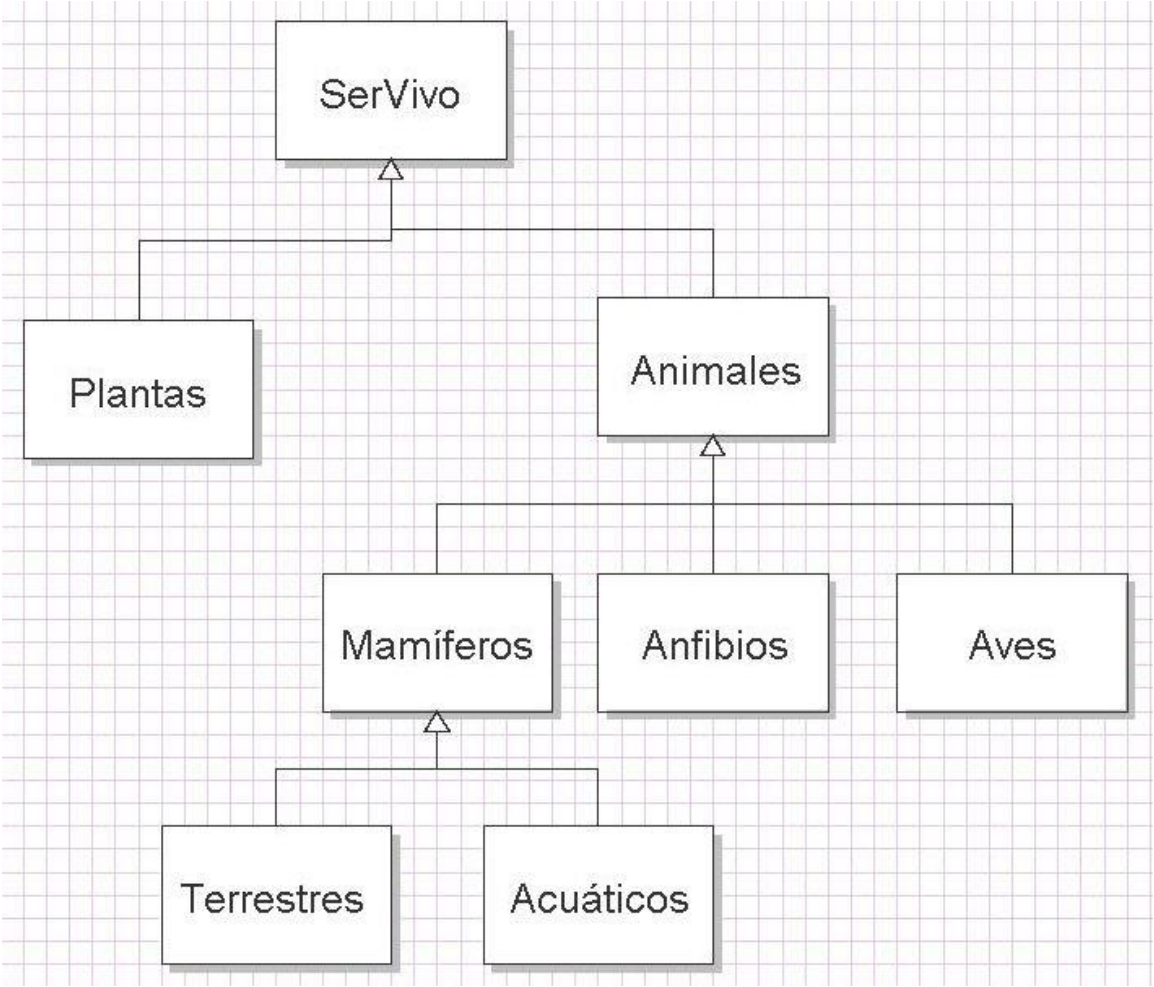
Todos estos tipos de relaciones se representan en software mediante uno o varios atributos en uno de los objetos que contienen una referencia al otro objeto.

2.4.2 Herencia

La herencia permite que una clase pueda basarse en otra ya existente para construirse; constituye, por tanto, un mecanismo muy potente de reutilización de código. Mediante ella una clase, denominada normalmente **clase hija**, **clase derivada** o **subclase**, hereda propiedades y comportamiento de otra clase, denominada **clase padre**, **clase base** o **superclase**. La herencia organiza jerárquicamente las clases que se necesitan para resolver un problema.

La herencia se apoya en el significado de ese concepto de la vida diaria. Así, los seres vivos se dividen en animales y plantas; los animales, a su vez, en mamíferos, anfibios, insectos, aves, reptiles... los mamíferos a su vez en terrestres y acuáticos... cada una de estas clases o categoría tiene un conjunto de atributos y comportamiento común a todos los miembros de la clase: todas las hembras de los mamíferos poseen glándulas mamarias que sirven para alimentar a sus crías cuando nacen. Estas propiedades y comportamiento son heredadas por todas las subclases de una clase: lo dicho antes para los mamíferos es igualmente válido para los perros, ya que los perros son mamíferos (pertenecen a la clase de los mamíferos).

En UML la relación de herencia entre dos clases se representa mediante una flecha cuyo extremo está cerrado y vacío y se encuentra apuntando hacia la clase padre:

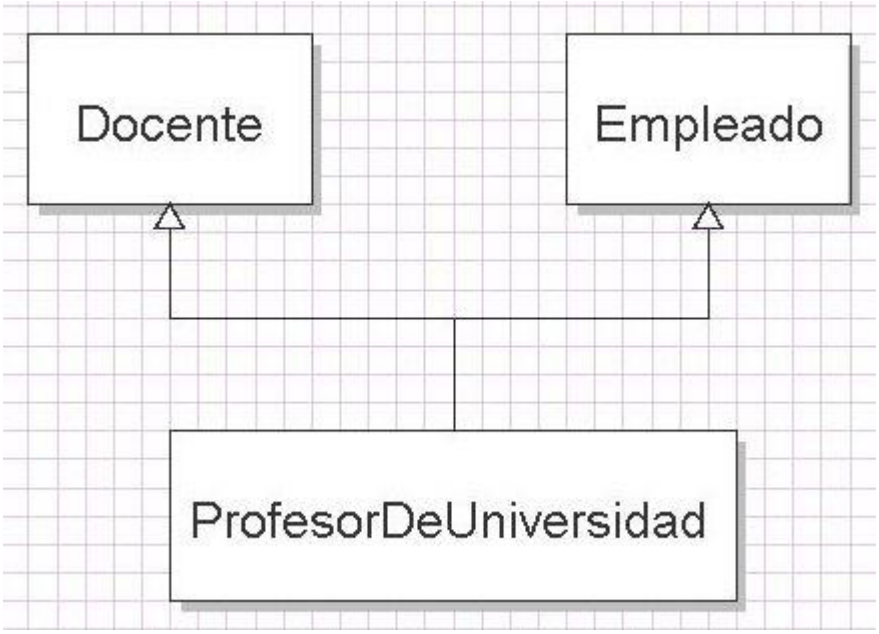


Una clase hija especializa y extiende el comportamiento de la clase padre; así, un perro es un mamífero, exactamente igual que la ballena; sin embargo, un perro también tiene pelo y cuatro patas, cosa que ya no sucede con todos los mamíferos. Podemos decir que un perro es "un tipo más especializado/concreto" de mamífero, que extiende las propiedades y el comportamiento de un mamífero incorporando pelo, cuatro patas, el alimentarse de carne y el ladrar, entre otros.

Una forma de identificar relaciones de tipo herencia es buscando en la descripción del problema las palabras "**es un**": un perro es un mamífero, un mamífero es un ser vivo, etcétera. El sustantivo que va antes del "es un" se corresponderá con una clase hija y el que va después con la clase padre. No debe confundirse la herencia con la composición. Ambos son mecanismos para reutilizar software que están disponibles en los lenguajes de programación orientada a objetos; sin embargo, mientras que la herencia es un mecanismo propio de la programación orientada a objetos la composición ya se empleaba en la programación estructurada. La composición se suele corresponder con las palabras "**tiene**

un" de la descripción de un problema: un equipo de fútbol tiene un entrenador; un equipo de fútbol tiene varios jugadores; una cuenta bancaria tiene un salto, etcétera.

La herencia puede ser simple o múltiple. La herencia simple se corresponde con una relación uno es a uno entre la clase hija y la padre: cada hijo sólo tiene un padre, aunque un padre puede tener varios hijos. La herencia múltiple se corresponde con una relación uno es a varios entre la clase hija y las padres: un hijo puede tener varios padres. Así, un profesor de Universidad es un docente y un empleado:



C++ soporta tanto la herencia simple como la herencia múltiple. Sin embargo, en lenguajes de programación más actuales, como Java y C#, se ha optado por no soportar la herencia múltiple ya que, si bien es muy potente, añade una gran complejidad al lenguaje de programación que no siempre justifica los beneficios que se obtiene de ella y que lleva a los programadores a tratar de evitarla. Uno de los problemas de la herencia múltiple se halla en la herencia repetida: la clase A que hereda de las clases B y G, que a su vez heredan de D. Por tanto, la clase A hereda dos veces los atributos y el comportamiento de la clase D. El otro problema de la herencia múltiple son los conflictos de nombres: una clase A hereda simultáneamente de B y C que usan el mismo nombre para un atributo o función miembro. ¿Qué definición empleará la clase hija para dicho atributo o método?

2.5 Todos los objetos de una misma clase pueden recibir los mismos mensajes

Dado que un objeto del tipo "perro" es también un objeto del tipo "mamífero" un perro aceptará todos los mensajes que acepta un mamífero. Esto quiere decir que es posible escribir código que interaccione con mamíferos y que sea capaz de gestionar cualquier tipo de mamífero. El tratar a un objeto no como un tipo específico, sino como al tipo de su clase base, permite que el código no dependa de detalles específicos de un objeto. Esta capacidad, que se denomina **polimorfismo**, es una de las herramientas más poderosas de la programación orientada a objetos.

El polimorfismo permite que el código no sufra alteraciones cuando se añaden nuevos tipos derivados (clases hijas) y añadir nuevos tipos derivados es precisamente el modo más común de extender un lenguaje de programación orientado a objetos para resolver nuevos problemas. El polimorfismo es, a fin de cuentas, una potente herramienta de abstracción: permite centrarnos en la semántica de una operación y no en sus detalles. Así, todos los mamíferos amamantan a sus crías cuando nacen, lo que en programación orientada a objetos podría traducirse en que la clase "mamífero" posee una función miembro "amamantar ()". Obviamente, el modo en el que una ballena amamanta su cría en el mar, el modo en que una canguro amamanta a la suya en su marsupio, o el cómo lo hace un gato doméstico, son completamente diferentes. Sin embargo, la semántica de la operación "amamantar" es común a todos estos casos.

¿Cómo se da soporte al polimorfismo en un lenguaje de programación orientado a objetos? Una clase hija puede redefinir la implementación de los métodos de la clase padre; mediante este mecanismo la clase padre define la especificación que deben cumplir todas las hijas y es responsabilidad de cada una de las hijas (de los programadores que las creen) implementar los detalles particulares de la especificación. Esto supone un problema: si vamos a invocar una función miembro genérica sobre un objeto que, por ejemplo, permite a un mamífero amamantar, ¿cómo puede el compilador saber qué código se va a ejecutar para cada objeto (tipo de mamífero) particular? Esto es precisamente lo que se quiere evitar: cuando un programador escribe código que gestiona mamíferos no quiere saber qué mamífero concreto va a ser gestionado por su código; él sólo se preocupa de la semántica de la operación que está aplicando. Por tanto, cuando se usa el polimorfismo el compilador

no es capaz de determinar qué código va a tener que invocar al realizar una llamada a una función miembro.

En los lenguajes de programación no orientados a objetos la llamada una función hace que el compilador genere una llamada a una función concreta, y que el enlazador resuelva esta llamada asignándole una dirección de memoria donde va a estar almacenado el código a ejecutar. A esto se le conoce como **ligadura estática**. Los lenguajes de programación orientados a objetos, para dar soporte al polimorfismo de herencia, emplean la **ligadura dinámica**: cuando se envía un mensaje a un objeto (mamífero) el código que se ejecutará (la forma concreta de amamantar a las crías) no se determina hasta tiempo de ejecución (cuando finalmente se sabe qué mamífero concreto estamos gestionando). No obstante, el compilador se asegura que la función existe en la clase base y chequea los parámetros que se le pasan.

2.5.1 Otros tipos de polimorfismo

Aunque el polimorfismo que hemos descrito es posiblemente el más potente, y el que constituye la principal aportación de la programación orientada a objetos, no es el único tipo de polimorfismo existente. A este polimorfismo se suele conocer como "**polimorfismo de sustitución**", ya que permite sustituir una clase base por cualquiera de sus subclases.

Otro tipo de polimorfismo interesante es el **polimorfismo por sobrecarga** de funciones. Éste es la capacidad de emplear el mismo nombre para distintas funciones que tienen una misma semántica y, por tanto, se corresponden con una misma operación conceptual. Para identificar que función concreta se está invocando se emplean los distintos argumentos que se le pasan. C no posee polimorfismo por sobrecarga; por tanto, si desea hacer una función que calcula el valor absoluto de un número entero y otra que calcule el valor absoluto de un número real tendrá que implementarse una función "valorabsoluto_entero (int a)" y otra "valorabsoluto_real (float a)". Ambas funciones son distintas implementaciones de una misma operación conceptual, hallar el valor absoluto de un número. Por culpa de cómo esos números están representados dentro de un ordenador (los enteros y los números reales se representan de modos diferentes) cuando el programador tiene que realizar una operación sobre un dato, calcular su valor absoluto, tiene que pensar en el modelo del ordenador (representación de los números) y decidir y qué función invocar. Además,

deberá conocer los distintos nombres de cada una de las funciones. En resumen: la ausencia de este polimorfismo obliga al programador a gastar más energía comprendiendo el modelo del ordenador y no el modelo correspondiente al problema que desea resolver (donde hay una única operación de valor absoluto).

Los lenguajes de programación que soportan polimorfismo por sobrecarga, como Java, permiten definir varias funciones con un mismo nombre que se diferencian en los argumentos que se le pasan: "valorabsoluto (int a)" y "valorabsoluto (flota a)". Esto permite que cuando el programador necesita calcular un valor absoluto no tiene que preocuparse por cómo el ordenador representa ese dato (si es un entero o un real).

El **polimorfismo de coerción** permite que una operación aritmética pueda producir resultados distintos dependiendo del tipo de los operandos. Este tipo de polimorfismo sí está soportado en C (y en prácticamente cualquier lenguaje de programación de alto nivel). Para sumar, restar, multiplicar... datos enteros o datos reales la CPU de un equipo tiene que realizar operaciones completamente diferentes. Las operaciones también difieren si los datos enteros son de 32 o de 64 bits, al igual que sucede con los reales, y si los datos enteros son con signo o sin signo. Sin embargo, todas estas operaciones de la máquina se corresponden con una única operación conceptual: realizar una suma, una resta... y gracias al polimorfismo de coerción podemos emplear un único operador (+,-,*,/) para realizar las diversas operaciones que permiten a la CPU sumar los distintos tipos de datos que gestiona.

El **polimorfismo paramétrico** permite que un objeto pueda usarse uniformemente como parámetro en distintos contextos o situaciones. Las clases genéricas o paramétricas permiten definir clases de ámbito genérico que no se quieren especificar completamente hasta determinar un tipo de datos. En estas clases se definen todos los métodos que se necesitarán, pero el tipo de datos que manipulan se especifica con un parámetro en el momento en que se crean los objetos de la clase. Este polimorfismo paramétrico se utilizará ya en el tema 3 y será muy importante en los temas 5 y 6.