

ENTORNOS DE DESARROLLO

1. Desarrollo de software

Toda aplicación informática, ya sea utilizada en un soporte convencional (como un ordenador de sobremesa o un ordenador portátil) o sea utilizada en un soporte de nueva generación (por ejemplo, dispositivos móviles como un teléfono móvil de última generación o una tableta táctil PC), ha seguido un procedimiento planificado y desarrollado detalle por detalle para su creación. Este irá desde la concepción de la idea o de la funcionalidad que deberá satisfacer esta aplicación hasta la generación de uno o varios ficheros que permitan su ejecución exitosa.

Para convertir esta concepción de una idea abstracta en un producto acabado que sea eficaz y eficiente habrá muchos más pasos, muchas tareas que hacer. Estas tareas deberán estar bien planificadas y que sigan un guión que puede tener en cuenta aspectos como:

- Analizar las necesidades que tienen las personas que utilizarán este software, escuchar como el querrán, atender a sus indicaciones ...
- Diseñar una solución que tenga en cuenta todas las necesidades antes analizadas: qué deberá hacer el software, qué interfaces gráficas tendrá y cómo serán éstas, qué datos se deberán almacenar y cómo se hará ...
- Desarrollar el software que implemente todo lo analizado y diseñado anteriormente, haciéndolo de una forma lo más modular posible para facilitar el posterior mantenimiento o manipulación por parte de otros programadores.
- Llevar a cabo las pruebas pertinentes, tanto de forma individualizada para cada módulo como de forma completa, a fin de validar que el código desarrollado es correcto y que hace lo que debe hacer según lo establecido en los requerimientos.
- Implantar el software en el entorno donde los usuarios finales lo utilizarán.

Este apartado se centrará en el tercer punto, el desarrollo de software.

1.1. Concepto de programa informático

Un primer paso para poder empezar a analizar cómo hay que hacer un programa informático es tener claro qué es un programa y qué significa este concepto. En contraste con otros términos usados en informática, es posible referirse a un "programa" en el lenguaje coloquial sin tener que estar hablando necesariamente de ordenadores. Se podría estar refiriéndose al programa de un ciclo de conferencias o de cine. Pero, aunque no se trata de un contexto informático, este uso ya aporta una idea general de su significado.

Un programa informático es un conjunto de eventos ordenados de manera que se suceden de forma secuencial en el tiempo, uno tras otro.

Otro uso habitual, ahora sí vinculado al contexto de las máquinas y los autómatas, podría ser referirse al programa de una lavadora o de un robot de cocina. En este caso, sin embargo, lo que se sucede son un conjunto, no tanto de eventos, sino de órdenes que el

ENTORNOS DE DESARROLLO

Por ejemplo, el programa de un robot de cocina para hacer una crema de zanahoria sería:

1. Espera que introduzca las zanahorias bien limpiadas, una patata y especias al gusto.
2. Gira durante 1 minuto, avanzando progresivamente hasta la velocidad 5.
3. Espera que introduzca leche y sal.
4. Gira durante 30 segundos a velocidad 7.
5. Gira durante 10 minutos a velocidad 3 mientras cuece a una temperatura de 90 grados.
6. Se detiene. La crema de zanahoria está lista!

Este conjunto de comandos no es arbitrario, sino que sirve para llevar a cabo una tarea de cierta complejidad que no se puede hacer de una sola vez. Se debe hacer paso por paso. Todas las órdenes están vinculadas entre sí para llegar a alcanzar este objetivo y, sobre todo, es muy importante la disposición en que se llevan a cabo.

Entrando ya, ahora sí, en el mundo de los ordenadores, la forma en que se estructuran las tareas que deben ser ejecutadas es similar a los programas de electrodomésticos anteriormente citados. En este caso, sin embargo, en lugar de transformar ingredientes (o lavar ropa sucia, si se tratase de una lavadora), lo que la computadora transforma es información o datos.

Un programa informático no es más que una serie de órdenes que se llevan a cabo secuencialmente, aplicadas sobre un conjunto de datos.

Qué datos procesa un programa informático? Bueno, esto dependerá del tipo de programa:

- Un editor procesa los datos de un documento de texto.
- Una hoja de cálculo procesa datos numéricos ubicadas en un fichero.
- Un videojuego procesa los datos que hacen referencia a la forma y ubicación de enemigos y jugadores, las interfaces gráficas donde se encontrará el jugador, los puntos conseguidos ...
- Un navegador web procesa las órdenes del usuario y los datos que recibe desde un servidor ubicado en internet.
- Un reproductor de vídeo procesa los fotogramas almacenados en un archivo y el audio relacionado.

Por lo tanto, la tarea de un programador informático es escoger qué órdenes constituirán un programa de ordenador, en qué orden se deben llevar a cabo y sobre qué datos hay que aplicarlas para que el programa lleve a cabo la tarea que debe resolver.

Ejecutar un programa

Por "ejecutar un programa" se entiende hacer que el ordenador siga todas sus órdenes, desde la primera hasta la última.

ENTORNOS DE DESARROLLO

ordenador para resolver una multiplicación de tres números que para procesar textos o visualizar páginas en Internet.

Por otra parte, una vez hecho el programa, cada vez que se ejecute, el ordenador cumplirá todas las órdenes del programa.

De hecho, un ordenador es incapaz de hacer absolutamente nada por sí mismo, siempre hay que decirle qué debe hacer. Y eso se le llama mediante la ejecución de programas. Aunque desde el punto de vista del usuario puede parecer que cuando se pone en marcha un ordenador este funciona sin ejecutar ningún programa concreto, hay que tener en cuenta que su sistema operativo es un programa que está siempre en ejecución.

1.2. Código fuente, código objeto y código executable: máquinas virtuales

Editores de texto simples

Un editor de texto simple es aquel que permite escribir sólo texto sin formato. Son ejemplos el Bloc de Notas (Windows), Gedit o Emacs (Unix).

Para crear un programa lo que se hará será crear un archivo y escribir a un fichero cuyo serie de instrucciones que se quiere que el ordenador ejecute. Estas instrucciones deberán seguir unas pautas determinadas en función del lenguaje de programación elegido. Además, deberían seguir un orden determinado que dará sentido al programa escrito. Para empezar bastará con un editor de texto simple.

Una vez se ha terminado de escribir el programa, el conjunto de archivos de texto resultantes, donde se encuentran las instrucciones, se dice que contienen el **código fuente**. Este código fuente puede ser desde un nivel muy alto, muy cerca del lenguaje humano, hasta un nivel más bajo, más cercano al código de las máquinas, como el código ensamblador.

En el apartado "Tipos de lenguajes de programación" se describen las características de los lenguajes máquina, ensamblador, de alto nivel y de propósito específico.

La tendencia actual es hacer uso de lenguajes de alto nivel, es decir, cercanos al lenguaje humano. Pero esto hace aparecer un problema, y es que los archivos de código fuente no contienen el lenguaje máquina que entenderá el ordenador. Por lo tanto, resultan incomprensibles para el procesador. Para poder generar código máquina hay que hacer un proceso de traducción desde los mnemotécnicos que contiene cada archivo a las secuencias binarias que entiende el procesador.

ENTORNOS DE DESARROLLO

10100101011100001110111

El proceso llamado **compilación** es la traducción del código fuente de los archivos del programa en archivos en formato binario que contienen las instrucciones en un formato que el procesador puede entender. El contenido de estos archivos se denomina **código objeto**. El programa que hace este proceso se denomina **compilador**.

El código objeto es el código fuente traducido (por el compilador) a código máquina, pero este código aún no puede ser ejecutado por el ordenador.

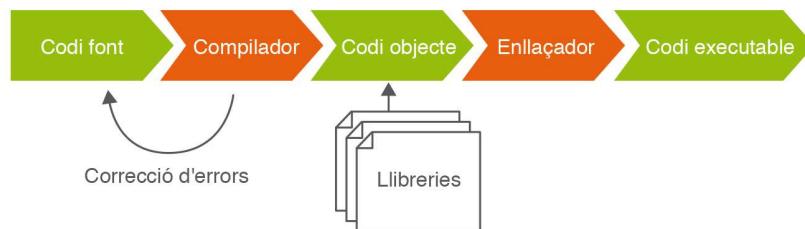
El código ejecutable es la traducción completa en código máquina, llevada a cabo por el enlazador (en inglés, *linker*). El código ejecutable es interpretado directamente por el ordenador.

El **enlazador** es el encargado de insertar el código objeto las funciones de las librerías que son necesarias para el programa y de llevar a cabo el proceso de montaje generando un archivo ejecutable.

Una **librería** es un colección de código predefinido que facilita la tarea del programador a la hora de codificar un programa.

En la [figura 1.1](#) se muestra un resumen ordenado de todos los conceptos definidos. El código fuente desarrollado por los programadores se convertirá en código objeto con la ayuda del compilador. Este ayudará a localizar los errores de sintaxis o de compilación que se encuentren en el código fuente. Con el enlazador, que recogerá el código objeto y las librerías, se generará el código ejecutable.

Figura 1.1. Proceso de transformación de un código fuente a un código ejecutable



1.2.1. máquina virtual

El concepto de máquina virtual surge con el objetivo de facilitar el desarrollo de compiladores que generan código para diferentes procesadores.

La **compilación** consta de dos fases:

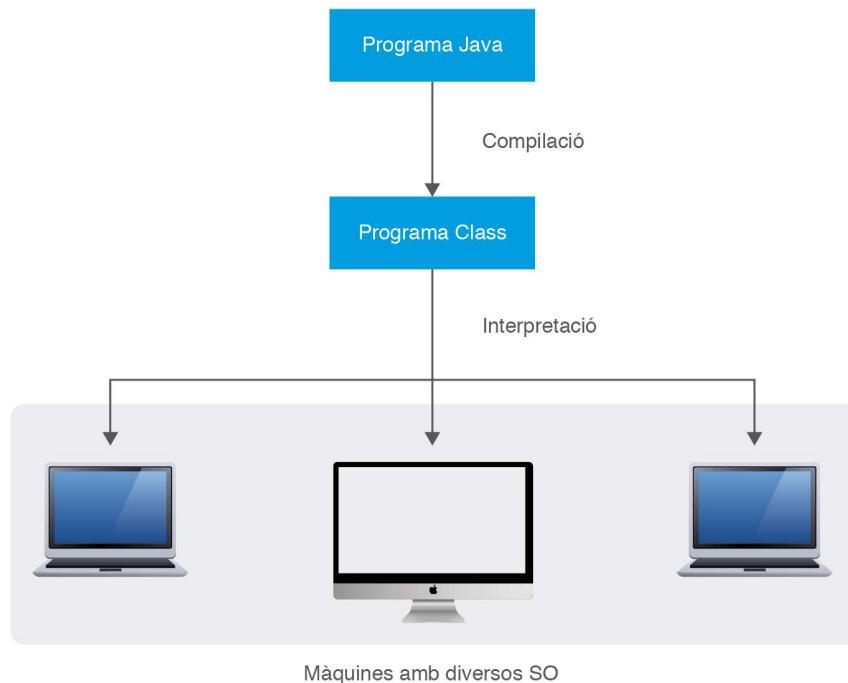
ENTORNOS DE DESARROLLO

que se ejecutará.

- La segunda fase traduce el lenguaje intermedio a un lenguaje comprensible para la máquina.

Llegado este punto se podría plantear la pregunta: ¿por qué dividir la compilación en dos fases? El objetivo es que el código de la primera fase, el código intermedio, sea común para cualquier procesador, y que el código generado en la segunda fase sea el específico para cada procesador. De hecho, la traducción del lenguaje intermedio al lenguaje máquina no se suele hacer mediante compilación sino mediante un intérprete, tal como se muestra en la [figura 1.2](#).

Figura 1.2. Máquina virtual



La máquina virtual Java

La máquina virtual Java (JVM) es el entorno en el que se ejecutan los programas Java. Es un programa nativo, es decir, ejecutable en una plataforma específica, que es capaz de interpretar y ejecutar instrucciones expresadas en un código de bytes o (el *bytecode* de Java) que es generado por el compilador del lenguaje Java.

Código de bytes

El código de bytes no es un lenguaje de alto nivel, sino un verdadero código máquina de bajo nivel, viable incluso como lenguaje de entrada para un microprocesador físico.

La máquina virtual Java es una pieza fundamental de la tecnología Java. Se sitúa en un nivel superior al hardware sobre el que se desea ejecutar la aplicación y actúa como un puente entre el código de bytes a ejecutar y el sistema. Así, cuando un programador escribe una aplicación Java, lo hace pensando en la JVM encargada de ejecutar la aplicación y no hay ningún motivo para pensar en las características de la plataforma física sobre la que se debe ejecutar la aplicación. La JVM será la encargada, al ejecutar la aplicación, de convertir el código de bytes a código nativo de la plataforma física.

ENTORNOS DE DESARROLLO

ejecutar en otros sistemas operativos (Linux, Solaris y Apple OS X) con el único requerimiento de disponer de la JVM para el sistema correspondiente.

El concepto de máquina virtual Java se usa en dos ámbitos: por un lado, para hacer referencia al conjunto de especificaciones que debe cumplir cualquier implementación de la JVM; por otra parte, para hacer referencia a las diversas implementaciones de la máquina virtual Java existentes y de las que hay que utilizar alguna para ejecutar las aplicaciones Java.

A los materiales web se puede encontrar una explicación, paso a paso, de cómo descargar e instalar la máquina virtual Java.

La empresa Sun Microsystems es la propietaria de la marca registrada Java, y esta se utiliza para certificar las implementaciones de la JVM que se ajustan y son totalmente compatibles con las especificaciones de la JVM, en el prefacio de las que se dice: "Esperamos que esta especificación documente suficientemente la máquina virtual de Java para hacer posibles implementaciones desde cero. Sun proporciona tests que verifican que las implementaciones de la máquina virtual Java operan correctamente. "

1.3. Tipo de lenguajes de programación

Establecido el concepto de programa informático y los conceptos de código fuente, código objeto y código ejecutable (así como el de máquina virtual), hay ahora establecer las diferencias entre los diversos tipos de código fuente existentes, a través de los cuales se llega a obtener un programa informático.

Un lenguaje de programación es un lenguaje que permite establecer una comunicación entre el hombre y la máquina. El lenguaje de programación identificará el código fuente, que el programador desarrollará para indicar a la máquina, una vez este código se haya convertido en código ejecutable, qué pasos debe dar.

Durante los últimos años ha existido una evolución constante en los lenguajes de programación. Se ha establecido una creciente evolución en la que se van incorporando elementos que permiten crear programas cada vez más sólidos y eficientes. Esto facilita mucho la tarea del programador para el desarrollo del software, su mantenimiento y la adaptación. Hoy en día, existen, incluso, lenguajes de programación que permiten la creación de aplicaciones informáticas a personas sin conocimientos técnicos de informática, por el hecho de existir una creación prácticamente automática del código a partir de unas preguntas.

Los diferentes tipos de lenguajes son:

- Lenguaje de primera generación o lenguaje máquina.
- Lenguajes de segunda generación o lenguajes de ensamblador.

ENTORNOS DE DESARROLLO

- Lenguajes de quinta generación.

El primer tipo de lenguaje que se desarrolló es el llamado **lenguaje de primera generación o lenguaje máquina**. Es el único lenguaje que entiende el ordenador directamente.

Su estructura está totalmente adaptada a los circuitos impresos de ordenadores o procesadores electrónicos y muy alejada de la forma de expresión y análisis de los problemas propios de los humanos (las instrucciones se expresan en código binario). Esto hace que la programación en este lenguaje resulte tediosa y complicada, ya que se requiere un conocimiento profundo de la arquitectura física del ordenador. Además, se valorará que el código máquina hace posible que el programador utilice la totalidad de recursos del hardware, con lo cual se pueden obtener programas muy eficientes.

`10110000 01100001`

Esta línea contiene una instrucción que move un valor al registro del procesador.



Actualmente, debido a la complejidad del desarrollo de este tipo de lenguaje, está prácticamente en desuso. Sólo se utilizará en procesadores muchos concretos o para funcionalidades muy específicas.

El segundo tipo de lenguaje de programación son los **lenguajes de segunda generación o lenguajes de ensamblador**. Se trata del primer lenguaje de programación que utiliza códigos mnemotécnicos para indicar a la máquina las operaciones que debe llevar a cabo. Estas operaciones, muy básicas, han sido diseñadas a partir del conocimiento de la estructura interna de la propia máquina.

Cada instrucción en lenguaje ensamblador corresponde a una instrucción en lenguaje máquina. Estos tipos de lenguajes dependen totalmente del procesador que utilice la máquina, por eso se dice que están orientados a las máquinas.

En la [figura 1.3](#) se muestra un esquema del funcionamiento de los lenguajes de segunda generación. A partir del código escrito en lenguaje ensamblador, el programa traductor (ensamblador) lo convierte en código de primera generación, que será interpretado por la máquina.

Figura 1.3. Lenguaje de segunda generación



ENTORNOS DE DESARROLLO

memoria.

1.3.1. Características de los lenguajes de primera y segunda generación

Como ventajas de los lenguajes de primera y segunda generación se pueden establecer:

- Permiten escribir programas muy optimizados que aprovechan al máximo el hardware (*hardware*) disponible.
- Permiten al programador especificar exactamente qué instrucciones quiere que se ejecuten.

Los inconvenientes son los siguientes:

- Los programas escritos en lenguajes de bajo nivel están completamente ligados al hardware donde se ejecutarán y no se pueden trasladar fácilmente a otros sistemas con un hardware diferente.
- Hay que conocer a fondo la arquitectura del sistema y del procesador para escribir buenos programas.
- No permiten expresar de forma directa conceptos habituales a nivel de algoritmo.
- Son difíciles de codificar, documentar y mantener.

El siguiente grupo de lenguajes se conoce como **lenguajes de tercera generación o lenguajes de alto nivel**. Estos lenguajes, más evolucionados, utilizan palabras y frases relativamente fáciles de entender y proporcionan también facilidades para expresar alteraciones del flujo de control de una forma bastante sencilla e intuitiva.

Los lenguajes de tercera generación o de alto nivel se utilizan cuando se quiere desarrollar aplicaciones grandes y complejas, donde se prioriza el hecho de facilitar y comprender cómo hacer las cosas (lenguaje humano) por encima del rendimiento del software o de su uso de la memoria.

Los esfuerzos encaminados a hacer la tarea de programación independiente de la máquina donde se ejecutarán dieron como resultado la aparición de los lenguajes de programación de alto nivel.

Los lenguajes de alto nivel son normalmente fáciles de aprender porque están formados por elementos de lenguajes naturales, como el inglés. A continuación se muestra un ejemplo de algoritmo implementado en un lenguaje de alto nivel, concretamente en Basic. Este algoritmo calcula el factorial de un número dado a la función como parámetro. Se puede observar como es fácilmente comprensible con un mínimo conocimiento del inglés.

En Basic, el lenguaje de alto nivel más conocido, las órdenes como **IF comptador = 10 THEN STOP** pueden utilizarse para pedir a la computadora que pare si la variable contador es igual a diez.

'-----
'Función Factorial

ENTORNOS DE DESARROLLO

```

For i = 1 To num - 1
    num = num * y
    Factorial = num
Next
End Function

```

Como consecuencia de este alejamiento de la máquina y acercamiento a las personas, los programas escritos en lenguajes de programación de tercera generación no pueden ser interpretados directamente por el ordenador, sino que es necesario llevar a cabo previamente su traducción a lenguaje máquina . Hay dos tipos de traductores: los compiladores y los intérpretes.

compiladores

Son programas que traducen el programa escrito con un lenguaje de alto nivel al lenguaje máquina. El compilador detectará los posibles errores del programa fuente para conseguir un programa ejecutable depurado.

Algunos ejemplos de códigos de programación que deberán pasar por un compilador son: Pascal, C, C ++, .NET, ...

En la [figura 1.4](#) se puede ver, en un esquema, la función del compilador entre los dos lenguajes.

Figura 1.4. Código compilado

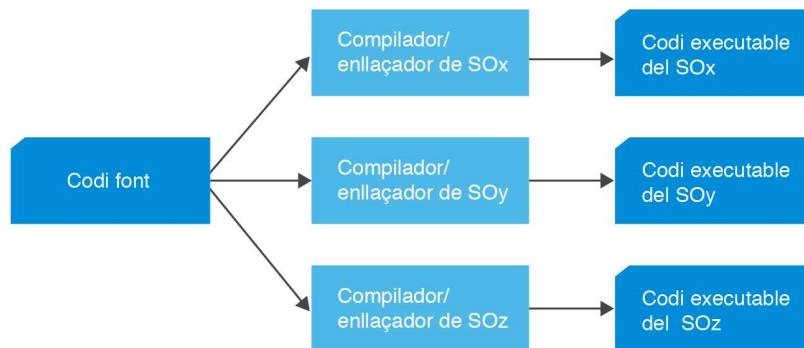


El procedimiento que deberá seguir un programador es el siguiente:

- Crear el código fuente.
- Crear el código ejecutable en uso de compiladores y enlazadores.
- El código ejecutable depende de cada sistema operativo. Para cada sistema hay un compilador, es decir, si se quiere ejecutar el código con otro sistema operativo debe recomilar el código fuente.
- El programa resultante se ejecuta directamente desde el sistema operativo.

En la [figura 1.5](#) se puede observar un esquema que representa la dependencia del sistema operativo a la hora de elegir y utilizar compilador.

Figura 1.5. Código compilado por SO



ENTORNOS DE DESARROLLO

máquina, pero, a diferencia del compilador, lo hace en tiempo de ejecución. Es decir, no se hace un proceso previo de traducción de todo el programa fuente a código de bytes, sino que se va traduciendo y ejecutando instrucción por instrucción.

Algunos ejemplos de códigos de programación que deberán pasar por un intérprete son: Javascript, PHP, ASP ...

Algunas características de los lenguajes interpretados son:

- El código interpretado no es ejecutado directamente por el sistema operativo, sino que hace uso de un intérprete.
- Cada sistema tiene su propio intérprete.

Compiladores ante intérpretes

El intérprete es notablemente más lento que el compilador, ya que lleva a cabo la traducción al tiempo que la ejecución. Además, esta traducción se hace siempre que se ejecuta el programa, mientras que el compilador sólo la lleva a cabo una vez. La ventaja de los intérpretes es que hacen que los programas sean más portables. Así, un programa compilado en un ordenador con sistema operativo Windows no funcionará en un Macintosh, o en un ordenador con sistema operativo Linux, a menos que se vuelva a compilar el programa fuente en el nuevo sistema.

1.3.2. Características de los lenguajes de tercera, cuarta y quinta generación

Los lenguajes de tercera generación son aquellos que son capaces de contener y ejecutar, en una sola instrucción, el equivalente a varias instrucciones de un lenguaje de segunda generación.

Las ventajas de los lenguajes de tercera generación son:

- El código de los programas es mucho más sencillo y comprensible.
- Son independientes del hardware (no hacen ninguna referencia). Por este motivo es posible "llevar" el programa entre diferentes ordenadores / arquitecturas / sistemas operativos (siempre que en el sistema de destino exista un compilador para este lenguaje de alto nivel).
- Es más fácil y rápido escribir los programas y más fácil mantenerlos.

Los inconvenientes de los lenguajes de tercera generación son:

- Su ejecución en un ordenador puede resultar más lenta que el mismo programa escrito en lenguaje de bajo nivel, aunque esto depende mucho de la calidad del compilador que haga la traducción.

Ejemplos de lenguajes de programación de tercera generación: C, C ++, Java, Pascal ...

Los lenguajes de cuarta generación o lenguajes de propósito específico. Aportan un nivel muy alto de abstracción en la programación, permitiendo desarrollar aplicaciones sofisticadas en un espacio corto de tiempo, muy inferior al necesario para los lenguajes de 3^a generación.

ENTORNOS DE DESARROLLO

Integrado de Desarrollo.

Se automatizan ciertos aspectos que antes había que hacer a mano. Incluyen herramientas orientadas al desarrollo de aplicaciones (IDE) que permiten definir y gestionar bases de datos, realizar informes (p. Ej.: Oracle reports), consultas (p. Ej.: informix 4GL), módulos ..., escribiendo muy pocas líneas de código o ninguna.

Permiten la creación de prototipos de una aplicación rápidamente. Los prototipos permiten tener una idea del aspecto y del funcionamiento de la aplicación antes de que el código esté terminado. Esto facilita la obtención de un programa que reúna las necesidades y expectativas del cliente.

Algunos de los aspectos positivos que muestran este tipo de lenguajes de programación son:

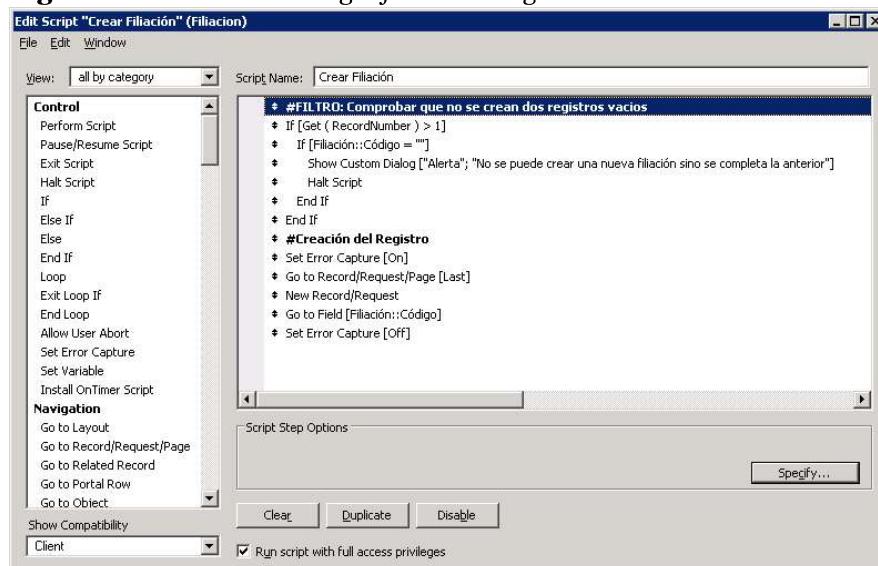
- Mayor abstracción.
- Menor esfuerzo de programación.
- Menor coste de desarrollo del software.
- Basados en generación de código a partir de especificaciones de nivel muy alto.
- Se pueden llevar a cabo aplicaciones sin ser un experto en el lenguaje.
- Suelen tener un conjunto de instrucciones limitado.
- Son específicos del producto que les ofrece.

Estos lenguajes de programación de cuarta generación están orientados, básicamente, a las aplicaciones de negocio y el manejo de bases de datos.

Algunos ejemplos de lenguajes de cuarta generación son Visual Basic, Visual Basic .NET, ABAP de SAP, FileMaker, PHP, ASP, 4D ...

En la [figura 1.6](#) se puede ver el entorno de trabajo y un ejemplo de código fuente de FileMaker.

Figura 1.6. FileMaker: Lenguaje de cuarta generación



ENTORNOS DE DESARROLLO

per al tractament de problemes relacionats amb la intel·ligència artificial i els sistemes experts.

En lloc d'executar només un conjunt d'ordres, l'objectiu d'aquests sistemes és "pensar" i anticipar les necessitats dels usuaris. Aquests sistemes es troben encara en desenvolupament. Es tractaria del paradigma lòtic.

Algunes exemples de llenguatges de cinquena generació són Lisp o Prolog.

1.4. Paradigmes de programació

És difícil establir una classificació general dels llenguatges de programació, ja que existeix un gran nombre de llenguatges i, de vegades, diferents versions d'un mateix llenguatge. Això provocarà que en qualsevol classificació que es faci un mateix llenguatge pertanyi a més d'un dels grups establerts. Una classificació molt estesa, atenent a la forma de treballar dels programes i a la filosofia amb què van ser concebuts, és la següent:

- Paradigma imperatiu/estructurat.
- Paradigma d'objectes.
- Paradigma funcional.
- Paradigma lòtic.

El paradigma imperativo / estructurado debe su nombre al papel dominante que ejercen las sentencias imperativas, es decir aquellas que indican llevar a cabo una determinada operación que modifica los datos guardados en memoria.

Algunos de los lenguajes imperativos son C, Basic, Pascal, Cobol ...

La técnica seguida en la programación imperativa es la **programación estructurada**. La idea es que cualquier programa, por complejo y grande que sea, puede ser representado mediante tres tipos de estructuras de control:

- Secuencia.
- Selección.
- Iteración.

Por otra parte, también se propone desarrollar el programa con la técnica de diseño descendente (*top-down*). Es decir, modular el programa creando porciones más pequeñas de programas con tareas específicas, que se subdividen en otros subprogramas, cada vez más pequeños. La idea es que estos subprogramas típicamente llamados funciones o procedimientos deben resolver un único objetivo o tarea.

Imaginemos que tenemos que hacer una aplicación que registre los datos básicos del personal de una escuela, datos como pueden ser el nombre, el DNI, y que calcule el salario de los profesores así como el de los administrativos, donde el salario de los

ENTORNOS DE DESARROLLO

```

const float SOU_BASE = 1.000;

struct Administrativo
{
    string nombre;
    string DNI;
    float Salario;
}

struct Profesor
{
    string nombre;
    string DNI;
    int numHores;
    float salario;
}

void AsignarSalariAdministratiu (Administrativo administratiu1)
{
    administratiu1. salario = SOU_BASE * 10;
}

void AsignarSalariProfessor (Profesor profesor1)
{
    profesor1. salario = SOU_BASE + (numHores * 12);
}

```

El paradigma de objetos, típicamente conocido como Programación Orientada a Objetos (POO, o OOP en inglés), es un paradigma de construcción de programas basado en una abstracción del mundo real. En un programa orientado a objetos, la abstracción no son procedimientos ni funciones sino los objetos. Estos objetos son una representación directa de algo del mundo real, como un libro, una persona, un pedido, un empleado ...

Algunos de los lenguajes de programación orientada a objetos son C++, Java, C# ...

Un objeto es una combinación de datos (llamadas atributos) y métodos (funciones y procedimientos) que nos permiten interactuar con él. En este tipo de programación, por lo tanto, los programas son conjuntos de objetos que interactúan entre ellos a través de mensajes (llamadas a métodos).

La programación orientada a objetos se basa en la integración de 5 conceptos: abstracción, encapsulación, modularidad, jerarquía y polimorfismo, que es necesario comprender y seguir de manera absolutamente rigurosa. No seguir sistemáticamente, omitirlos puntualmente por prisa u otras razones hace perder todo el valor y los beneficios que nos aporta la orientación a objetos.

```

class Trabajador {
    private:
        string nombre;
        string DNI;
    protected:
        static const float SOU_BASE = 1.000;
    public:
        string GetNom () {return this.nom;}

```

ENTORNOS DE DESARROLLO

```

        virtual float salario () = 0;
    }

class Administrativo: public Trabajador {
    public:
        float Salario () {return SOU_BASE * 10;};
}

class Profesor: public Trabajador {
    private:
        int numHores;
    public:
        float Salario () {return SOU_BASE + (numHores * 15);;}
}

```

El paradigma funcional está basado en un modelo matemático. La idea es que el resultado de un cálculo es la entrada del siguiente, y así sucesivamente hasta que una composición produzca el resultado deseado.

Los creadores de los primeros lenguajes funcionales pretendían convertirlos en lenguajes de uso universal para el procesamiento de datos en todo tipo de aplicaciones, pero, con el paso del tiempo, se ha utilizado principalmente en ámbitos de investigación científica y aplicaciones matemáticas .

Uno de los lenguajes más típicos del paradigma funcional es el Lisp. Véase un ejemplo de programación del factorial con este lenguaje:

```

> (Defun factorial (n)
  (If (= n 0)
      1
      (* N (factorial (- n 1))))))
FACTORIAL
> (Factorial 3)
6

```

El paradigma lógico tiene como característica principal la aplicación de las reglas de la lógica para inferir conclusiones a partir de datos.

Un programa lógico contiene una base de conocimiento sobre la que se llevan a cabo consultas. La base de conocimiento está formada por hechos, que representan la información del sistema expresada como relaciones entre los datos y reglas lógicas que permiten deducir consecuencias a partir de combinaciones entre los hechos y, en general, otras reglas.

Uno de los lenguajes más típicos del paradigma lógico es el Prolog.

Ejemplo de desarrollo práctico del paradigma lógico

ENTORNOS DE DESARROLLO

Reglas de la base de conocimiento:

- R1: Si fiebre, entonces estar en casa en reposo.
- R2: Si malestar, entonces ponerse termómetro.
- R3: Si termómetro marca una temperatura > 37º, llavors febre.
- R4: Si diarrea, llavors dieta.

Si seguim un raonament d'encadenament cap endavant, el procediment seria:

Indicar el motor d'inferència, els fets: malestar i termòmetre marca 39.

```
<code>Base de fets = { malestar, termòmetre marca 39º }</code>
```

El sistema identifica les regles aplicables: R2 i R3. L'algorisme s'inicia aplicant la regla R2, incorporant en la base de fets "posar-se el termòmetre".

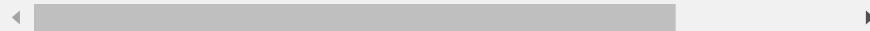
Base de fets = { malestar, termòmetre marca 39º, posar-se termòmetre }

Com que no s'ha solucionat el problema, continua amb la següent regla R3, afegint a la base de fets "febre".

Base de fets = { malestar, termòmetre marca 39º, posar-se termòmetre, febre }

Com que no s'ha solucionat el problema, torna a identificar un subconjunt de regles aplicables, excepte les ja utilitzades. El sistema identifica les regles aplicables: R1, tot incorporant a la base de fets "estar a casa en repòs".

Base de fets = { malestar, termòmetre marca 39º, posar-se termòmetre, febre, es



Com que repòs està a la base de fets, s'ha arribat a una resposta positiva a la pregunta formulada.

El paradigma és àmpliament utilitzat en les aplicacions que tenen a veure amb la Intel·ligència Artificial, particularment en el camp de sistemes experts i processament del llenguatge humà. Un sistema expert és un programa que imita el comportament d'un expert humà. Per tant conté informació (és a dir una base de coneixements) i una eina per comprendre les preguntes i trobar la resposta correcta examinant la base de dades (un motor d'inferència).

També és útil en problemes combinatoris o que requereixin una gran quantitat o amplitud de solicions alternatives, d'acord amb la naturalesa del mecanisme de tornada enrere (*backtracking*).

1.5. Característiques dels llenguatges més difosos

Existeixen molts llenguatges de programació diferents, fins al punt que moltes tecnologies tenen el seu llenguatge propi. Cada un d'aquests llenguatges té un seguit de particularitats que el fan diferent de la resta.

Els llenguatges de programació més difosos són aquells que més es fan servir en cadascun dels diferents àmbits de la informàtica. En l'àmbit educatiu, per exemple, es considera un

ENTORNOS DE DESARROLLO

Els llenguatges de programació més difosos corresponents a diferents àmbits, a diferents tecnologies o a diferents tipus de programació tenen una sèrie de característiques en comú que són les que marquen les similituds entre tots ells.

1.5.1. Características de la programació estructurada

La programació estructurada va ser desenvolupada pel neerlandès Edsger W. Dijkstra i es basa en el denominat teorema de l'estructura. Per això utilitza únicament tres estructures: seqüència, selecció i iteració, essent innecessari l'ús de la instrucció o instruccions de transferència incondicional (GOTO, EXIT FUNCTION, EXIT SUB o múltiples RETURN).

D'aquesta forma les característiques de la programació estructurada són la claredat, el teorema de l'estructura i el disseny descentrat.

Claredat

Hi haurà d'haver prou informació al codi per tal que el programa pugui ser entès i verificat: comentaris, noms de variables comprensibles i procediments entenedors... Tot programa estructurat pot ser llegit des del principi a la fi sense interrupcions en la seqüència normal de lectura.

Teorema de l'estructura

Demostra que tot programa es pot escriure utilitzant únicament les tres estructures bàsiques de control:

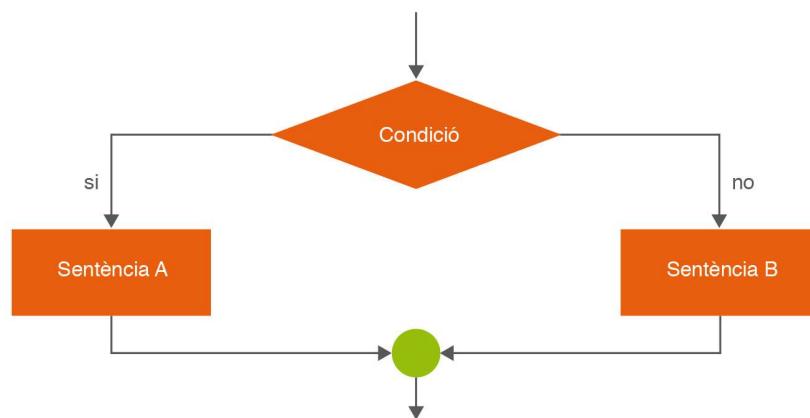
- Secuencia: instrucciones ejecutadas sucesivamente, una tras otra. En la figura 1.7 se puede observar un ejemplo de la estructura básica de secuencia, donde primero se ejecutará la sentencia A y, posteriormente, la B.

Figura 1.7. Ejemplo de secuencia



- Selección: la instrucción condicional con doble alternativa, de la forma "si condición, entonces sentencias, sino SentènciaB". En la figura 1.8 se puede observar un esquema que ejemplifica la estructura básica de selección.

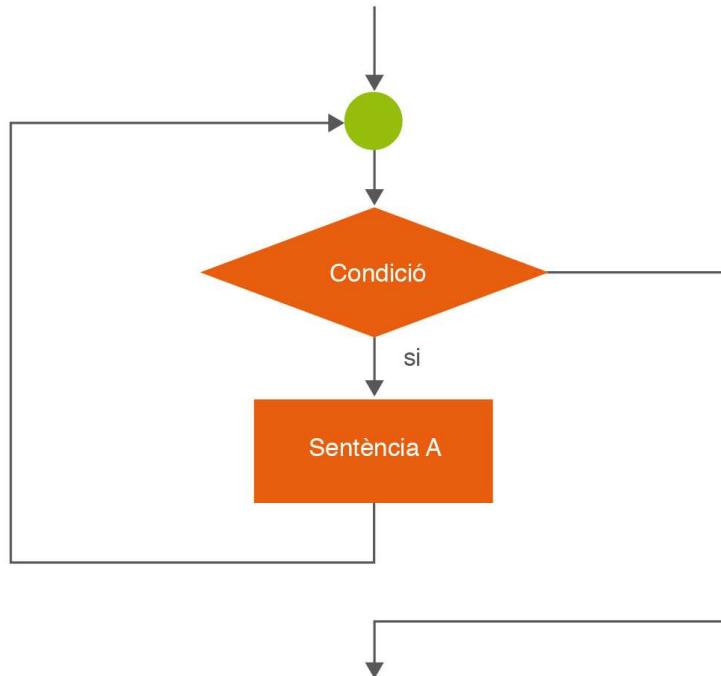
Figura 1.8. Ejemplo de selección



ENTORNOS DE DESARROLLO

~~Permet observar un esquema que ejemplifica la esencia básica de iteración.~~

Figura 1.9. Ejemplo de iteración

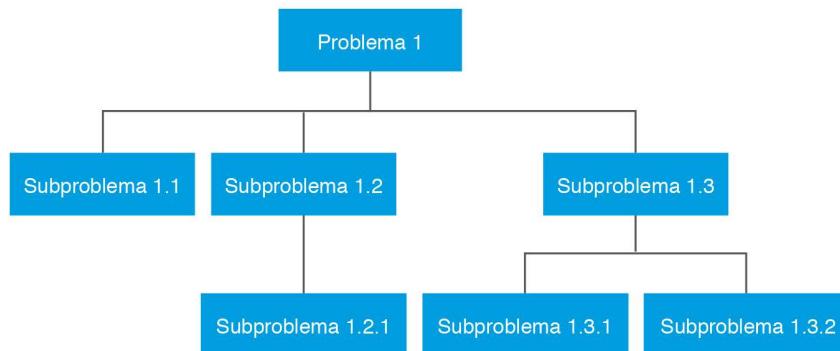


diseño descendente

El diseño descendente es una técnica que se basa en el concepto de "divide y vencerás" para resolver un problema en el ámbito de la programación. Se trata de la resolución del problema a lo largo de diferentes niveles de abstracción partiendo de un nivel más abstracto y finalizando en un nivel de detalle.

A la [figura 1.10](#) es pot observar un exemple del disseny descendent. A partir del problema 1 s'obtenen diversos subproblemes (subproblema 1.1, subproblema 1.2 i subproblema 1.3). La resolució d'aquests subproblemes serà molt més senzilla que la del problema original per tal com se n'ha reduït considerablement l'abast i la mida. De forma iterativa es pot observar com aquests subproblemes es tornen a dividir, a la vegada, en altres subproblemes.

Figura 1.10. Disseny descendent



La visió moderna de la programació estructurada introduceix les característiques de programació modular i tipus abstractes de dades (TAD).

ENTORNOS DE DESARROLLO

menudo un conjunto enorme de sentencias la ejecución de las que es compleja de seguir, y de entender, con lo cual se hace casi imposible la depuración de errores y la introducción de mejoras. Incluso, puede darse el caso de tener que abandonar el código preexistente porque resulta más fácil empezar de nuevo.

Cuando se habla de programación modular, nos referimos a la división de un programa en partes más manejables e independientes. Una regla práctica para lograr este propósito es establecer que cada segmento del programa no exceda, en longitud, de un palmo de codificación.

En la mayoría de lenguajes, los módulos se traducen en:

- Procedimientos: son subprogramas que llevan a cabo una tarea determinada y devuelven o más de un valor. Se utilizan para estructurar un programa y mejorar su claridad.
- Funciones: son subprogramas que llevan a cabo una determinada tarea y devuelven un único resultado o valor. Se utilizan para crear operaciones nuevas que no ofrece el lenguaje.

Tipos abstractos de datos (TAD)

En programación, el *tipo de datos* de una variable es el conjunto de valores que la variable puede asumir. Por ejemplo, una variable de tipo booleano puede adoptar sólo dos valores posibles: *verdadero* *falso*. Además, hay un conjunto limitado pero bien definido de operaciones que tienen sentido sobre los valores de un tipo de datos; así, operaciones típicas sobre el tipo booleano son AND OR.

Los lenguajes de programación asumen un número determinado de tipos de datos, que puede variar de un lenguaje a otro; así, en Pascal tenemos los *enteros*, los *reales*, los *booleanos*, los *caracteres* ... Estos tipos de datos son llamados *tipos de datos básicos* en el contexto de los lenguajes de programación.

Hasta hace unos años, toda la programación se basaba en este concepto de tipo y no eran pocos los problemas que aparecían, ligados muy especialmente a la complejidad de los datos que se tenían que definir. Apareció la posibilidad de poder definir *tipos abstractos de datos*, donde el programador puede definir un nuevo tipo de datos y sus posibles operaciones.

Ejemplo de implementación de un tipo abstracto de datos implementado en el lenguaje C

```

struct TADpila
{
    int top;
    int elementos [MAX_PILA];
}

void crear (struct TADpila * pila)
{
    Pila.top = -1;
}

void apilar (struct TADpila * pila, int elem)
{
    Pila.elementos [pila.top ++] = elem;
}

```

ENTORNOS DE DESARROLLO

1.5.2. Características de la programación orientada a objetos

Un dels conceptes importants introduïts per la programació estructurada és l'abstracció de funcionalitats a través de funcions i procediments. Aquesta abstracció permet a un programador utilitzar una funció o procediment coneixent només què fa, però desconeixent el detall de com ho fa.

Aquest fet, però, té diversos inconvenients:

- Les funcions i procediments comparteixen dades del programa, cosa que provoca que canvis en un d'ells afectin a la resta.
- Al moment de dissenyar una aplicació és molt difícil preveure detalladament quines funcions i procediments necessitarem.
- La reutilització del codi és difícil i acaba consistint a copiar i enganxar determinats trossos de codi, i retocar-los. Això és especialment habitual quan el codi no és modular.

L'orientació a objectes, concebut als anys setanta i vuitanta però estesa a partir dels noranta, va permetre superar aquestes limitacions.

L'orientació a objectes (en endavant, OO) és un paradigma de construcció de programes basat en una abstracció del món real.

En un programa orientat a objectes, l'abstracció no són els procediments ni les funcions, són els objectes. Aquests objectes són una representació directa d'alguna cosa del món real, com ara un llibre, una persona, una organització, una comanda, un empleat...

Un **objecte** és una combinació de dades (anomenades atributs) i mètodes (funcions i procediments) que ens permeten interactuar amb ell. En OO, doncs, els programes són conjunts d'objectes que interactuen entre ells a través de missatges (crides a mètodes).

Els llenguatges de POO (programació orientada a objectes) són aquells que implementen més o menys fidelment el paradigma OO. La programació orientada a objectes es basa en la integració de 5 conceptes: abstracció, encapsulació, modularitat, jerarquia i polimorfisme, que és necessari comprendre i seguir de manera absolutament rigorosa. No seguir-los sistemàticament o ometre'ls puntualment, per pressa o altres raons, fa perdre tot el valor i els beneficis que aporta l'orientació a objectes.

Abstracció

És el procés en el qual se separen les propietats més importants d'un objecte de les que no ho són. És a dir, per mitjà de l'abstracció es defineixen les característiques essencials d'un objecte del món real, els atributs i comportaments que el defineixen com a tal, per després

ENTORNOS DE DESARROLLO

En la tecnologia orientada a objectes l'eina principal per suportar l'abstracció és la **classe**. Es pot definir una classe com una descripció genèrica d'un grup d'objectes que comparteixen característiques comunes, les quals són especificades en els seus atributs i comportaments.

Encapsulació

Permet als objectes triar quina informació és publicada i quina informació és amagada a la resta dels objectes. Per això els objectes solen presentar els seus mètodes com a interfícies públiques i els seus atributs com a dades privades o protegides, essent inaccessible des d'altres objectes. Les característiques que es poden atorgar són:

- Públic: qualsevol classe pot accedir a qualsevol atribut o mètode declarat com a públic i utilitzar-lo.
- Protegit: qualsevol classe heretada pot accedir a qualsevol atribut o mètode declarat com a protegit a la classe mare i utilitzar-lo.
- Privat: cap classe no pot accedir a un atribut o mètode declarat com a privat i utilitzar-lo.

Modularitat

Permet poder modificar les característiques de cada una de les classes que defineixen un objecte, de forma independent de la resta de classes en l'aplicació. En altres paraules, si una aplicació es pot dividir en mòduls separats, normalment classes, i aquests mòduls es poden compilar i modificar sense afectar els altres, aleshores aquesta aplicació ha estat implementada en un llenguatge de programació que suporta la modularitat.

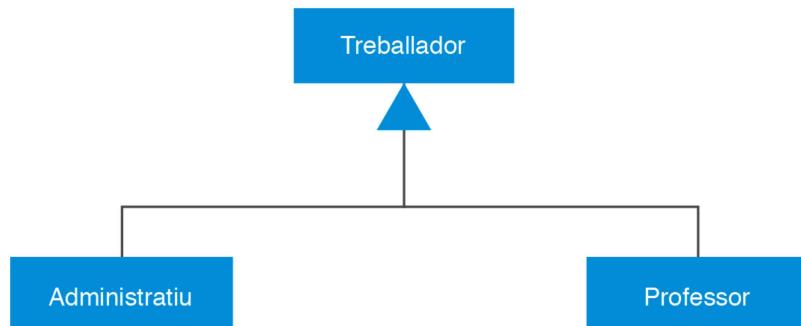
Jerarquia

Permet l'ordenació de les abstraccions. Les dues jerarquies més importants d'un sistema complex són l'herència i l'agregació.

L'herència també es pot veure com una forma de compartir codi, de manera que quan s'utilitza l'herència per definir una nova classe només s'ha d'afegir allò que sigui diferent, és a dir, reaprofita els mètodes i variables, i especialitza el comportament.

Per exemple, es pot identificar una classe *pare* anomenada *treballador* i dues classes *filles*, és a dir dos subtipus de treballadors, *administratiu* i *professor*.

Figura 1.11. Exemple d'herència

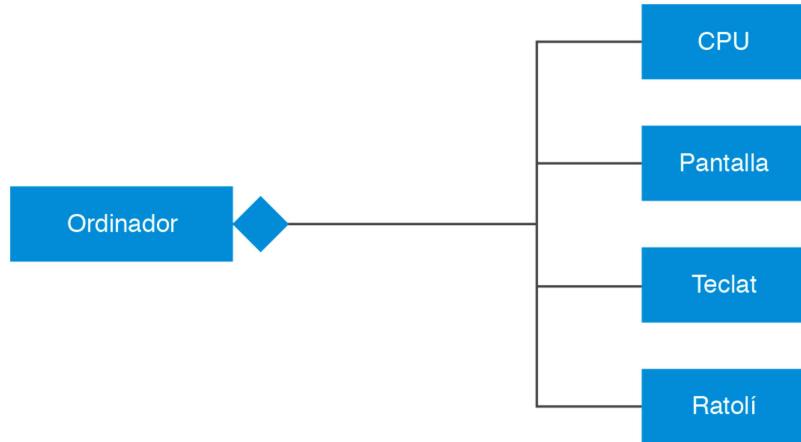


A la [figura 1.11](#) es pot observar la representació en forma de diagrama de l'exemple expli- cat anteriorment: les classes *administratiu* i *professor* que hereten de la classe *treballador*.

ENTORNOS DE DESARROLLO

ponents no tenen sentit sense l'ordinador. A la [figura 1.12](#) es pot observar un exemple d'agregació en què la classe ordinador està composta per les altres quatre classes.

Figura 1.12. Exemple d'agregació



El polimorfisme

És una característica que permet donar diferents formes a un mètode, ja sigui en la definició com en la implementació.

La sobrecàrrega (*overload*) de mètodes consisteix a implementar diverses vegades un mateix mètode però amb paràmetres diferents, de manera que, en invocar-lo, el compilador decideix quin dels mètodes s'ha d'executar, en funció dels paràmetres de la crida.

Un exemple de mètode sobrecarregat és aquell que calcula el salari d'un treballador en una empresa. En funció de la posició que ocupa el treballador tindrà més o menys conceptes a la seva nòmina (més o menys incentius, per exemple).

El mateix mètode, que podríem anomenar *CàlculSalari* quedará implementat de forma diferent en funció de si es calcula el salari d'un operari (amb menys conceptes en la seva nòmina, la qual cosa provoca que el mètode rebi menys variables) o si es calcula el salari d'un directiu.

La sobreescritura (*override*) de mètodes consisteix a reimplementar un mètode heretat d'una superclasse exactament amb la mateixa definició (incloent nom de mètode, paràmetres i valor de retorn).

Un exemple de sobrecàrrega de mètodes podria ser el del mètode *Area()*. A partir d'una classe *Figura* que conté el mètode *Area()*, existeix una classe derivada per a alguns tipus de figures (per exemple, *Rectangle* o *Quadrat*).

La implementación del método *Area()* será diferente en cada una de las clases derivadas; éstas pueden implementarse de forma diferente (en función de cómo se calcule en cada caso el área de la figura) o definirse de forma diferente.

1.6. Fases del desarrollo de los sistemas de información

ENTORNOS DE DESARROLLO

acompañamiento a lo largo de este desarrollo, proporcionando pautas, indicaciones, métodos y documentos para ayudar, sobre todo, los jefes de proyecto más inexpertos.

Dentro de estas metodologías hay Métrica v3.0. Ha sido desarrollada por el Ministerio de Administraciones Públicas. Se trata de una metodología para la planificación, desarrollo y mantenimiento de los sistemas de información de una organización. Para el desarrollo de software hay que fijarse en la parte que hace referencia al desarrollo de los sistemas de información (SI), dentro de la metodología Métrica. Divide el desarrollo en 5 fases, que se siguen de forma secuencial.

También es importante tener claramente identificados los roles de los componentes del equipo de proyecto que participarán en el desarrollo de la aplicación informática. A Métrica estos perfiles son:

- Partes interesadas (*stakeholders*)
- Jefe de Proyecto
- consultores
- analistas
- Desarrolladores

En la [figura 1.13](#) podéis observar las cinco fases principales de la metodología Métrica v3.0.

Figura 1.13. Fases de desarrollo de una aplicación



1.6.1. Estudio de viabilidad del sistema

El propósito de este proceso es analizar un conjunto concreto de necesidades, con la idea de proponer una solución a corto plazo. Los criterios con los que se hace esta propuesta no serán estratégicos sino tácticos y relacionados con aspectos económicos, técnicos, legales y operativos.

Los **resultados** del estudio de viabilidad del sistema constituirán la base para tomar la decisión de seguir adelante o abandonar el proyecto.

1.6.2. Análisis del sistema de información

El propósito de este proceso es conseguir la **especificación detallada** del sistema de información, por medio de un catálogo de requisitos y de una serie de modelos que cubran las necesidades de información de los usuarios para los que se desarrollará el sistema de información y que serán la entrada para el proceso de diseño del sistema de información.

ENTORNOS DE DESARROLLO

En primer lugar, se describe el sistema de información, a partir de la información obtenida en el estudio de viabilidad. Se delimita su alcance, se genera un catálogo de requisitos generales y se describe el sistema mediante unos modelos iniciales de alto nivel.

Se recogen de forma detallada los requisitos funcionales que el sistema de información debe cubrir. Además, se identifican los requisitos no funcionales del sistema, es decir, las facilidades que debe proporcionar el sistema, y las restricciones a las que estará sometido, en cuanto a rendimiento, frecuencia de tratamiento, seguridad ...

Normalmente, para efectuar el análisis se suele elaborar los modelos *de casos de uso y de clases*, en desarrollos orientados a objetos, y *de datos y procesos* en desarrollos estructurados. Por otro lado, se aconseja llevar a cabo una definición de interfaces de usuario, ya que facilitará la comunicación con los usuarios clave.

1.6.3. Diseño del sistema de información

El propósito del **diseño** es obtener la definición de la arquitectura del sistema y del entorno tecnológico que le apoyará, junto con la especificación detallada de los componentes del sistema de información. A partir de esta información, se generan todas las especificaciones de construcción relativas al propio sistema, así como la especificación técnica del plan de pruebas, la definición de los requisitos de implantación y el diseño de los procedimientos de migración y carga inicial.

En el diseño se generan las especificaciones necesarias para la construcción del sistema de información, como por ejemplo:

- Los componentes del sistema (módulos o clases, según el caso) y de las estructuras de datos.
- Los procedimientos de migración y sus componentes asociados.
- La definición y revisión del plan de pruebas, y el diseño de las verificaciones de los niveles de prueba establecidos.
- El catálogo de excepciones, que permite establecer un conjunto de verificaciones relacionadas con el propio diseño o con la arquitectura del sistema.
- La especificación de los requisitos de implantación.

1.6.4. Construcción del sistema de información

La **construcción del sistema de información** tiene como objetivo final la construcción y la prueba de los diferentes componentes del sistema de información, a partir de su conjunto de especificaciones lógicas y físicas, obtenido en la fase de diseño. Se desarrollan los procedimientos de operación y de seguridad, y se elaboran los manuales de usuario final y de explotación, estos últimos cuando proceda.

ENTORNOS DE DESARROLLO

del sistema de información y se van llevando a cabo, a medida que se vaya finalizando la construcción, las pruebas unitarias de cada uno de ellos y las de integración entre subsistemas. Si fuera necesario efectuar una migración de datos, es en este proceso donde se lleva a cabo la construcción de los componentes de migración y de los procedimientos de migración y carga inicial de datos.

1.6.5. Implantación y aceptación del sistema

Este proceso tiene como objetivo principal el **entrega** y la **aceptación** del sistema en su totalidad, que puede comprender varios sistemas de información desarrollados de manera independiente, y un segundo objetivo, que es llevar a cabo las actividades oportunas para el paso a producción del sistema.

Una vez revisada la estrategia de implantación, se establece el plan de implantación y se detalla el equipo que lo llevará a cabo.

Para el inicio de este proceso se toman como punto de partida los componentes del sistema probados de forma unitaria e integrados en el proceso de construcción, así como la documentación asociada. El sistema debe someterse a las pruebas de implantación con la participación del usuario de operación. La responsabilidad, entre otros aspectos, es comprobar el comportamiento del sistema bajo las condiciones más extremas. El sistema también será sometido a las pruebas de aceptación, que serán llevadas a cabo por el usuario final.

En este proceso se elabora el plan de mantenimiento del sistema, por lo que el responsable del mantenimiento conozca el sistema antes de que éste pase a producción.

También se establece el acuerdo de nivel de servicio requerido una vez que se inicie la producción. El acuerdo de nivel de servicio se refiere a servicios de gestión de operaciones, de apoyo a usuarios y el nivel con el que se prestarán estos servicios.