

campusformacion.imf.com © EDICIONES ROBLE, S.L.
Manuel Vázquez Enriquez

Desarrollo de software © EDICIONES ROBLE, S.L.

campusformacion.imf.com © EDICIONES ROBLE, S.L.
Manuel Vázquez Enriquez

campusformacion.imf.com © EDICIONES ROBLE, S.L.
Manuel Vázquez Enriquez


Indice

Desarrollo de software	3
I. Introducción	3
II. Objetivos	3
III. Contenidos	4
3.1. Concepto de programa informático	4
3.2. Ejecución de programas en ordenadores	5
3.2.1. El hardware y el software	5
3.2.2. Estructura funcional de un ordenador	6
3.2.3. Instrucciones y datos	7
3.2.4. La implementación	8
3.2.5. Tipos de software	9
3.2.6. Código fuente, código objeto y código ejecutable	10
3.3. Lenguajes de programación	12
3.3.1. Tipos de lenguajes de programación	12
3.3.2. Clasificación de lenguajes de programación por nivel	13
3.3.3. Clasificación de lenguajes de programación por generación	14
3.3.4. Clasificación de lenguajes de programación por técnica utilizada	15
3.3.5. Lenguajes de programación actuales	17
3.4. Ingeniería del software	17
3.4.1. Conceptos previos	17
3.4.2. Ciclo de vida del software	18
3.4.3. Procesos del ciclo de vida del software	19
3.4.4. Fases de desarrollo	21
3.4.5. Modelos de desarrollo	22
3.4.6. Herramientas CASE	25
3.4.7. Errores en el desarrollo	28
3.4.8. Reutilización del código	31
IV. Actividades interactivas	32
Actividad 1: Completa el siguiente crucigrama	32
Actividad 2: Relaciona los siguientes elementos	32
V. Resumen	33
VI. Lecturas obligatorias	33
Ejercicios	34
Ejercicio 1	34
Se pide	34
Elementos a tener en cuenta	34
Recursos	36
Enlaces de Interés	36
Bibliografía	37
Glosario.	37

Desarrollo de *software*

I. Introducción

Podemos ver un *software* como una *herramienta* que sirve para realizar nuestro *trabajo*, facilitarnos la búsqueda de información o simplemente para ser usado por *ocio*. Son muchas las cosas que hacemos hoy gracias a los programas informáticos, ya sea desde nuestros ordenadores o dispositivos móviles.

Existen *muchos tipos de software*, unos se instalan localmente en *nuestros ordenadores*, otros funcionan *bajo* un *servidor* con el que accedemos por medio de un navegador. Pero las preguntas que nos planteamos en esta unidad son: ¿qué hay detrás de estos programas?, ¿cómo se construyó esta aplicación? Hay un gran trabajo detrás de cada pantalla, cuando pulsamos un botón. 

Todo *proyecto de software* tiene un *ciclo de vida* para desarrollarse y *consta de* unos *pasos* que se van completando en diferentes tiempos. Este ciclo de desarrollo *depende* directamente de la *metodología* que utilizamos para el mismo, y no *es* más que *una estrategia que debemos seguir*.

II. Objetivos

Los objetivos de aprendizaje del alumno para esta unidad son los siguientes:

Identificar los tipos de los lenguajes de programación.

Diferenciar los diferentes paradigmas de programación.

Enumerar las fases del ciclo de vida de desarrollo de *unsoftware*.

Conocer las metodologías de desarrollo.

Identificar las diferentes herramientas para aumentar la producción.

Diferenciar el origen de los tipos de errores que pueden cometerse en el desarrollo de un programa.

III. Contenidos

3.1. Concepto de programa informático

Los ordenadores son un conjunto de elementos de *hardware* muy complejos, compuestos por muchos elementos que están comunicados entre sí por medio de circuitos.

Para llevar a cabo todas sus funciones, son necesarios programas que le indiquen qué hacer de una forma ordenada. Todo ello siguiendo una serie de procesos que interactúan con el usuario.

Estos programas son capaces de hacer tres tipos de operaciones:

1

Aritméticas.

2

Lógicas (comparación de valores).

3

De almacenaje de la información.



Ligando y combinando estas operaciones, se forma un programa. Por lo tanto, podemos describir el concepto de programa como: **un conjunto de órdenes que se ejecutan en el ordenador para conseguir un objetivo.**

Estas órdenes se proporcionan a través de un código que, mediante algoritmos escritos en un determinado lenguaje de programación, forman un conjunto de instrucciones que el *hardware* reconoce y ejecuta.

El ordenador siempre funciona bajo control de un programa, incluso en las operaciones más básicas que realiza, como comunicarse con los dispositivos de entrada/salida, interactuar con el usuario, gestionar los propios recursos del ordenador...

Algoritmo

Conjunto de procedimientos con los que se consigue una acción. Pueden estar compuestos a través de textos, números o símbolos. La expresión de uno o más algoritmos es lo que se define como programa.

Programa

Conjunto de instrucciones que procesa un ordenador con el fin de obtener un resultado. Estos programas pueden estar codificados en diferentes lenguajes de programación.

Lenguajes de programación

Notación para escribir un programa. Está definido por una gramática y tienen un léxico, una sintaxis y una semántica propia.

3.2. Ejecución de programas en ordenadores

3.2.1. El *hardware* y el *software*

Un ordenador se compone de una serie de elementos físicos que, mediante impulsos eléctricos, es capaz de realizar operaciones con un objetivo determinado. Estos elementos están interconectados entre ellos y cada uno tiene una función determinada.



El microprocesador es el encargado de realizar las operaciones. La RAM es la memoria primaria donde guardar los datos que se están procesando; el disco duro es el lugar donde almacenar datos de forma permanente, etc.

Pero todos estos elementos no tienen ninguna función si no podemos indicarles las tareas que deben realizar, es decir, darle los datos e instrucciones para que calculen y transmitan los resultados. Estas instrucciones son proporcionadas por el *software*. Así pues, el *hardware* necesita de un *software* para funcionar.

3.2.2. Estructura funcional de un ordenador

Un **ordenador** realiza las **operaciones** a partir de unos **datos**, que proceden de un sistema de almacenamiento o son introducidos por el usuario. Por ello, **necesita componentes (periféricos)** de **entrada** y de **salida** de datos para mostrar los resultados. Además, debe facilitar el desarrollo de estas actividades, agilizando los procesos y simplificando el uso.

periféricos de entrada

Como periféricos de entrada tenemos: teclado, ratón, escáner, cámara fotográfica... Estos, entre otros, nos ayudan a introducir información dentro de nuestro ordenador para ser procesada.

periféricos de salida

Los periféricos de salida son con los que el ordenador nos entrega la información en un formato que podamos comprender.



Monitor, impresoras, altavoces...

periféricos mixtos

También existen periféricos mixtos, capaces de recibir información y también de enviarla, como las **tarjetas de red**.

Todos estos dispositivos son controlados por un núcleo central, donde se encuentran los componentes básicos para el procesamiento (CPU, memoria RAM, discos duros...).

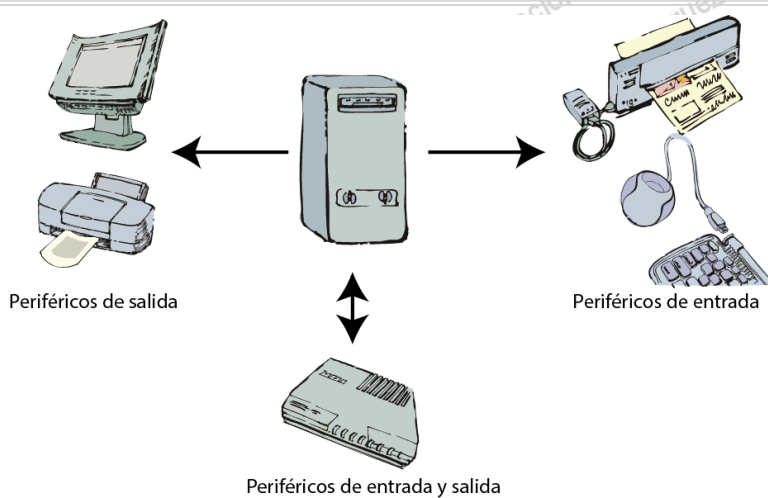
Imagen 1. Esquema de componentes periféricos de entrada y salida

Imagen 1. Esquema de componentes periféricos de entrada y salida.
Fuente: elaboración propia.

3.2.3. Instrucciones y datos

1

Un ordenador trabaja con un sistema de numeración en base 2, lo que significa que solo es capaz de trabajar con cadenas de números de dos estados o, lo que es lo mismo, con 1 y 0 (con tensión y sin tensión).

Dada la velocidad de los procesadores de hoy en día, esto no les impide realizar cálculos complejos. Pero a los programadores les resulta muy difícil codificar algoritmos con este sistema.

2

Por eso, los microprocesadores facilitan una serie de instrucciones para realizar operaciones matemáticas, comparaciones o saltos de una forma más sencilla para el programador. Uno de estos primeros **lenguajes es el ensamblador**, que utiliza directamente las instrucciones del microprocesador para definir los algoritmos, es decir, las operaciones que queremos que realice nuestro programa.



Ejemplo de código lenguaje ensamblador

Este ejemplo está completamente desarrollado en lenguaje ensamblador que usa servicios o funciones de MS-DOS (*system calls*) para imprimir el mensaje "¡Hola mundo!" en pantalla.

```

STACK SEGMENT STACK ; Segmento de pila

DW 64 DUP (?) ; Define espacio en la pila

STACK ENDS ; Segmento de datos

SALUDO DB "¡Hola mundo!",13,10,"$" ; Cadena

DATA ENDS

CODE SEGMENT ; Segmento de Código

ASSUME CS:CODE, DS:DATA, SS:STACK

INICIO: ; Punto de entrada al programa

MOV AX,DATA ; Pone dirección en AX

MOV DS,AX ;Pone la dirección en los registros

MOV DX,OFFSET SALUDO ; Obtiene dirección del
mensaje

MOV AH,09H ; Función: Visualizar cadena

INT 21H ; Servicio: Funciones alto nivel DOS

MOV AH,4CH ; Función: Terminar

INT 21H

CODE ENDS

END INICIO ; Marca fin y define INICIO

```

El lenguaje ensamblador está considerado como uno de los más difíciles de codificar. Aunque aún se utiliza en determinados sectores para trabajar a un bajo nivel (se comunica con el *hardware* sin intermediarios), hoy en día la mayor parte de las aplicaciones, las realizamos con lenguajes de programación más abstractos, como Java o C.

3.2.4. La implementación

Como hemos comentado, para la realización de un programa, lo primero es **codificar algoritmos y estructuras de datos en un lenguaje de programación.**



La implementación es la codificación de un algoritmo en un determinado lenguaje, que finalmente se comunica con la computadora y produce la ejecución de un programa.

Esta codificación se puede realizar en **lenguajes con comandos y funciones cercanas** al modo en el que una computadora procesa los datos (lenguajes de bajo nivel), como puede ser el código máquina o el lenguaje ensamblador. O en lenguajes con una sintaxis más cercana a la forma en la que nos comunicamos los humanos (lenguajes de alto nivel), que nos facilitan la representación de los procesos a realizar.

3.2.5. Tipos de *software*

El ordenador, para realizar las tareas que le demandamos actualmente, necesita programas que trabajen conjuntamente con objetivos comunes. Estos tienen diversas funciones (comunicarse con los periféricos, realizar cálculos, almacenar información...).

El *software* se suele clasificar en **tres grandes grupos**:

sistema operativo

Encargado de comunicarse directamente con el *hardware*, haciendo que todos los periféricos realicen las tareas. Es el intermediario entre las aplicaciones, el *hardware* y el usuario.



Imagen 2. Capas en las que trabaja el sistema operativo.

Fuente: elaboración propia.

Un sistema operativo es un *software* complejo y tiene una estructura por capas, dependiendo de las funciones que tenga que realizar. Estas comunican el *hardware*, que está en el nivel base, a través de bits o la capa binaria (unos y ceros).

aplicaciones

Programas que tienen funciones específicas que se instalan en un ordenador. Tienen que estar programados para un sistema operativo determinado, pues este será el encargado de atender las diferentes peticiones que realice.

Desde el punto de vista de un desarrollador de software, es importante saber en qué nivel estamos trabajando. No es lo mismo realizar un programa con un sistema operativo que se comunica directamente con el hardware; un aplicativo que funciona bajo un sistema operativo, del cual tendremos que conocer su estructura, librerías y posibilidades...

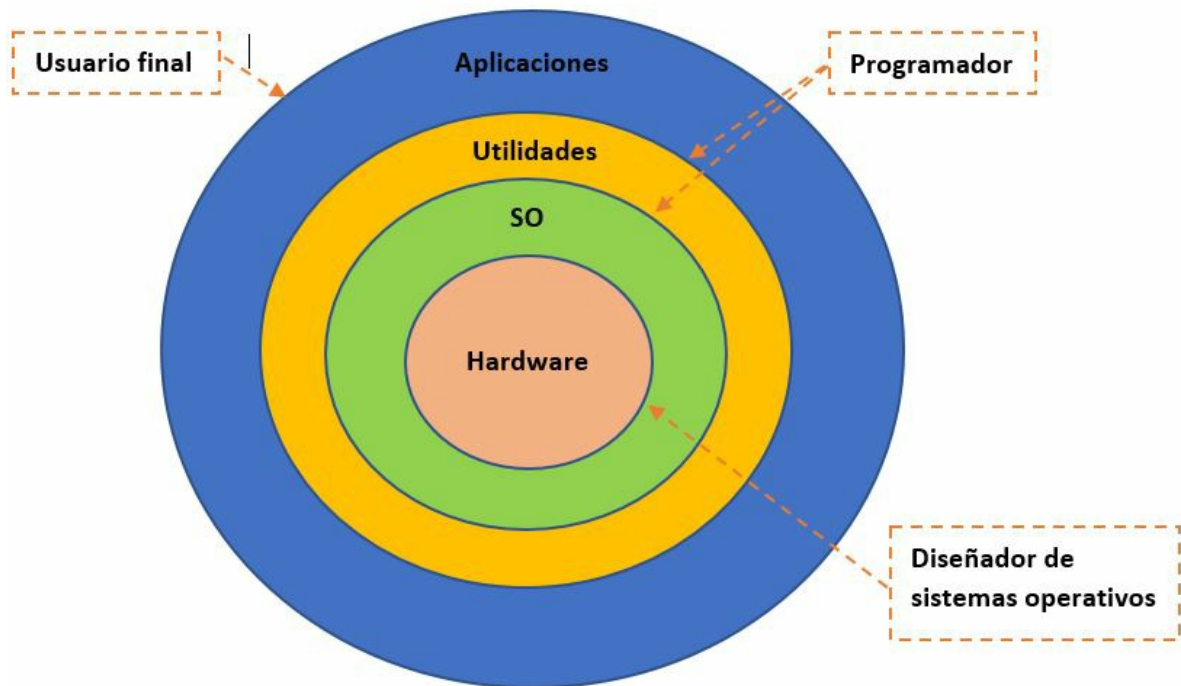


Imagen 3. Niveles de acceso a un ordenador.
Fuente: elaboración propia.

drivers

Grupo de programas que, aunque también se podrían calificar como aplicaciones de usuario, suelen diferenciarse del resto porque su función es indicar al sistema operativo cómo debe operar con periféricos que desconoce. Suele suceder cuando el dispositivo se fabricó después de que el sistema operativo (SO) se desarrollara. Actualmente, las compañías desarrolladoras realizan actualizaciones periódicas de su software, donde incluye drivers de dispositivos, además de parches o mejoras.

Si el sistema operativo no detecta un periférico, deberemos ir a la página web del fabricante. Allí suele estar el driver adecuado para que funcione en nuestro sistema.

3.2.6. Código fuente, código objeto y código ejecutable

Como ya hemos comentado, un programa es una secuencia de instrucciones escritas en un lenguaje de programación específico. El **programa** se **escribe** en el **lenguaje elegido por el programador** y este genera el **código fuente** (código legible para humanos).

El ordenador no entiende este código, así que deberá convertirse para que sí lo sea. Una vez traducido, el programa podrá ser ejecutado en el SO.

Código fuente

Es el conjunto de **instrucciones escritas en un lenguaje de programación**. Existen muchos lenguajes: **JAVA, PHP, C, .Net...**

Estos **códigos no pueden ser directamente ejecutados por la computadora**. Los **códigos** creados son **compilados** posteriormente **o interpretados** para su ejecución en los diferentes SO. Este proceso de traducción genera otro tipo de código, el código objeto.

Todo **código fuente** debe:

- Ser **fácil de leer y entendible** por otros desarrolladores.
- Tener **comentarios** que expliquen su finalidad y sus componentes.
- **Realizar** los **procesos** de la forma más **simple** posible.
- Ser **flexible a cambios** para posteriores **mantenimientos**.
- Ser **funcional**.



El 80 % de la vida útil de un trozo de software se produce en el mantenimiento.

Código objeto

Código que el **ordenador entiende** mejor porque está **escrito** en su idioma, es decir, en **código máquina o binario**. Es la traducción del código fuente escrito por el programador en cualquier lenguaje.

El código objeto ya forma parte del programa pero necesita ser compilado para enlazarse con otros y crear el archivo ejecutable, que será el encargado de poner en marcha nuestro programa. En este proceso, además, el compilador comprueba que el código no tenga errores y esté listo para funcionar.

Código ejecutable

Último proceso por el que pasa nuestro programa. Partiendo de los dos códigos anteriores, **crea los archivos ejecutables para que nuestro sistema reconozca la aplicación**. Estos serán diferentes para las diversas plataformas (Windows, Mac OS, Linux...).

Imagen 4. Código fuente, objeto y ejecutable

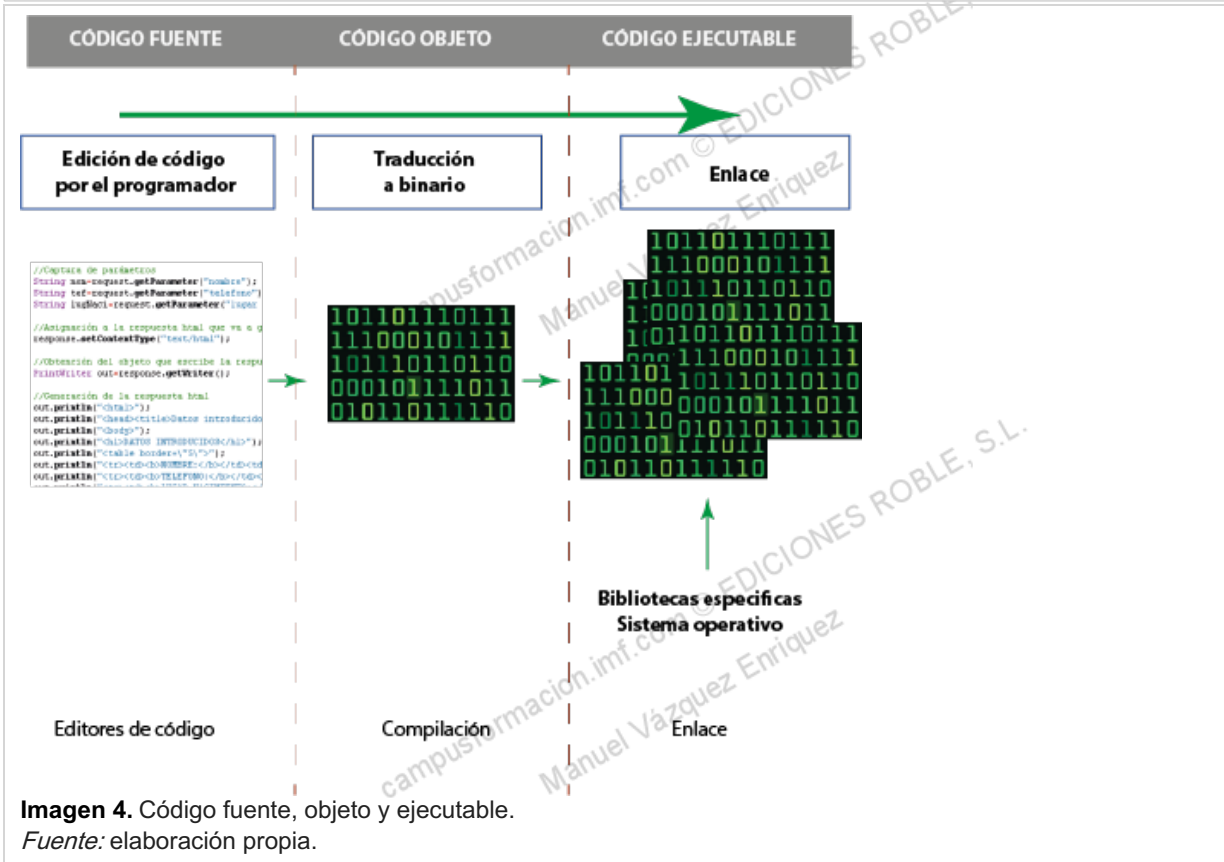


Imagen 4. Código fuente, objeto y ejecutable.
Fuente: elaboración propia.

3.3. Lenguajes de programación

3.3.1. Tipos de lenguajes de programación

Los programadores tienen que escribir el código de los programas en un lenguaje de programación. Estos pueden ser de diferentes tipos y trabajar en niveles distintos.

Lenguajes de alto nivel

Actualmente, los lenguajes más modernos buscan una sintaxis que el humano pueda escribir e interpretar. Estos lenguajes son **considerados de alto nivel**.

Lenguajes de bajo nivel

Al contrario de los que tienen una sintaxis para que la máquina lo lea, los **considerados de bajo nivel**, como el código máquina.

Un ordenador utiliza la codificación binaria para comunicarse. Para un programador, ese sistema resulta muy difícil al realizar estructuras complejas. Por ello, lenguajes como Java o C++ intentan asemejarse más al modo en que nosotros nos comunicamos.



Un ejemplo de esto es el pseudocódigo, un falso lenguaje con el que los programadores plantean sus algoritmos, sin pensar en un lenguaje de programación específico.

Algoritmo de DIJKSTRA en pseudocódigo	Macro realizada en ensamblador
<p>DIJKSTRA (Grafo G, nodo s)</p> <p>para $u \in V[G]$ hacer</p> <p> distancia[u] = INFINITO</p> <p> padre[u] = NULL</p> <p> visto[u] = false</p> <p>distancia[s] = 0</p> <p>adicionar (cola, (s, distancia[s]))</p> <p>mientras que cola no es vacía hacer</p> <p> u = extraer_mínimo(cola)</p> <p> visto[u] = true</p> <p> para todos $v \in \text{adyacencia}[u]$ hacer</p> <p> si no visto[v] y distancia[v] > distancia[u] + peso (u, v) hacer</p> <p> distancia[v] = distancia[u] + peso (u, v)</p> <p> padre[v] = u</p> <p> adicionar(cola,(v, distancia[v]))</p>	<p>IP = 0xFFFF</p> <p>SR = 0xFFFE</p> <p>OUT = 0xFFFFD</p> <p>mem[IP] = 0</p> <p>while True:</p> <p> i = mem[IP]</p> <p> a = mem[i+0]</p> <p> b = mem[i+1]</p> <p> r = mem[i+2]</p> <p> mem[IP] = i+3</p> <p> f = nor(mem[a], mem[b])</p> <p> mem[r] = f</p> <p> mem[SR] = ((f >> 15) & 1) ((f & 0x7FFF) << 1)</p> <p> if mem[IP] >= OUT:</p> <p> break</p> <p>print "OUT: ", mem[OUT]</p>

Imagen 5. Pseudocódigo.

Fuente: elaboración propia.



Como se puede ver, resulta más fácil interpretar la acción que realiza el algoritmo mostrado en pseudocódigo que el que está codificado en lenguaje ensamblador.

3.3.2. Clasificación de lenguajes de programación por nivel

Podemos realizar una clasificación de los lenguajes según el nivel, es decir, la cercanía de las instrucciones del lenguaje de programación con el lenguaje humano.

Se pueden agrupar en **tres tipos**:

Lenguajes de bajo nivel

Lenguajes totalmente orientados a la máquina, específicos para un tipo en concreto. No se pueden usar en otras con diferente arquitectura.

Al estar prácticamente diseñados a medida del *hardware*, aprovechan al máximo las características de este.

Lenguaje máquina

Combinación de números binarios.

Lenguajes ensambladores

Con instrucciones concretas definidas en el propio *hardware* de la máquina.

Lenguajes de nivel intermedio

Este nivel no es aceptado por todos, aunque se utiliza frecuentemente para clasificar lenguajes que pueden acceder a los registros del sistema, trabajar con direcciones de memoria como C.



C, Eiffel, Sather, Esterel, algunos dialectos de Lisp...

Lenguajes de alto nivel

Aquellos que se encuentran **más cercanos al lenguaje natural que al máquina.**

Más fácil de interpretar. Podemos utilizarlos en diferentes plataformas, aunque necesitan ser traducidos al lenguaje de máquina para ejecutarse.



Lenguajes de propósito general (cualquier tipo de aplicación) y de propósito específico (Java, PHP, C+...).

3.3.3. Clasificación de lenguajes de programación por generación

Otra forma de clasificar los lenguajes de programación es por las distintas etapas en la creación.

primera generación

En los **inicios** de la **informática**, para los primeros ordenadores, se genera el lenguaje que habla directamente con la CPU del computador: **lenguaje máquina**. Esto puede ser representado en series de números binarios almacenados en memoria, o las antiguas tarjetas perforadas.

segunda generación

Para facilitar la creación de programas, los ordenadores incluyeron un lenguaje con una **serie** de **instrucciones** para realizar los algoritmos, el **lenguaje ensamblador**. Aunque comparado con lenguajes actuales, puede parecer complicado, en aquel momento fue un gran avance, dado lo complejo que resultaba trabajar con unos y ceros (con interruptores).

tercera generación

La **abstracción** permite **crear los primeros lenguajes de alto nivel**: C, Pascal, Cobol, JAVA. Estos cuentan con una sintaxis que resulta más natural al programador. Sustituyen las instrucciones simbólicas por códigos independientes de la máquina, similar al lenguaje humano o al de las matemáticas.

cuarta generación

Con la entrada de las **herramientas CASE** y los **entornos de desarrollo**, se generan **lenguajes capaces de generar código por ellos mismos**, los llamados RAD, con lo que se pueden realizar aplicaciones sin ser un experto en el lenguaje.

quinta generación

Lenguajes orientados a la inteligencia artificial Esta generación aún no ha sido desarrollada.



Un ejemplo es el lenguaje LISP.

3.3.4. Clasificación de lenguajes de programación por técnica utilizada

Por último, podemos clasificar los lenguajes de programación según su modelo de actuación o paradigma utilizado.

Encontramos lenguajes de programación que utilizan técnicas diferentes para construir el código. Los **más comunes** son los siguientes:

Secuencial

Programas que ejecutan el código de forma secuencial, es decir, va ejecutando las sentencias en orden una detrás de otra.

Estructurada

Utiliza estructuras de control, es una programación secuencial pues ejecuta una seguida de otra, pero incorpora estructuras selectivas para la toma de decisiones y estructuras repetitivas. La mayor parte de los lenguajes actuales derivan de este tipo de programación, adaptándose a otros paradigmas.

Modular

La programación modular consta de varios módulos que interactúan entre sí. Un módulo principal coordina las llamadas al resto de módulos y pasa los datos necesarios por parámetros.

Programación orientada a objetos (POO)

Una de las técnicas más extendidas hoy en día, donde cada conjunto de datos es un objeto con sus atributos y métodos.

Esta técnica aumenta considerablemente la velocidad del desarrollo de programas, gracias a la reutilización de las partes del código.

Todas estas técnicas se pueden subdividir también en **dos grandes grupos**:

Programación imperativa

Se escriben sentencias que modifican el estado de un programa. En este paradigma, se indica detalladamente cada uno de los pasos y la toma de decisiones que deben ocurrir en la ejecución.

Dentro de esta categoría están la **programación estructurada**, **programación modular** y **programación orientada a objetos**.



BASIC, C, Fortran, Pascal, Perl, PHP, Java, Python, Javascript...

Programación declarativa

La solución se alcanza mediante **procesos internos del lenguaje**, sin especificar exactamente el proceso para llevarlo a cabo.



Haskell, ML, Lisp, Prolog, SQL, QML...

3.3.5. Lenguajes de programación actuales

Actualmente son muchos los lenguajes utilizados por los programadores, unos por su evolución y facilidad de programación y otros por los requisitos que el programa pide realizar en cada caso.



Por ejemplo, si queremos realizar una aplicación web, podremos usar lenguajes como Java, PHP, .NET..., además de otros como JavaScript y HTML. Pero si queremos realizar una aplicación para móviles, tendremos que hacerla en Android o Swift.

No se puede indicar cuál de los lenguajes es mejor, depende siempre del programador o tipo de programa, además su popularidad o uso cambian anualmente.



Ver clasificación de lenguajes de programación: <https://tiobe.com/tiobe-index>

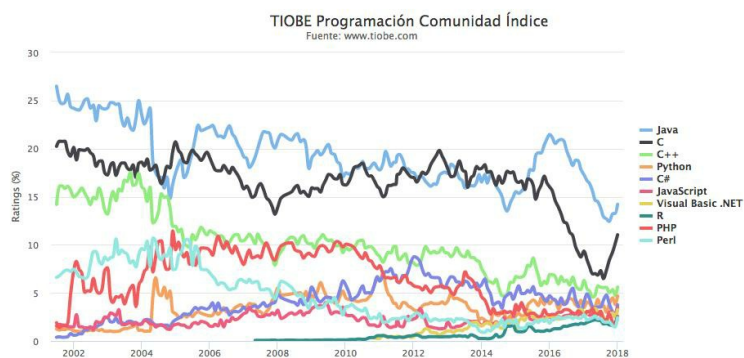


Imagen 6. Comunidades de desarrolladores según lenguaje de programación.

Fuente: www.tiobe.com

3.4. Ingeniería del *software*

3.4.1. Conceptos previos



La ingeniería del *software* comprende todos los aspectos de la creación de este, desde sus procesos iniciales (definición) hasta el mantenimiento de este una vez se está utilizando. Muchas personas asocian el término *software* con programas de ordenador, pero realmente no es solo el código final, sino también todos los documentos asociados y la configuración de los datos para su funcionamiento.

En la creación de un *software*, debemos seguir diferentes procesos y utilizar herramientas que no solo sirven para la generación del propio código.

Para comenzar, vamos a responder algunas de las cuestiones que se plantea un desarrollador al crear un programa:

¿Qué es el software?

Programas de ordenador y documentación asociada. Los productos de *software* se pueden desarrollar para algún cliente en particular o para un mercado general.

¿Qué es un proceso de software?

Un conjunto de actividades cuya meta es el desarrollo o evolución del *software*.

¿Cuál es el coste de un desarrollo de software?

A grandes rasgos, el 60 % de los costos son de desarrollo, el 40 % restante son de pruebas. En el caso del *software* personalizado, los costos de evolución a menudo exceden los de desarrollo.

¿Qué son los métodos de desarrollo?

Enfoques estructurados para el desarrollo de *software* que incluyen modelos de sistemas, notaciones, reglas, sugerencias de diseño y guías de procesos.

¿Qué es el CASE?

Herramientas que intentan proporcionar ayuda automatizada a las actividades del proceso del *software*.

3.4.2. Ciclo de vida del *software*



La norma "ISO/IEC Standard 12207:2008: *software* life-cycle processes", propuesta por la ISO (International Organization for Standardization), define el ciclo de vida del *software* como, "un marco de referencia que contiene los procesos, las actividades y las tareas involucradas en el desarrollo, explotación y mantenimiento de un producto *software*, abarcando la vida del sistema desde la definición de requisitos hasta que se deja de utilizar".

Cada etapa del desarrollo lleva asociadas una serie de actividades que debemos realizar en cada una de ellas, dispondremos de unas herramientas para ayudarnos y tendremos que generar una serie de documentos que servirán como guía a la siguiente fase de desarrollo.

Según la norma ISO, las actividades que debemos llevar a cabo durante el ciclo de vida de un *software* son:

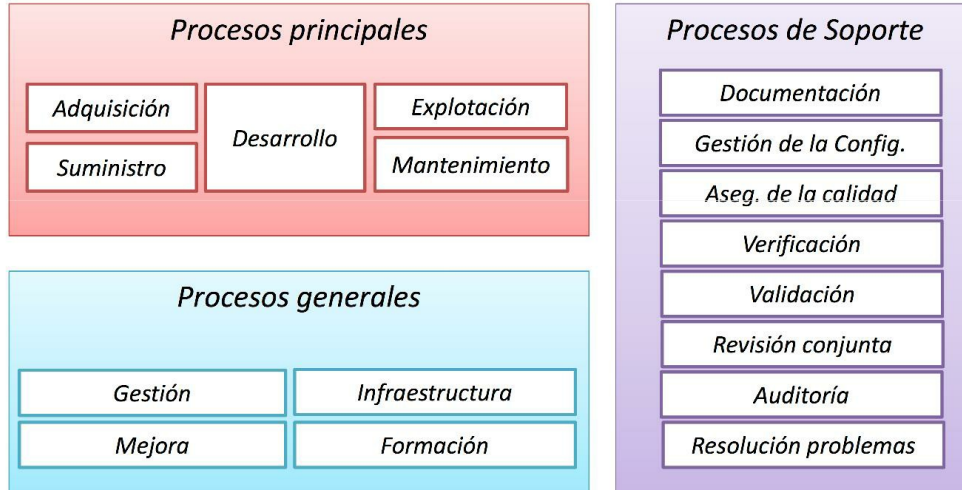


Imagen 7. Actividades durante el ciclo de vida de un *software*.

Fuente: ISO/IEC Standard 12207:2008: *software* life-cycle processes.

3.4.3. Procesos del ciclo de vida del *software*

Procesos principales
Adquisición <p>Preparación de la oferta para suministrar el <i>software</i>, seleccionaremos los agentes implicados en su realización.</p>
Suministro <p>Contratos, identificación de los recursos necesarios para llevar a cabo el desarrollo.</p>
Desarrollo <p>Actividades enfocadas a la creación del <i>software</i>, este es el punto en el que intervendremos más como desarrolladores, pues es la parte en la que se produce. Está subdividida en las actividades de análisis, diseño, codificación, pruebas, integración e implantación.</p>
Explotación <p>Tiempo en el que la aplicación está en uso, en esta fase se da soporte operativo a los usuarios.</p>

Mantenimiento

Todo producto necesita un mantenimiento, más si se trata de un *software*. En esta fase corregiremos posibles errores, aplicaremos mejoras y realizaremos adaptaciones del producto para que se adecúe a diferentes cambios del *hardware* o de los SO.

Procesos de soporte

Documentación

Registra la información de todas las tareas que se realizan en las diferentes fases y actividades del ciclo de vida.

Gestión de la configuración

Actividades que controlan las modificaciones y los cambios que se producen en las versiones de los elementos.

Aseguramiento de la calidad

Actividades para que el producto cumpla con los requisitos establecidos.

Verificación

Permiten determinar el correcto funcionamiento del producto.

Validación

Comprobar que el producto de *software* cumple con los requisitos establecidos, es decir, con los objetivos que se marcaron para su creación.

Revisión conjunta

Puesta en común de los diferentes grupos implicados en el proyecto para determinar y revisar las diferentes fases del ciclo de vida.

Auditorías

Actividades que se realizan en un determinado momento para comprobar que se están consiguiendo los objetivos propuestos: requisitos, cumplimiento de los compromisos establecidos en el contrato, comprobar los plazos...

Resolución de problemas

Resolver problemas o disconformidades con los requisitos o el contrato, surgidos durante la ejecución del proyecto.

Procesos generales

Gestión

Planificación, seguimiento, revisiones...

Infraestructura

Recursos para la puesta en marcha del producto de *software*, tanto para su desarrollo como implantación y soporte futuro. Tanto en instalaciones, *hardware*, *software*, consumos...

Mejora

Evaluar y mejorar cada uno de los procesos de vida del *software*.

Formación

Planes de formación para todos los agentes implicados en el proceso del ciclo de vida.

3.4.4. Fases de desarrollo

Todos los procesos enumerados en el punto anterior representan los tipos de acciones que debemos llevar a cabo en el ciclo de vida de un *software*. Ahora bien, **tenemos que establecer el orden** en que se realizarán estas tareas. Para ello, disponemos de multitud de modelos o estrategias de desarrollo.

Estas estrategias, que ayudan a organizar las diferentes etapas y actividades del ciclo de vida, diferenciando las del ciclo de vida del *software*, también se les llama modelos de ciclo de vida del *software*.

En la figura, vemos cada uno de los procesos (principales, de apoyo y de organización), en orden secuencial, que deben realizarse en un desarrollo.



Imagen 8. Procesos a realizar en un desarrollo.

Fuente: elaboración propia.

Cada una de estas actividades está compuesta por diferentes tareas.

3.4.5. Modelos de desarrollo



Un modelo de desarrollo es la representación abstracta de los diferentes procesos que tenemos que realizar durante el ciclo de vida del *software*.

En el punto anterior, hablábamos de estrategias. Cada modelo de *software* busca el mejor modo de llevar a cabo los objetivos. A continuación, estudiaremos los más usados en las empresas, aunque no existe un modelo ideal ya que cada desarrollo y entorno de trabajo puede adaptarse a uno u otro.

Modelo en cascada

Este modelo está dividido en **etapas que se realizan secuencialmente**, es similar al proceso natural del ciclo de vida. Es uno de los modelos más utilizados y de los más antiguos, sirviendo de base a muchos otros más actuales.

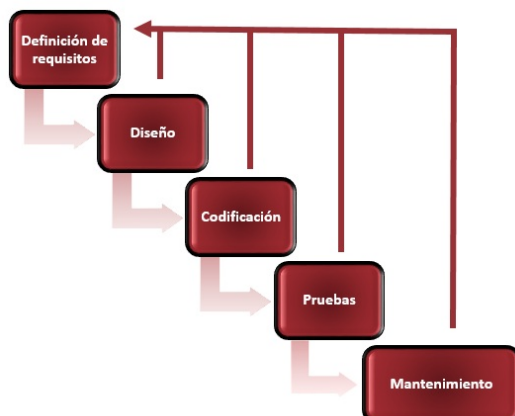


Imagen 9. Modelo en cascada (waterfall).

Fuente: elaboración propia.

Una etapa no puede empezar hasta que no haya terminado la anterior. En cada una de las fases, generaremos toda la documentación necesaria para explicar lo realizado y las bases para el siguiente paso. Es un **modelo dirigido por documentos**.

Definición de requisitos

Se profundiza en las necesidades del cliente y los requerimientos que debe cumplir el *software*. Marca los objetivos a cumplir y especifica el ámbito de la aplicación.

Diseño

Se describe el *software* y se planifican todas sus partes y procesos paso a paso. Es importante tener diseñado todo el *software* en esta fase para evitar problemas en las siguientes fases.

Codificación

Se escribe el código de nuestro *software*, siguiendo los pasos que se han definido en la fase de diseño.

Pruebas

Se realizan las pruebas necesarias para comprobar que *software* funciona como se ha establecido en los objetivos iniciales.

Mantenimiento

Modificaciones, corrección de errores no localizados en la fase de pruebas, mejoras de la aplicación, adecuaciones...

Ventajas

- Útil cuando se tienen claros los requisitos desde el principio.
- Útil con equipos de trabajo pequeños o inexpertos, pues tiene una estructura de trabajo muy definida y se apoya en mucha documentación.
- Útil cuando se realizan migraciones de *software*.

Inconvenientes

- Tener que definir al inicio todos los requisitos no es realista, pues suelen aparecer nuevas necesidades a lo largo del desarrollo.
- El cliente no ve el producto terminado hasta finalizar el proceso.
- Poco flexible a cambios.

Modelo iterativo

Es un modelo evolutivo que **se basa en prototipos que se pueden probar**. Esto ayuda a perfeccionar los requisitos del sistema.

Está compuesto por iteraciones y, en cada una de ellas, se revisan y mejoran las diferentes partes del producto hasta llegar a la solución final.



Imagen 10. Modelo Iterativo.

Fuente: elaboración propia.

Ventajas

- Se puede ir viendo los resultados durante el proceso.
- Útil cuando el cliente no sabe lo que quiere y los requisitos no están bien definidos desde el principio.
- Reduce el riesgo de no cumplir las necesidades de los usuarios.
- Útil cuando los requisitos cambian durante el proceso.

Inconvenientes

- El diseño del prototipo hace que los desarrolladores utilicen herramientas que faciliten la rápida generación de código, dejando a un lado la eficiencia, fiabilidad y otros aspectos de calidad.
- Probablemente no se tendrá un código óptimo.
- Exige disponer de las herramientas específicas.

Modelo incremental

Es un modelo evolutivo que **permite a los ingenieros desarrollar versiones cada vez más completas del producto**. Parte de los elementos del modelo en cascada, repitiendo los procesos con el objetivo de la construcción de prototipos.



Imagen 11. Modelo incremental.

Fuente: elaboración propia.

Ventajas

- Los clientes se involucran más en todo el proceso.
- Se puede ir revisando el producto en toda la fase de desarrollo.
- Fácil introducción de cambios.

Inconvenientes

- Los incrementos deben ser pequeños.
- Se necesita un grado muy alto de planificación.
- Difícil de documentar.

3.4.6. Herramientas CASE



Las herramientas CASE (*Computer Aided Software Engineering*) son aplicaciones que nos ayudan a aumentar la productividad en el desarrollo del *software*, reduciendo los tiempos de creación y el coste económico.

Dentro de los objetivos que se buscan al usar estas herramientas en el desarrollo, podemos destacar:

Mejorar la productividad.

Aumentar la calidad del producto.

Reducir el tiempo y los costes.

Mejorar la planificación en los procesos de desarrollo.

Aumentar la biblioteca de conocimiento.

Automatizar procesos en un desarrollo.

Ayudar a la reutilización del código.

Facilitar el uso de metodologías de desarrollo.

Clasificación

Las herramientas CASE pueden **clasificarse según la fase del ciclo de vida** en la que intervienen.

En un grupo principal, podemos clasificar las herramientas integradas, I-CASE (*Integrated CASE*, CASE integrado), que abarcan todas las fases del ciclo de vida del desarrollo de sistemas. Este grupo puede dividirse en dos:

Herramientas de alto nivel

Las herramientas de alto nivel, U-CASE (*Upper CASE*, CASE superior) o *front-end*, orientadas a la automatización y el soporte de las actividades desarrolladas durante las primeras fases del desarrollo: definición, análisis y diseño.

Herramientas de bajo nivel

Las herramientas de bajo nivel, L-CASE (*LowerCASE*, CASE inferior) o *back-end*, dirigidas a las últimas fases del desarrollo: implementación, pruebas y mantenimiento.

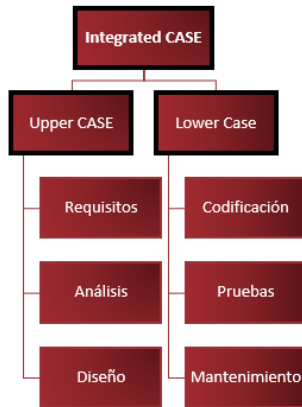


Imagen 12. Clasificación herramientas CASE.

Fuente: elaboración propia.

Ejemplos de uso

Feature Support	Visio	Rational Rose	Visible Analyst	Argo UML	Meta Edit+	mart draw	Visual Case	DB-Main	Omni graffle	ADO it
UPPER CASE										
Diagramas de flujo de datos	X	X	X	X	x	X	X		X	X
Meta Edit +					X					
Diagramas de flujo de trabajo	x	X	X	X	X	X	X		X	X
Diagramas orientados a objetos	X	X	X	X	X	X	X		X	X
Diccionario de datos	X	X	X	X	X		X	X		X
Adopción de reglas de negocio	X	X	X	X	X		X			X
Herramientas de análisis		X	X				X			X
LOWER CASE										
Generación de código	X	X	X	X	X		X	X		X
Generación de formularios	X	X	X		X					X
Generación de informes	X	X	X		X		X			X
Generación de documentación	X	X	X	X	X			X		X
Utilidades de importación y exportación	X	X	X	X	X			x		x
Codificación del programa		X	X		X		X			
Testing		X	x		x		x			

Imagen 13. Ejemplos de uso de las herramientas CASE.

Fuente: elaboración propia.

Son muchas las herramientas CASE de las que disponemos, cada una de ellas con diferentes funciones. A continuación, indicamos algunas de las más populares.



- [Rational Rose](#)
- [Visible Analyst](#)
- [ArgoUML](#)
- [MetaEdit+](#)
- [Smart Draw](#)
- [Visual Case](#)
- [DB-Main](#)
- [Omni graffle](#)
- [ADOit](#)
- [Visio](#)

3.4.7. Errores en el desarrollo

Los errores en el desarrollo pueden proceder de cualquiera de las fases del ciclo de vida. Una mala planificación, una expectativa muy alta, dotar de recursos insuficientes al proyecto... pueden hacer que el producto no llegue a buen término.

A continuación, indicamos algunos de los errores más comunes que se pueden producir, clasificándolos por el posible origen.

Referente a las personas

Motivación indeterminada: "La motivación es el mayor factor sobre la productividad y la calidad del desarrollo de software" (Boehm).
Personal poco capacitado: después de la motivación, las capacidades de los individuos o su relación como equipo son los siguientes factores en importancia en la productividad.
Falta de actuación con los elementos problemáticos.
Heroísmo: no interesan actuaciones heroicas, sino el desarrollo continuo, pausado y sin sobresaltos.
Añadir más personal a los proyectos retrasados.
Oficinas repletas y ruidosas. Los trabajadores que están en oficinas silenciosas tienden a funcionar mejor que los que ocupan cubículos en salas ruidosas y repletas.
Fricciones entre los clientes y los desarrolladores.
Expectativas poco realistas: según el <i>standish group</i> , las expectativas realistas son uno de los cinco factores necesarios para asegurar el éxito de los desarrollos de software internos.
Falta de un promotor efectivo del proyecto: ayuda a realizar una planificación realista, control de cambios y la introducción de nuevos métodos de desarrollo.
Falta de participación de los implicados: promotores, ejecutivos, responsables del equipo, miembros del equipo, personal de ventas, usuarios finales, clientes, marketing...
Falta de participación del usuario: según el <i>standish group</i> , la razón fundamental de que los proyectos de sistemas de información tengan éxito es la implicación del usuario.
Política sobre el contenido: Los grupos de desarrolladores tienen diferente personalidad igual que las personas. Podemos encontrar políticos (se concentran en las relaciones con los responsables), investigadores (su principal preocupación es investigar, buscar y recopilar más información), aislacionistas (buscan marcar su zona y aislarse del exterior)... Los equipos en los que predominan los políticos funcionan bien al principio pero luego pierden el interés de la dirección.
Buenos deseos: no es optimismo, es cerrar los ojos y esperar que algo funcione, sin tener bases razonables para pensarlo.

Imagen 14. Cuadro de errores referentes a las personas.

Fuente: S. McConnell, 1996. McGrawHill (Desarrollo y gestión de proyectos informáticos).

Referente al proceso de desarrollo

Planificaciones excesivamente optimistas: además reduce en gran medida la autoestima y la productividad de los desarrolladores.
Nula o poca gestión de riesgos.
Fallos de los subcontratistas (retrasos, baja calidad, no respuesta a los requisitos).
Planificación insuficiente.
Abandono de la planificación por un exceso de presión: se hacen planes pero se abandonan al aparecer las dificultades.
Perder el tiempo en los trámites iniciales: tiempo empleado en obtener la aprobación de la dirección y los fondos. Es más fácil no perder ese tiempo que recuperarlo al programar.
Saltar a codificar: eliminar las actividades iniciales “improductivas”. Ante la pregunta: ¿Enséñame la documentación de diseño?; la respuesta: “No ha habido tiempo de diseñar. Eliminamos el diseño, el análisis y los requisitos”.
Diseño inadecuado: no se tiene el tiempo ni la tranquilidad suficiente para llevarlo a cabo.
Eliminar las actividades de control de calidad: un ahorro de un día en calidad puede costar entre 3 y 10 días al final del proyecto.
Poco seguimiento del proyecto.
Forzar la convergencia de todas las actividades (módulos) del producto: si intentamos que todo encaje antes de tiempo, tendremos trabajo extra.
Omitir tareas importantes al hacer la estimación.
Mantener una planificación cuando ya se han producido retrasos: hay que actualizar la planificación cuando vamos teniendo más datos del producto (cambios en los requisitos).
Programación a destajo (code-like-hell): piensan que si los programadores están suficientemente motivados, salvarán cualquier obstáculo.

Imagen 15. Cuadro de errores referentes al proceso de desarrollo.

Fuente: S. McConnell, 1996. McGrawHill (Desarrollo y gestión de proyectos informáticos).

Referente al producto

Exceso de requisitos: se pide características complejas al sistema, que el usuario no necesita. Además, se da excesiva importancia al rendimiento.
Cambio en los requisitos: el proyecto medio experimenta un 25 % de cambio en los requisitos durante su desarrollo, lo que origina un 25 % de incremento en el tiempo de desarrollo.
Exceso de funcionalidades: los desarrolladores, fascinados por la tecnología u otros productos, añaden funcionalidades no solicitadas por el cliente.
Tiras y aflojas en la negociación: la dirección asume el retraso en el proyecto pero, a la vez, añade nuevas funcionalidades.
Desarrollo orientado a la investigación: Seymour Cray, diseñador de los ordenadores Cray, dice que nunca trata de superar los límites de la ingeniería en más de dos áreas simultáneamente, porque el riesgo de fallar es demasiado alto. La investigación no es predecible.

Imagen 3.16. Cuadro de errores referentes al producto.

Fuente: S. McConnell, 1996. McGrawHill (Desarrollo y gestión de proyectos informáticos).

Referente a la tecnología

Síndrome de la panacea: se confía demasiado en las nuevas tecnologías antes de ser probadas.
Exceso en la estimación de los ahorros de los nuevos métodos y herramientas. Raramente se producen grandes mejoras instantáneas en productividad. Los beneficios de las nuevas prácticas se ven reducidas por las curvas de aprendizaje asociadas con ellas.
Cambio de herramientas a mitad del proyecto.
Falta de control automático de código fuente.

Imagen 17. Cuadro de errores referentes a la tecnología.

Fuente: S. McConnell, 1996. McGrawHill (Desarrollo y gestión de proyectos informáticos).

3.4.8. Reutilización del código

En muchas ocasiones, los programas que realizamos proceden de un código que ya existe, o partimos de otro proyecto donde ya disponemos de algo de código que tenemos que adaptar. Esto puede suponer un considerable ahorro en el tiempo de desarrollo.

Es esencial **entender el código para reutilizarlo**, lo cual no es tan simple como parece ya que este puede haber sido escrito por varios desarrolladores o puede provenir de múltiples fuentes.

Cuando creamos un código, no solo tenemos que tener en cuenta el funcionamiento del mismo, también debemos hacerlo entendible por otros desarrolladores, realizando comentarios y estructurándolo de forma lógica y, a poder ser, lo más simple posible.

Las razones principales para **mantener un código legible** se pueden resumir en:

Documentación

Documentar el código para entenderlo mejor y para que esté preparado para la próxima vez que el desarrollador se retire o se mude (documentar implica la existencia de diagramas que expliquen el código).

Reutilización

Revisar o actualizar la aplicación a una versión más reciente.

Modificación

Mantener su código existente.

Mantenimiento

Reutilizar ese código o partes de él para nuevas aplicaciones o ampliaciones del mismo.



Para poder entender un código, una herramienta excelente es el lenguaje UML (*Unified Modeling Language*).

Este es un lenguaje gráfico estándar de representación de código, permite ilustrar las partes del código, indicando sus procesos, la forma en que interactúan... Está compuesto por diversos diagramas que, en su conjunto, describen todos los aspectos de un programa. Debido a que es un lenguaje, cuenta con reglas para combinar estos elementos.

Recordemos que un modelo es una representación simplificada de la realidad. El UML describe lo que supuestamente hará un sistema pero no dice cómo implementar dicho sistema. Algunos de los diagramas más comunes del UML son:

- Diagrama de clases.
- Diagrama de objetos.
- Diagrama de casos de uso.
- Diagrama de estados.
- Diagrama de secuencias.
- Diagrama de actividades.
- Diagrama de colaboraciones.
- Diagrama de componentes.
- Diagrama de distribución.
- Diagrama de paquetes.

IV. Actividades interactivas

Actividad 1: Completa el siguiente crucigrama



Crucigrama

Actividad 2: Relaciona los siguientes elementos



Relacionar elementos

V. Resumen



En esta unidad, hemos aprendido:

- **Qué es un programa** y cómo interactúa con el *hardware* de nuestro equipo; desde lo que escribimos en diferentes lenguajes hasta su traducción (que debe producirse para que la máquina lo entienda).
- Los diferentes **tipos de software dependiendo de la función que realicen** como los sistemas operativos o programas de gestión. Todos ellos realizados en diferentes lenguajes de programación, los cuales podemos clasificar por el nivel en el que interactúan con la máquina o por el tipo de programación que utilizamos.
- Además de los diferentes tipos de lenguajes, hemos visto como **todo software tiene un ciclo de vida**, desde que se plantea su necesidad hasta que se da soporte a los usuarios que lo utilizan. Para ello, disponemos de metodologías, que nos indican paso a paso las tareas que debemos realizar y el orden a seguir para poder concluir con éxito un proyecto.
- Las **herramientas CASE**, las cuales nos facilitarán el trabajo y aumentarán nuestra producción, reduciendo los costes de creación del *software*.

VI. Lecturas obligatorias



- Principales herramientas CASE del mercado y su uso.
- Herramientas para el desarrollo de Sistemas de Información.
- Introducción a las metodologías de desarrollo de software.
- Metodologías ágiles.

Ejercicios

Ejercicio 1

Duración estimada del ejercicio



180
minutos



Este ejercicio es **opcional**, si tienes alguna duda puedes contactar con tu tutor.

Dada la siguiente tabla con diferentes ciclos de vida, rellenar los campos con las ventajas e inconvenientes de cada uno.

Se pide

- Usar la tabla de conceptos para rellenar la información.
- Informarse en Internet de los conceptos asociados.
- Rellenar las celdas correspondientes.

Nombre	Función	Características
CASCADA		
INCREMENTAL		
EVOLUTIVO		
PROTOTIPADO		
ESPIRAL		
CONCURRENTE		
ITERATIVO		
MODELO EN V		
DESARROLLO RÁPIDO		

- Para ello deberá informarse en Internet de cada concepto.
- Se pueden añadir a la tabla los que se consideren oportunos, eso se valorará de manera positiva.

Elementos a tener en cuenta

- Se valorará la explicación de los conceptos y su explicación y extensión.
- Se valorará la aportación de conceptos nuevos.



Descarga aquí la [tabla de ciclos de vida](#).

campusformacion.imf.com © EDICIONES ROBLE,
Manuel Vázquez Enriquez

campusformacion.imf.com © EDICIONES ROBLE, S.L.
Manuel Vázquez Enriquez

campusformacion.imf.com © EDICIONES ROBLE, S.L.
Manuel Vázquez Enriquez

campusformacion.imf.com © EDICIONES ROBLE, S.L.
Manuel Vázquez Enriquez

Recursos

Enlaces de Interés



<http://www.visible.com/Products/Brochures/Analyst/Visible%20Analyst%20Product%20Overview.pdf>

<http://www.visible.com/Products/Brochures/Analyst/Visible%20Analyst%20Product%20Overview.pdf>



<http://argouml.tigris.org/>

<http://argouml.tigris.org/>



<http://www.metacase.com/mep/>

<http://www.metacase.com/mep/>



<https://www.smartdraw.com/>

<https://www.smartdraw.com/>



<http://visualcase.com/>

<http://visualcase.com/>



<http://www.db-main.eu/?q=en>

<http://www.db-main.eu/?q=en>



<https://www.omnigroup.com/omnigraffle>

<https://www.omnigroup.com/omnigraffle>



<http://www.adoit-community.com/>

<http://www.adoit-community.com/>



<http://www-03.ibm.com/software/products/es/enterprise>

<http://www-03.ibm.com/software/products/es/enterprise>



<https://products.office.com/es-es/visio/flowchart-software>

<https://products.office.com/es-es/visio/flowchart-software>



<https://products.office.com/es-es/visio/flowchart-software>

<https://products.office.com/es-es/visio/flowchart-software>



<http://www-03.ibm.com/software/products/es/enterprise>

<http://www-03.ibm.com/software/products/es/enterprise>



<http://www.visible.com/Products/Brochures/Analyst/Visible%20Analyst%20Product%20Overview.pdf>

<http://www.visible.com/Products/Brochures/Analyst/Visible%20Analyst%20Product%20Overview.pdf>



<http://argouml.tigris.org/>

<http://argouml.tigris.org/>



<http://www.metacase.com/mep/>

<http://www.metacase.com/mep/>



<https://www.smartdraw.com/>

<https://www.smartdraw.com/>



<http://visualcase.com/>

<http://visualcase.com/>



<http://www.db-main.eu/?q=en>
<http://www.db-main.eu/?q=en>



<https://www.omnigroup.com/omnigraffle>
<https://www.omnigroup.com/omnigraffle>



<http://www.adoit-community.com/>
<http://www.adoit-community.com/>



http://www.um.es/docencia/barzana/IAGP/Enlaces/CASE_principales.html
http://www.um.es/docencia/barzana/IAGP/Enlaces/CASE_principales.html



<http://www.gobiernodigital.gob.pe/publica/metodologias/Lib5083/INDEX.HTM>
<http://www.gobiernodigital.gob.pe/publica/metodologias/Lib5083/INDEX.HTM>



<http://wiki.uqbar.org/wiki/articles/introduccion-a-las-metodologias-de-desarrollo-de-software.html>
<http://wiki.uqbar.org/wiki/articles/introduccion-a-las-metodologias-de-desarrollo-de-software.html>



<http://roa.ult.edu.cu/bitstream/123456789/476/1/TodoAgil.pdf>
<http://roa.ult.edu.cu/bitstream/123456789/476/1/TodoAgil.pdf>

Bibliografía

- Basterrechea Molina, E. Ingeniería del software de gestión II. Scrib Sitio., 2017. [En línea] URL disponible en: <https://es.scribd.com/doc/51957036/errores-clasicos-en-el-desarrollo-de-software.>
- McConnell, S. Desarrollo y gestión de proyectos informáticos. Rapid Development. Microsoft Press. McGrawHill., 1996.:
- Sabeli, H. Lenguaje ensamblador., 2017. [En línea] URL disponible en: https://es.scribd.com/search?content_type=tops&page=1&query=reutilizaci%C3%B3n%20de%20codigo.
- Scouler, J., Bakal, M. Reutilización exitosa de código con desarrollo y modelaje basado en código. IBM., 2012. [En línea] URL disponible en: <https://www.ibm.com/developerworks/ssa/rational/library/reuse-code-centric-development-modeling/index.html.>
- Sommerville, I. Ingeniería del software. España: Pearson Educación., 2011:

Glosario.

- **ACM:** (Asociación de los Sistemas Informáticos) organización a nivel mundial que reúne a los profesionales, estudiantes, investigadores y personas interesadas en la ciencia de la computación.
- **Actividad:** proceso que tiene un tiempo y lugar, en que un agente actúa con unos objetivos determinados.
- **Actividad de requisitos:** obtención, análisis, especificaciones y validaciones de requisitos de software.
- **Algoritmia:** ciencia que nos permite evaluar el efecto que tienen diferentes factores externos sobre los algoritmos disponibles, de tal modo que sea posible seleccionar el que más se ajuste a nuestras circunstancias particulares.

- ➔ **Algoritmo de Dijkstra:** También conocido como algoritmo de caminos mínimos, es un algoritmo que busca el camino más corto entre nodos. Su aplicación está presente en todos los ámbitos de la programación, aunque destaca su aplicación en la gestión de redes.
- ➔ **Análisis:** proceso de estudiar las necesidades del usuario para obtener una definición detallada de los requisitos.
- ➔ **Ciclo de vida:** describe el desarrollo de software, desde la fase inicial hasta la final.
- ➔ **Ciencias de la computación:** aquellas que abarcan las bases teóricas de la información y la computación, así como su aplicación en sistemas computacionales.
- ➔ **Compilador:** programa informático que traduce un programa escrito en un lenguaje a otro lenguaje de programación. Usualmente el segundo lenguaje es lenguaje de máquina, pero también puede ser un código intermedio (bytecode), como es el caso de JAVA.
- ➔ **Complejidad:** algo que posee la cualidad de complejo y es medible.
- ➔ **Herramienta:** programas, aplicaciones o instrucciones usadas para efectuar una tarea.
- ➔ **Lenguaje LISP:** Lenguaje orientado a la inteligencia artificial. Es un lenguaje del tipo multiparadigma.
- ➔ **Método:** secuencia de actividades orientadas a un objetivo o meta común.
- ➔ **Metodología:** conjunto de métodos que se siguen en una investigación científica, un estudio o una exposición doctrinal.
- ➔ **Modelo:** representación de un objeto, sistema o idea. Los modelos tienen como objetivo ayudarnos a explicar, entender o mejorar un sistema.
- ➔ **Proceso:** conjunto de actividades relacionadas o que al interactuar juntas nos dan un resultado.
- ➔ **Proceso de software:** conjunto coherente de políticas, estructuras organizativas y tecnológicas que se necesitan para concebir su desarrollo implantar y mantener un producto de software.
- ➔ **RAD:** (Desarrollo Rápido de Aplicaciones). Estos lenguajes reducen el tiempo de programación mediante herramientas gráficas y la inserción de componentes. Tienen una curva menor de aprendizaje y se puede realizar un software en un tiempo menor, aunque no ofrece las opciones de optimización de otros lenguajes como JAVA.
- ➔ **Requerimiento:** Características que se desea que posea un sistema o un software.
- ➔ **Software:** soporte lógico de un sistema informático.
- ➔ **UML:** lenguaje unificado de modelado (Unified Modeling Language)., lenguaje de modelado de sistemas de software más conocido y utilizado actualmente. Es un lenguaje gráfico para construir y documentar un sistema.