Tema 2

Conceptos y principios de la orientación a objetos

Índice de contenidos

1. Introducción

Vivimos en un mundo de objetos. Estos objetos existen en la naturaleza, en entidades hechas por el hombre, en los negocios y los diversos productos que usamos. Pueden ser clasificados, descritos, organizados, combinados, manipulados y creados. Por eso no es de extrañar que se proponga una visión orientada a objetos para la creación de software de computadora, una abstracción que modela el mundo de forma tal que nos ayuda a entenderlo mejor.

A finales de los 60 se propuso un enfoque orientado a objetos para el desarrollo de software, pero esta tecnología ha necesitado casi veinte años para ser ampliamente usada. Durante los 90, la POO (programación orientada a objetos) se convirtió en el paradigma para muchos desarrolladores de software.

Las tecnologías de objetos llevan a reutilizar, y la reutilización lleva a un desarrollo de software más rápido y a programas de mejor calidad. El software **orientado a objetos** es más fácil de mantener debido a que **su estructura** es **inherentemente poco acoplada**. Esto lleva a menores efectos colaterales cuando se deben hacer cambios y provoca menos frustración en el ingeniero del software y en el cliente. Además, los sistemas orientados a objetos son más fáciles de adaptar y escalan más fácilmente (por ejemplo: pueden crearse grandes sistemas ensamblando subsistemas reutilizables).

En este capítulo presentamos los principios y conceptos básicos que forman el fundamento para la comprensión de la tecnología de objetos.

2. Factores de calidad del software

Algunos de los aspectos que han puesto de manifiesto las limitaciones de la metodología clásica de desarrollo del software son: la evolución constante de las aplicaciones informáticas para adaptarlas a nuevos requerimientos (tanto tecnológicos como funcionales), la necesidad de reutilización para acortar tiempos de desarrollo, y la capacidad de producir software más fiable y con un mayor grado mantenimiento.

Para atender estas necesidades y hacer que las aplicaciones informáticas sean viables económicamente, es necesario facilitar la reutilización de programas minimizando el trabajo de hacer adaptaciones a las funcionalidades nuevas. Un prerrequisito para facilitar **la reutilización** es que el software desarrollado responda a ciertos criterios o **factores de calidad**.

El software básicamente es un mecanismo, que se usa para ejecutar rutinas de forma automática y de forma enormemente rápida a través de un ordenador. Pero aparte de estos aspectos, existen otros muy importantes como pudieran ser la fiabilidad con la que se ejecutan estas rutinas por ejemplo, lo fácil que es interactuar con el programa o la facilidad con la que se puede modificar o extender el programa para adecuarse a nuevos requisitos.

Aunque hay muchas medidas de la calidad de software, *la corrección, facilidad de mantenimiento, integridad y facilidad de uso* son algunos de los factores más importantes.

Hace 25 años McCall y Cavano [MCC78] definieron un conjunto de factores de calidad como los primeros pasos hacia el desarrollo de **métricas de calidad del software**. Estos factores evalúan el software desde las siguientes dimensiones:

- Operación del producto *(utilizándolo).*
- Revisión del producto *(cambiándolo).*
- Transición del producto (modificándolo para que funcione en un entorno diferente *(portándolo).*

Para cada una de estas categorías, McCall definen una serie **factores de calidad** como se muestra la siguiente figura:

Figura 1 Criterios de calidad de McCall

La siguiente tabla muestra un resumen con todos los **factores de calidad** del modelo de McCall.

Factores de calidad McCall		
Aspecto que trata	**Factor Calidad**	**Relacionado con …**
Operación del Producto	Corrección	Hasta donde satisface un programa su especificación y logra los objetivos propuestos por el cliente.
	Fiabilidad	Probabilidad de operación libre de fallos de un programa en un entorno determinado durante un tiempo específico.
	Eficiencia	Cantidad de recursos de computadora y de código requeridos por un programa para llevar a cabo sus funciones.
	Integridad	Grado en que puede controlarse el acceso al software o a los datos, por personal no autorizado.
	Facilidad de uso	Esfuerzo requerido para aprender un programa, trabajar con él, preparar su entrada e interpretar su salida.
Revisión del producto	Facilidad de Mantenimiento	Esfuerzo requerido para localizar y arreglar un error en un programa.
	Flexibilidad	Esfuerzo requerido para modificar un programa operativo.
	Facilidad de prueba	Esfuerzo requerido para probar un programa de forma que se asegure que realiza la función requerida.
Transición	Portabilidad	Esfuerzo requerido para transferir el programa desde un hardware y/o entorno de

del producto		sistemas de software a otro.
	Reusabilidad	Grado en que un programa (o partes de un programa) puede volverse a usar en otras aplicaciones.
	Facilidad de interoperación	Esfuerzo requerido para acoplar un sistema a otro

La calidad no es tema desligado de los aspectos más técnicos, de hecho son algunos de estos factores y su cumplimiento lo que impulsan determinadas mejoras en la forma de abordar la construcción de un sistema software.

3. El paradigma de orientación a objetos

Un **paradigma** de programación **hace referencia a como establecer, ver y resolver problemas (en nuestro caso de programación)**. El paradigma orientado a objetos establece el tipo de objeto como la unidad de representación y compresión de sistemas software complejos.

El termino orientado a objetos sé uso para referirse al enfoque de desarrollo de software que usaban lenguajes como ADA 95,C++, SmallTalk, etc. Hoy en día el **paradigma orientado a objetos** encierra una visión completa de la ingeniería de software.

Hay muchas formas de enfocar un problema utilizando una solución basada en software. El **modelado y diseño orientado a objetos** es una **manera estructurar un sistema**, en la que **se utilizan objetos para representar conceptos del mundo real**. El dominio del problema se representa mediante un conjunto de objetos con atributos y comportamientos específicos. Los objetos son manipulados mediante una colección de funciones llamados métodos y se comunican entre sí mediante un protocolo de intercambio de mensajes.

Según **Grady Booch**: *La POO es un método de implementación en el que los programas se organizan como colecciones de objetos cooperativos, donde cada uno de los cuales representa un instancia de alguna clase…* .

En esta definición se pueden distinguir tres cuestiones:

1. La POO utiliza objetos cooperativos, no algoritmos (programación estructurada), como bloques de construcción principales.
2. Visión cooperativa de objetos
3. Cada objeto es una instancia de una clase.

El concepto fundamental es el **objeto**, entendido como una **combinación estructural de datos y comportamiento** en una única entidad. De esta forma podemos definir un objeto como:

Un **objeto** es un módulo auto contenido de datos y operaciones que representa una entidad tangible o intangible con significado en el dominio del problema.

De manera general la programación orientada a objetos divide el proceso en dos partes:

1. Definición de clases (tipos de objetos) y sus relaciones.
2. Puesta en funcionamiento de los objetos mediante la instanciación y el intercambio de mensajes entre ellos.

El paradigma de **orientación a objetos** es atractivo debido a que **favorece la creación y reutilización de componentes independientes** lo que va unido a factores de calidad como son la reutilización y la extensibilidad, mantenimiento, etc. Además, los componentes de software derivados mediante el paradigma de objetos muestran características asociadas con el software de alta calidad, como la independencia funcional, la ocultación de información, etc.

Para entender la orientación a objetos, es necesario abordar una serie de conceptos que influyen en el diseño. Muchos son conceptos anteriores a la POO, pero que representan una serie de importantes ideas que fundamentan las ventajas de la orientación de objetos.

3.1. El principio de modularidad

La modularidad es la **posibilidad de subdividir una aplicación en piezas más pequeñas** (denominadas módulos), donde cada una de las cuales debe ser tan independiente como sea posible.

La programación modular se concibió como el desarrollo de programas a partir de acoplar pequeñas piezas de código, denominadas subrutinas, que al mismo tiempo se agrupaban en módulos cuando éstas trabajaban con los mismos datos.

El cumplimiento de muchos de los criterios de calidad vistos en el apartado anterior depende en gran manera de la modularidad de las estructuras de representación, tanto de código como de datos.

En este apartado profundizaremos en esta definición informal, y nos centraremos en aquellas propiedades y características que debe tener un método para garantizar la modularidad. Además discutiremos los criterios que se tienen que cumplir para que una estructura de representación se pueda calificar de modular y, por lo tanto, permita una metodología de desarrollo modular.

3.1.1. El modulo y su estructura

Pensemos en un gran bloque monolítico de código. Desde el punto de vista del mantenimiento esta forma de codificar presenta numerosos problemas: la depuración de errores se vuelve extremadamente difícil y la comprensión del código se vuelve virtualmente imposible. La solución es dividir el bloque en otros bloques más pequeños, conocidos como **módulos.**

La palabra modulo tiene diversas acepciones y significados. Un módulo es una unidad claramente definida y manejable, con una interfaz definida. En otras palabras, es la unidad mínima con sentido propio en un sistema software. No obstante vamos a ver una serie de definiciones de la palabra modulo.

Stevens, Myers y Constantine [1974] realizan un primer intento de definición. Ellos definen un módulo *"como un conjunto de una o más instrucciones contiguas agrupadas con un nombre, por el cual otras partes del sistema pueden invocarlas y que preferiblemente poseen su propio conjunto de variables".*

En otras palabras un módulo es un bloque de código que puede ser invocado como un procedimiento, función o método.

Diccionario Webster ."*Una pieza independiente de software que es parte integrante de una u otras de mayor envergadura. Los módulos se compilan normalmente de forma separada e independiente y proporcionan un mecanismo de abstracción o de ocultación de información de forma que su implementación pueda ser cambiada sin afectar a otros módulos."*

Un módulo **se caracteriza fundamentalmente por su interfaz y por su implementación**. **Parnas** define el módulo como *"un conjunto de acciones denominadas, funciones o sub-módulos que corresponden a una misma abstracción, que comparten un conjunto de datos comunes llamados atributos. Las acciones o funciones de un módulo que son susceptibles de ser llamadas desde el exterior se denominan primitivas o puntos de entrada del módulo".*

El concepto de modulo ha evolucionado a lo largo del tiempo junto con los lenguajes de programación. Desde los subprogramas (funciones y procedimientos), pasando por lo tipos abstractos de datos hasta el concepto de clase. El módulo ha ido cambiando de forma. Pero en todos los casos su fin es el de proporcionar unidades auto contenidas, independientes y que funcionen como si se permiten el símil como cajas negras.

Cada módulo tendrá un significado específico y debe asegurarse que **cualquier cambio en su implementación no afecte a su exterior** (o al menos, que afecte lo mínimo posible). De igual modo, se debe intentar asegurar que los errores posibles, condiciones de límites o frontera, comportamientos erráticos, no se propaguen más allá del módulo (o como máximo a los módulos que estén directamente relacionados).

La **Figura 2** muestra un esquema de las partes más importantes de un módulo idealizado desde el punto de vista del tipo abstracto de datos o clase.

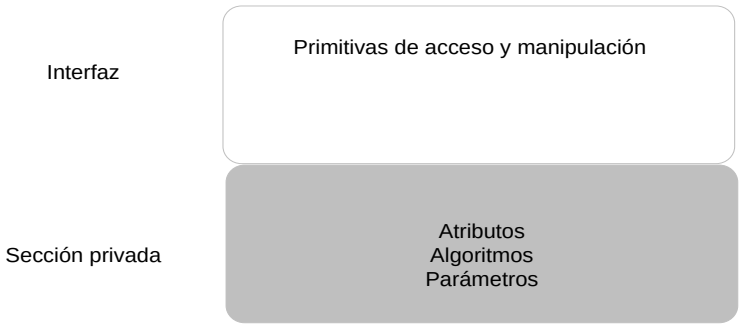


Figura 2 Estructura de un módulo

3.1.2. Características de la programación modular

La modularización es una **técnica de división** como estamos viendo, que permite al programador resolver un problema, dividiéndolo en sub-problemas más pequeños, de forma que se puedan resolver independientemente unos de otros, para después combinarlos.

3.1.3. Tamaño del modulo

Al dividir u organizar de forma modular se deberá decidir la envergadura del módulo. Existen módulos grandes y módulos pequeños, pero una característica esencial es que el módulo sea auto contenido, completo, cohesivo y con el menor acoplamiento posible respecto a otros módulos, de forma que se pueda integrar con otras partes y que se pueda considerar para su corrección de forma aislada.

3.1.4. Ocultación de la información

En la etapa de diseño de un módulo, es imprescindible especificar el conjunto de las propiedades del módulo que constituirán la información a la cual tendrán acceso los otros módulos y a cuáles no. Estas propiedades las podemos dividir en dos tipos, propiedades públicas y propiedades privadas:

- Las **propiedades públicas** son aquellas que son visibles desde fuera del módulo tanto para los usuarios como para otros módulos .Serán la interfaz del modulo
- En cambio, las **propiedades privadas** son aquéllas que son internas al módulo, por lo que su visibilidad es exclusivamente dentro del módulo y estará oculta al exterior.

3.2. Diseño de módulos. Acoplamiento y cohesión

Aunque el diseño modular persigue la división de un sistema grande en módulos más pequeños y manejables, **no siempre esta división es garantía de un sistema bien organizado**. Muchos aspectos de la modularización pueden ser comprendidos solo si se examinan módulos en relación con otros. Los **módulos deben diseñarse con los criterios de acoplamiento y cohesión.** El primer criterio busca la menor dependencia posible entre módulos y el segundo busca que todo lo agrupado bajo el modulo obedezca al mismo propósito. En este sentido **Booch** define la modularidad como :"*la propiedad de un sistema que ha sido descompuesto en un conjunto de módulos cohesivos y débilmente acoplados".*

Myers[1978] define **cohesión y acoplamiento** de la siguiente forma:

- El **acoplamiento** entre módulos como el **grado de interdependencia entre dos módulos.**
- La **cohesión** como el **grado de interacción dentro de un módulo.**

Veremos a continuación estos dos conceptos tan importantes en el diseño de módulos

3.2.1. **Acoplamiento**

El acoplamiento es una **medida de interdependencia entre módulos.** Se dice que existe alto acoplamiento cuando el cambio de un modulo fuerza al cambio de los que dependen de él. El objetivo es hacer que está interdependencia sea mínima.

En la siguiente Figura (Figura 3) el módulo C1 tienen un acoplamiento relevante con C10, si un cambio en C10 implica un cambio en C1.

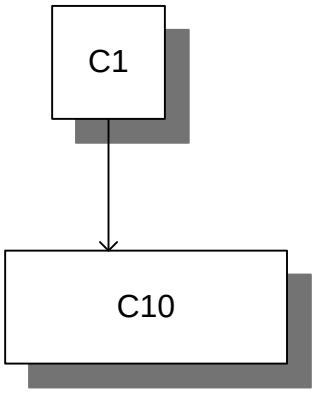


Figura 3 Interdependencia entre módulos

Como el acoplamiento se puede producir sólo en las relaciones entre módulos, cuantas más relaciones tenga un módulo, mayor probabilidad habrá que se produzcan acoplamientos. Por lo tanto podemos extraer las siguientes conclusiones o reflexiones:

- Cuantas menos conexiones existan entre módulos, menos oportunidad habrá para que se produzca el «efecto onda», es decir, que un defecto en un módulo pueda afectar a otro.
- Cuanto menos acoplados estén los módulos menos implicaciones o efectos colaterales podrán tener los cambios realizados.

A la vista de los anterior podría deducirse que una buena medida de diseño que evita el acoplamiento sería la de **limitar al máximo las conexiones entre módulos**. Esto en parte es cierto, ya que a menor número de relaciones menor probabilidad de verse afectado por cambios en otros módulos, pero muchas veces es inevitable y es necesario que un módulo se comunique con otros muchos y no necesariamente aumenta su acoplamiento. Para entender esto hay que considerar además, el grado de exposición que tiene un módulo respecto a los detalles internos de otro.

Pensemos en dos módulos M1 y M2. Supongamos que M2 es un módulo que representa un almacén de medidas de temperatura, y ofrece capacidades a otros módulos para que puedan ser extraídas en orden a como se obtuvieron. Y supongamos que M1es un módulo estadístico que opera sobre estas medidas a través de M2. Imaginemos que M1 "conoce" el tamaño del array que tiene M2 para guardar las medidas. Si este tamaño cambiase en M2, el cambio afectaría de manera irreversible a M1. Esto es debido a que M1 tiene un fuerte acoplamiento con M2, no sólo por su relación con M2 sino también por su exposición a los detalles internos de M2 de cómo funciona. La conclusión es que el grado de acoplamiento entre módulos no necesariamente está solamente en función del número de conexiones entre módulos.

El objetivo por tanto es obtener módulos **débilmente acoplados.**

Un acoplamiento bajo indica un sistema bien particionado y puede obtenerse de dos maneras:

- *Reduciendo el número de relaciones*: Cuanto menos conexiones existan entre módulos, menor será la posibilidad del efecto en cadena (un error en un módulo provoca otro colateral en otro modulo).

- *Debilitando el nivel de dependencia en las relaciones necesarias*: Ningún módulo debería depender de los detalles internos de implementación de cualquier otro. Lo único que tiene que conocer un módulo de otro es como invocarlo y como interpretar el resultado obtenido.

La siguiente tabla muestra la clasificación de acoplamientos entre módulos y su grado de acoplamiento desde un punto de vista de la programación estructurada.

Tipo de acoplamiento	Grado de acoplamiento	Facilidad de mantenimiento
Por contenido	Alto (fuerte)	Bajo
Común	|	|
De control	|	|
Por (Estampado)	|	|
Datos	↓	↓
	Bajo (débil)	Alto

Las ventajas de un sistema débilmente acoplado son muchas. Tal como define **Booch**, un sistema modular débilmente acoplado facilita:

- La sustitución de un módulo por otro, de modo que los módulos afectados por un cambio serán menos.
- El seguimiento de un error y el aislamiento del módulo defectuoso que produce ese error.

3.2.2. **Cohesión**

Otro medio para evaluar la partición en módulos (además del acoplamiento) es observar como las actividades dentro de un módulo están relacionadas unas con otras; este es el criterio de cohesión.

La cohesión tiene que ver con el grado de relación que tienen los elementos internos de un módulo, o más generalmente, la **forma en la que agrupamos elementos en una unidad conceptual de mayor nivel**. Por ejemplo, la forma en la que agrupamos funciones en una librería, o la **forma en la que agrupamos métodos en una clase**, o la forma en la que agrupamos clases en una librería, etc. Su objetivo es organizar los elementos de tal manera que los que tengan más relación a la hora de realizar una tarea pertenezcan al mismo módulo, y los elementos no relacionados figuren en módulos separados.

La cohesión también se puede definir corno la **medida de la relación funcional de los elementos de un módulo**; entendiendo por elementos, tanto la sentencia o grupo de sentencias que lo componen, como las definiciones de datos o las llamadas a otros módulos. Idealmente, un módulo coherente sólo debe hacer una única cosa.

El **objetivo es diseñar módulos con una alta cohesión**, cuyos elementos estén fuerte y genuinamente relacionados unos con otros.

Myers define siete categorías o niveles de cohesión. La siguiente tabla muestra estos grados de cohesión: baja cohesión (no deseable) y alta cohesión (deseable).

Tipo de cohesión	Grado de cohesión	Grado de mantenimiento
Por coincidencia	Bajo	Bajo
Lógica	↑	↑
Temporal	↑	↑
Procedimental	↑	↑
Por comunicaciones	↑	↑
Secuencial	↓	↓
Funcional	↓	↓
	Alto	Alto

Tabla. Clasificación de cohesión de módulos

4. Tipos de datos y Tipos abstractos de datos

Todos los lenguajes de programación soportan el concepto de tipo de datos. Por ejemplo, el lenguaje de programación **C** soporta los tipos int, float, char, así como los tipos: colección y estructuras (struct).

Un tipo de datos define un conjunto de valores posibles y un conjunto de operaciones definidas para ese tipo de valores.

Por ejemplo un tipo entero define un conjunto de datos posibles (números enteros negativos y positivos) además de las operaciones permitidas sobre ellos (las operaciones aritméticas).

Los **tipos abstractos de datos** (TAD), vistos en el siguiente apartado, extienden la función de los tipos de datos tradicionales .Y tienen la capacidad de ocultar detalles internos de su implementación o funcionamiento. Esta capacidad de ocultar la información permite el desarrollo de componentes de software reutilizables y extensibles.

El tipo constituye así la forma en la cual un desarrollador modela las entidades de la realidad que intenta representar y manipular y para el compilador representa la manera de reservar memoria y saber cómo manipular determinados valores. Los lenguajes modernos suelen permitir definir al programador sus propios tipos de datos permitiendo así una mayor flexibilidad a la hora de representar entidades.

5. De los módulos a los tipos abstractos de datos.

Se ha ido avanzando en la independencia de conjuntos de instrucciones para hacerlas no solamente una unidad funcional independiente, sino que esta una unidad funcional sea cada vez de mayor envergadura. Los procedimientos y funciones pueden considerarse las primeras aproximaciones a la programación modular, los subprogramas (funciones o procedimientos) pueden ser vistos como módulos. Posteriormente el tipo abstracto de datos ha constituido un avance significativo en la construcción de unidades modulares más completas al unir estructuras de datos y funciones en una misma unidad, permitiendo reducir el acoplamiento y aumentando la cohesión.

Una **idea fundamental** en la abstracción de los tipos abstractos de datos es la **separación** de la **especificación** del tipo y su **implementación interna**. Ya que si la representación interna se oculta, el modulo que llama o que usa esta unidad funcional tendrá un acoplamiento muy bajo.

Para ver más en detalle esto, vemos que en esencia un TAD es un tipo de dato que consta de datos y operaciones definidas para estos datos.

T.A.D= Datos+operaciones (funciones y procedimientos)

En el siguiente ejemplo se muestra un TAD que ofrece la característica de ocultamiento y encapsulación estando compuesto de dos partes: interfaz pública e implementación interna.

Ejemplo de un TAD. Definición e interfaz

```
TAD ListaCrtlProcesos:

Datos internos

      Lista: Proceso

Operaciones accesibles

AñadirProc ( Proceso)

Proceso sacarProc ( )

Proceso obtenerPrimerProceso();
```

Implementación interna

```
AñadirProc ( Proceso){

Lista.add(Proceso);

}

Proceso sacarProc (

(Proceso)Lista.obtenerPrimerProceso();

 )
```

Entre las ventajas que ofrecen los tipos abstractos de datos podemos destacar las siguientes:

- Permiten una **mejor conceptualización** del mundo real. Las estructuras de datos complejas como son los TAD permiten mejorar la representación, mejorando con ello la comprensión.
- Separa la implementación de la especificación con lo que **disminuye el acoplamiento.**
- Favorece el **diseño cohesivo** de módulos puesto que obliga a plantear de manera conjunta datos y operaciones.
- **Aumenta la reusabilidad** de unidades funcionales puesto que estas se han concebido como independientes, no acopladas y cohesivas.

6. Clases y Objetos

La **clase** unifica los principios del diseño modular y la definición de tipo de datos. **La clase en los lenguajes orientados a objetos es la unidad mínima de descomposición funcional del software (módulo).**

Desde el punto de vista de los tipos de datos. Una clase no es un tipo abstracto de datos compuesto por datos y operaciones. Desde el punto de vista del diseño y la programación modular, una clase es un módulo. La clase lleva al límite las ventajas de ofrecidas por los tipos abstractos de datos: bajo acoplamiento y alta cohesión. Desde un punto de vista cognitivo una clase representa una abstracción del mundo real, caracterizada por un conjunto de atributos y por el conjunto de operaciones que admite.

En principio una clase puede representar casi cualquier cosa, una aproximación de lo que representa una clase sería la siguiente:

Una clase representa una generalización de una serie de objetos que tienen el mismo conjunto de características.

Como vemos, íntimamente relacionado con el concepto de clase, está el de objeto. El objeto representa una instancia particular, es decir un ejemplar especifico de una clase (Por ejemplo de la clase archivos, un SSOO puede tener los siguientes ejemplares: cab234.txt (3Mb), runtime.exe(2Kb))

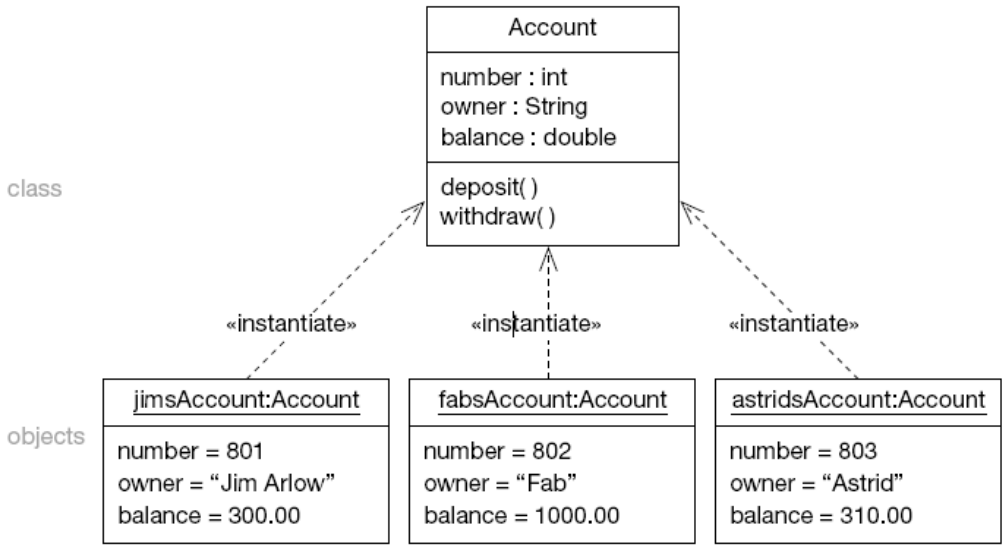
A veces, es difícil diferenciar los conceptos de clase y objeto. Para ver mejor esto con un ejemplo, cuando hablamos de un objeto *cuenta corriente*, todo el mundo entiende que nos referimos a un concepto que sirve para reflejar el dinero que una persona tiene en el banco, a través de un saldo, un código de identificación, y una serie de titulares. En este caso concreto, cuenta corriente se debe concebir como una clase de objetos, ya que representa un grupo o conjunto de entidades con ciertas características comunes (número, saldo, titulares).**La clase describe** como es una cuenta corriente para ser considerada como tal.

Cuando hablamos de la cuenta corriente 6773-0100-89-2223873, con saldo 10.000 euros, titulares Juan Antonio y Rosa , nos referimos a una cuenta en concreto. En este caso, esta *cuenta corriente* se debe entender como un objeto particular o más concretamente en programación, como una instancia de la clase *Cuenta Corriente* que identifica un miembro individual y concreto de la clase de objetos descritos.

La **clase describe** un grupo de objetos que comparten un mismo conjunto de características, mientras que **el objeto es un representante particular** de este grupo.

Así podemos ver que una clase es una descripción generalizada, como por ejemplo una plantilla, que describe una colección de objetos similares. Por definición, todos los objetos que pertenecen a una clase deben tener sus atributos y las operaciones disponibles para la manipulación de los atributos.

La relación entre clases y objetos es una relación de pertenencia. Así un objeto se dice que pertenece a una clase. Enfocado desde un punto de vista más técnico el siguiente ejemplo muestra como **un objeto instancia a una clase** o dicho de otra forma una variable de tipo clase recibirá como valor un objeto concreto.



Ejemplo de instancias de la clase cuenta

Objetos

El manual de referencia de UML (*The UML Reference Manual* [Rumbaugh] define un objeto como: ***"Una entidad discreta con limites perfectamente definidos, que encapsula un estado y un comportamiento instancia de una clase".***

Según (Booch). "***Un objeto es un conjunto cohesivo de datos y funciones.*** "

Cuando hablamos de objetos, hacemos referencia a un conjunto de datos concretos que pueden cambiar, mientras que cuando hablamos de clase hacemos referencia a como es la estructura que representa a esos objetos y que se usa como plantilla para crearlos. En este sentido **la clase es un concepto estático**, mientras que **el objeto es un concepto más dinámico**.

En programación orientada a objetos una instancia se produce con la creación de un objeto perteneciente a una clase . A diferencia de una variable convencional, **un objeto no se crea simplemente definiendo una variable hay que llamar a un método especial denominado constructor (visto más adelante) que indica la forma de crear el objeto**. Para ver esto en la siguiente línea de código Java se crea un objeto cuenta corriente.

```
CuentaCorriente cc01= new CuentaCorriente(Juan,Casas Fuente,5000);
```

En nuestro caso `cc01` es una **instancia** de la clase `CuentaCorriente`**.** O dicho de otra forma `cco1` es un objeto de la clase `CuentaCorriente` En la documentación utilizaremos instancia, objeto o ejemplar indistintamente.

Representación conceptual y representación a nivel de lenguaje de programación de una clase.

Conceptualmente y de forma general una clase se caracteriza por un nombre de clase, un conjunto de atributos y un conjunto de métodos. La siguiente figura (Figura 4) muestra una representación simbólica de una clase en la notación **UML** *(Unified Modeling Language ver anexo I).*

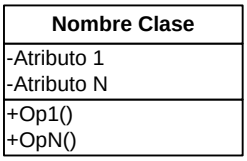


Figura 4 . Representación UML de una clase

A nivel del lenguaje de programación la clase debe representarse con los elementos propios que proporcionan los lenguajes de programación. En concreto para Java y C++ seria de la siguiente forma.

Representación y especificación de una clase en Java:

```
public class Ordenador {
    //atributos

    String marca;
    String modelo;
    String procesador;
    int memoria;
    float frecuencia;
    boolean encendido;

    //metodos
    public Ordenador(){}
    public Ordenador (String mar,String mod,String micro,int men,float fre){
    marca=mar;
    modelo=mod;
    procesador=micro;
    memoria=men;
    frecuencia=fre;
    }
    public void encender(){
        if (!encendido)
            encendido=true;

    }
}
```

Representación y especificación de una clase en C++

Archivo Ordenador.h

```
class Ordenador
{

    private:
       char * marca;
       char * modelo;
       char * procesador;
       int memoria;
       float frecuencia;
       bool encendido;
   public:

              Ordenador();
              Ordenador(char *mar,char *mod,char *micro,int men,float fre);
              void encender();

              ~Ordenador();
};
```

Archivo Ordenador.cpp

```
#include "ordenador.h"


Ordenador::Ordenador()
{

}
Ordenador::Ordenador(char *mar,char *mod,char *micro,int men,float fre)
{
 marca=mar;
 modelo=mod;
 procesador=micro;
 memoria=men;
 frecuencia=fre;
}
 void Ordenador::encender(){
   if (!encendido)
     encendido=true;

}

Ordenador::~Ordenador()
{

}
```

Como se puede ver el mismo concepto es representado de forma diferente en los diferentes lenguajes. Pero ambas representaciones comprenden elementos comunes como son atributos y métodos.

6.1. Atributos

Cualquier objeto o entidad existente se puede describir a partir de una serie de atributos .Una clase puede tener un número arbitrario de atributos que definen las características más relevantes en el dominio del problema y que identifican unívocamente el tipo de objetos que está representado.

Un **atributo** es una característica o propiedad de una entidad

Un atributo toma valor en un dominio de información, que define el tipo de valores válidos en la interpretación de ese atributo .Por ejemplo, suponga que una clase

denominada **Coche** con el atributo **color.** El dominio de valores de **color** podría ser **(blanco, negro, plata, gris, azul, rojo, amarillo, verde).**

6.2. Estado de un objeto.

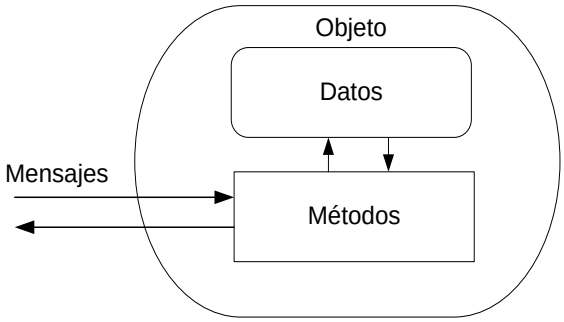
El **estado** de un objeto es el valor de sus atributos observados en un instante de tiempo concreto.

En un instante de tiempo determinado un objeto tendrá una serie de valores en todos y cada uno de sus atributos, al conjunto de valores que toman los atributos de un objeto en un instante de tiempo dado se denomina **estado del objeto**. Como es evidente el estado de un objeto puede variar a lo largo del ciclo de vida del objeto en el programa.

El concepto de estado define los valores transitorios y estables por los que puede pasar un objeto, de forma que se puede determinar la corrección de un sistema a partir del conjunto de estados alcanzado por los objetos del sistema.

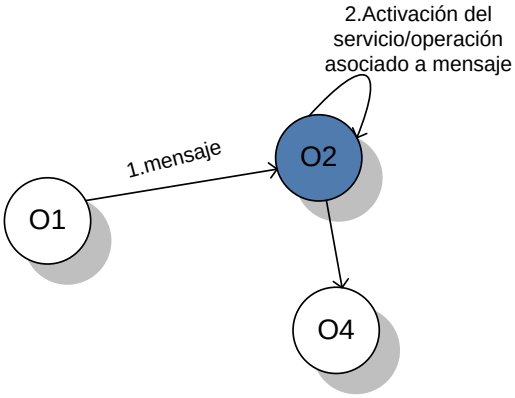
6.3. Mensajes ,operaciones y métodos

Cuando se diseña una clase, **la estructura interna y los detalles de implementación se ocultan**, con la posibilidad de intercambiar mensajes como la única posibilidad de conexión.



En un modelo de clases y sus objetos, los **métodos** (operaciones o servicios) asociados a estas clases **describen el comportamiento asociado a un objeto**.

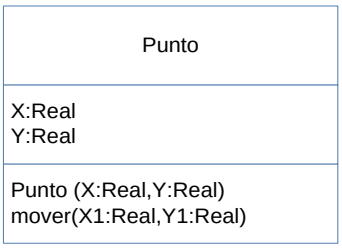
Cada método tiene un nombre y un cuerpo que realiza la acción o comportamiento asociado con el nombre del método. Un método dentro de un objeto se activa por un **mensaje** que se envía por otro objeto al objeto que contiene el método.



6.3.1. Comportamiento de un objeto

Cuando un objeto recibe algún mensaje, siempre inicia algún tipo de procesamiento. Cada una de las operaciones que tiene la clase proporciona una representación del comportamiento de los objetos a los que representa. Por eso **el conjunto de operaciones** que admite un objeto o clase de denomina **comportamiento del objeto**, y está íntimamente relaciona con la funcionalidad que puede realizar.

Por ejemplo, dada la siguiente clase llamada ***Punto*** que representa puntos en un espacio bidimensional y que podemos representar de la siguiente forma:



La consecuencia de la existencia de la operación *mover(X1:Real,Y1:Real)* es que la clase punto ha sido diseñada para recibir un estimulo (llamaremos a este estímulo mensaje).Cada vez que un objeto recibe un mensaje/estimulo inicia un comportamiento. El comportamiento especificado para un objeto instancia de esta clase es moverse a un desplazamiento determinado. El método para moverse vendrá implementado en el método.

6.3.2. Método

Un **método** es la especificación e implementación de un servicio u operación que puede ser requerido a cualquier objeto de la clase. El método muestra, o define, parte del comportamiento de los objetos representados en la clase.

Un *método* se implementa en una *clase* de objetos, e indica cómo debe actuar el objeto cuando recibe el mensaje vinculado a ese método. Por ejemplo el método para mover un punto es incrementar en X1 e Y1 las coordenadas del punto. A su vez un método también puede enviar mensajes a otros objetos.

Los métodos describen *el comportamiento* asociado a un objeto. Debemos considerar que la ejecución de un método puede conducir a un cambio de estado del objeto.

6.3.3. Definición de un Método

Un método se define por un nombre, por los argumentos o parámetros de entrada que necesita y por el valor de retorno que se obtiene al ejecutar el comportamiento que implementa. Estos elementos (valor de retorno, nombre y parámetros) de un método se conocen como **prototipo del método o de una función miembro.**

```
NombreMetodo (parametro1 Tipoparam1..., parametroNTipo paramN)→Tipo de retorno
```

La implementación define el comportamiento del objeto ante la llamada del método

```
NombreMetodo (parametro1Tipo param1..., parametroNTipo paramN)→Tipo de retorno

<Implementación del método>
```

Podemos distinguir **de manera informal** varios tipos de métodos dentro de una clase:

- **Métodos constructores y destructores de objetos.**
- **Métodos de acceso a propiedades: métodos set (establecer) y get (obtener)**
- **Métodos de servicio (en general).**

La primera categoría representa los métodos dedicados a cómo crear y eliminar objetos en. Veremos más adelante estos dos métodos en profundidad.

La segunda categoría corresponde un **tipo de métodos muy comunes** en cualquier clase. Los métodos "set y get" están diseñados para obtener y establecer el valor de los atributos de un objeto. Un ejemplo típico de este tipo de métodos lo podemos encontrar en la clase *Ordenador* anteriormente comentada. Para esta clase se podría haber diseñado los siguientes métodos set y get.

Ejemplo de **métodos set y get** para la clase Ordenador.java

```
public class Ordenador {
    private String marca;
    private String modelo;
    private String procesador;
    private int memoria;
    private float frecuencia_micro;
    private boolean encendido;

    public Ordenador(){}
    public Ordenador (String mar,String mod,String micro,int men,
    float fre){
    marca=mar;
    modelo=mod;
    procesador=micro;
    memoria=men;
    frecuencia_micro=fre;
    }
```

```
    public void encender(){
        if (!isEncendido())
            setEncendido(true);

    }


    public String getMarca() {
        return marca;
    }


    public void setMarca(String marca) {
        this.marca = marca;
    }


    public String getModelo() {
        return modelo;
    }

    public void setModelo(String modelo) {
        this.modelo = modelo;
    }


    public String getProcesador() {
        return procesador;
    }


    public void setProcesador(String procesador) {
        this.procesador = procesador;
    }

    public int getMemoria() {
        return memoria;
    }

    public void setMemoria(int memoria) {
        this.memoria = memoria;
    }

    public float getFrecuencia() {
        return frecuencia;
    }

    public void setFrecuencia(float frecuencia_micro) {
        this.frecuencia = frecuencia_micro;
    }

    public boolean isEncendido() {
        return encendido;
    }

    public void setEncendido(boolean encendido) {
        this.encendido = encendido;
    }
}
```

Es muy importante asignar los valores de los atributos de un objeto siempre por medio de los métodos de acceso correspondientes, ya que permiten modificar la estructura interna de la clase sin modificar las interfaces de comunicación con las otras clases y, por lo tanto, no habrá que modificarlas cuando se realicen cambios que sólo afecten a la estructura interna. De esta manera, podemos mantener el principio de encapsulamiento y ocultación.

Dentro de la última categoría denominada métodos de servicio agrupamos todos aquellos métodos que definen el comportamiento restante de la clase.

6.3.4. Mensajes

Los objetos generan un comportamiento enviándoles mensajes, esto es lo que se denomina colaboración.

Cuando se crea un programa orientado a objetos, los objetos recibirán, interpretarán y responderán a mensajes de otros objetos. Usando la terminología presentada en la sección precedente, un mensaje estimula la ocurrencia de cierto comportamiento en el objeto receptor.

6.4. Ocultación de información .Visibilidad de los atributos y métodos

Al definir una clase se debe establecer **el grado de visibilidad u ocultamiento** que va a tener su estructura respecto al exterior. La visibilidad define por tanto el grado de acceso que tendrá una clase respecto a otra.

Para poder establecer claramente la visibilidad de los atributos y métodos, los lenguajes de programación normalmente definen tres niveles de acceso distintos.

- **Público (public):** cualquier clase puede acceder y utilizar cualquier atributo o método declarado como público de otra clase.

- **Privado (private):** ninguna clase puede acceder a un atributo o método declarado como privado ni utilizarlo.

Existe un tercer nivel denominado **Protegido (protected)**: cualquier clase heredera puede acceder a cualquier atributo o método declarado como protegido en la clase padre y utilizarlo.

Por ejemplo la clase java ordenador.

```
public class Ordenador {
    private String marca;
    private String modelo;
    private String procesador;
    private int memoria;
    private float frecuencia_micro;
    private boolean encendido;
```

```
    //…

    public String obtenerMarca() {
        return marca;
    }


    public void asignarMarca(String marca) {
        this.marca = marca;
    }
}
```

En la clase ordenador del ejemplo anterior sólo se puede acceder al atributo marca a través de los métodos obtenerMarca() y asignarMarca() , ya que por defecto todos los atributos son **prívate(privados) y no pueden ser accedidos desde fuera.**

6.5. Propiedades de instancia y propiedades de clase

Como ya hemos mencionado el concepto de clase y objeto aunque están íntimamente relacionados, guardan ciertas diferencias importantes.

A la hora de definir los atributos de una clase se puede especificar que **determinados atributos y sus valores sean de clase y no de cada objeto,** significa que serán compartidos y comunes a todos los objetos de una clase y que su valor no depende del objeto en particular. Por ejemplo en la case ordenador podríamos definir periodo de garantía como nuevo atributo para la clase, pero si nos fijamos bien este atributo es compartido por todos los tipos y modelos de ordenadores, luego se podría pensar en el él como un *atributo de clase,* no de instancia.

Ejemplo de una **atributo de Clase**

```
public class Ordenador {
    static int periodo_garantia;
    private String marca;
    private String modelo;
    private String procesador;
    private int memoria;
    private float frecuencia_micro;
    private boolean encendido;

}
```

Los lenguaje de programación (java y C++) permiten esta construcción através del modificador **static.**

6.6. Los métodos constructor y destructor de una clase

A diferencia de una variable convencional, cuando se define una variable y como tipo una clase, está variable no es un objeto hasta que no se construya (necesita tener asignados una serie de valores en sus atributos). El compilador necesita saber cómo construir y posteriormente destruir los objetos de una clase.

Los métodos constructores y destructores de una clase definen como construir un objeto y como destruirlo.

Constructor de clase

El **objetivo** del constructor es el de **inicializar** un objeto a través de sus atributos cuando éste es creado. El siguiente ejemplo muestra la creación de un objeto *Circulo* con un determinado radio.

Java
```
Circulo c1= new Circulo(10);
```

C++
```
Circulo *c1= new Circulo(12);
```

Cuando se crea un objeto (instanciación), se pasan los valores de los parámetros al constructor utilizando una sintaxis similar a una llamada a un método.

Un constructor es el mecanismo que disponen las clases para inicializar los objetos.

Cuando se ejecuta la instrucción anterior el compilador reservará memoria para albergar un objeto instancia de la clase *Circulo* y llamará al código asociado el constructor de esta clase.

Un constructor se ejecuta cuando se crea una instancia de una clase. El constructor tiene como propósito la **inicialización de los datos miembro o atributos** del objeto de la clase.

Podemos ver un **constructor** como un procedimiento con las siguientes **características**

- Es un método obligatorio de la clase
- Tiene el mismo nombre que la clase
- Puede tener cualquier número de argumentos (incluso ninguno).
- Puede haber más de un constructor, es decir distintas maneras de crear un objeto

Formato general de un constructor de clase:

```
    NombleClase(lista de argumentos)
        {
         // Instrucciones de inicialización
        }
```

Ejemplos de constructores en los distintos lenguajes:

Java
```
public class Contador {
  int n;
  Contador(){n=0;};
  Contador( int v_ini){
  n=v_ini;
  }
}
```

Ejemplo de constructors en C++

```
class Contador
{
    private:
            int n;
       public:

             Contador(){ n=0;};
             Contador (int v_ini){n=v_ini ;}
             ~Contador();
};
```

La construcción de un objeto consta de tres etapas:

- Se reserva espacio en memoria para la estructura de datos que define la clase.
- Se inicializa los campos de la instancia con los valores por defecto. Garantiza que cada atributo de una clase tenga un valor inicial antes de la llamada al constructor
- Se inicia el código definido por el constructor de clase.

Tipos de constructores

Generalmente existen **dos tipos de constructores** para una clase

- Constructor por defecto (sin argumentos/vacio).
- Constructor con argumentos

Constructor por defecto es aquel que inicializa los atributos de un objeto a su valor por defecto.

Constructor con argumentos es un constructor que recibe a través de argumentos los valores de inicialización de los atributos del objeto.

En el ejemplo visto anteriormente de la clase contador C++ vemos estos tipos de constructores.

Ejemplo en C++ de constructor por defecto y con argumentos

```
class Contador
{
    private:
            int n;
       public:

             Contador(){ n=0;}; //constructor por defecto
             Contador (int v_ini){n=v_ini ;} //constructor con argumentos
             ~Contador();
};
```

Instanciación de la clase Contador

```
#include <cstdlib>
#include <iostream>
#include "contador.h"
using namespace std;

int main(int argc, char *argv[])
```

```
{

  Contador *cont1= new Contador();
  Contador *cont2=new Contador(2);
  cont1->incrementar();
  cont2->incrementar();
  cout<<cont1->getValor();
  cout<<cont2->getValor();
   system("PAUSE");
}
```

Destructor

Si lo que queremos es eliminar un objeto previamente creado con un constructor habrá que llamar directa o indirectamente al método denominado destructor. Aquí también existen diferencias notables entre los lenguajes. Mientras C++ requiere una llamada explicita al destructor del objeto, otros lenguajes como Java realizan esta tarea de forma automática a través de un proceso especial llamado recolector de basura ***(garbage collector)*** liberando al programador de esta responsabilidad. Aunque la destrucción se puede realizar explícitamente si se desea.

La primera aproximación si se realiza bien garantiza la liberación de memoria y el momento de la liberación. La segunda sin embargo libera la memoria pero no ofrece garantías de cuando se hará.

```
poligono p1 = new Poligono(10,20,40,15,50,20);
poligono p2 = new Poligono(10,30,20,40,50,60);
p1 = p2;  //El objeto p1  deja de estar referenciado , puede ser liberado por el sistema.

p1 = null;// null: palabra reservada; se utiliza para indicar referencia nula.

El objeto Poligono(10,30,20,40,50,60);       deja de estar referenciado (identificado), puede ser liberado por el sistema.
```

Destrucción de objetos

```
Java
cont=null;

C++
delete cont;
```

Las ventajas que se obtienen de una liberación a través de recolector de basura son la de una liberación más segura de la memoria reservada, ya es frecuente que los programadores no borren explícitamente los objetos creados (por descuido), así se consigue un ahorro significativo en el espacio utilizado.

Ejemplo de **destrucción explicita en C++** del objeto Contador

```
#include <cstdlib>
#include <iostream>
#include "contador.h"
using namespace std;
```

```
int main(int argc, char *argv[])
{

  Contador *cont1= new Contador();
  Contador *cont2=new Contador(2);
  cont1->incrementar();
  cont2->incrementar();
  cout<<cont1->getValor();
  cout<<cont2->getValor();

  delete cont1;
  delete cont2;
  system("PAUSE");}
```