

UD1.- Desarrollo de software.



Caso práctico

En BK Programación todos han vuelto ya de sus vacaciones.

Les espera un septiembre agitado, pues acaban de recibir una petición por parte de una cadena hotelera para desarrollar un proyecto software.



Ada, la supervisora de proyectos de BK Programación, se reúne con Juan y María (trabajadores de la empresa) para empezar a planificar el proyecto.

Ana, cuya especialidad es el diseño gráfico de páginas web, acaba de terminar el Ciclo de Grado Medio en Sistemas Microinformáticos y Redes y realizó la FCT en BK Programación. Trabaja en la empresa ayudando en los diseños, y aunque está contenta con su trabajo, le gustaría participar activamente en todas las fases en el proyecto. El problema es que carece de los conocimientos necesarios.

Antonio se ha enterado de la posibilidad de estudiar el nuevo Ciclo de Grado Superior de Diseño de Aplicaciones Multiplataforma a distancia, y está dispuesta a hacerlo. (No tendría que dejar el trabajo).

Le comenta sus planes a su amigo Antonio (que tiene conocimientos básicos de informática), y éste se une a ella.

Después de todo... ¿qué pueden perder?




[Ministerio de Educación y Formación Profesional.](#)
(Dominio público)

Materiales formativos de FP Online propiedad del Ministerio de Educación y Formación Profesional.


[Aviso Legal](#)

1.- Software y programa. Tipos de software.

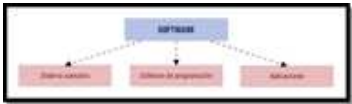


Caso práctico

Todos en la empresa están entusiasmados con el proyecto que tienen entre manos. Saben que lo más importante es planificarlo todo de antemano y elegir el tipo de software más adecuado. Ana les escucha hablar y no llega a entender por qué hablan de "tipos de software". ¿Acaso el software no era la parte lógica del ordenador, sin más? ¿Cuáles son los tipos de software?

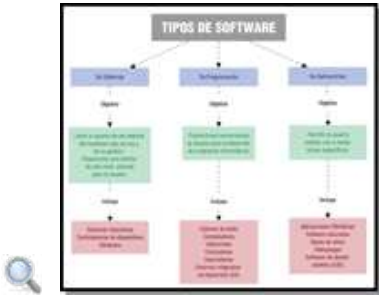


Es conocido que el ordenador se compone de dos partes bien diferenciadas: 🖨️ Hardware y 🖨️ Software.





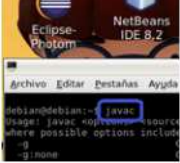

El software es el conjunto de programas informáticos que actúan sobre el hardware para ejecutar lo que el usuario desee.

Según su función se distinguen **tres tipos de software**: software de sistema , software de programación y 🖨️ aplicaciones.



Tipos de software

	El software de sistema es el software base que ha de estar instalado y configurado en nuestro ordenador para que las aplicaciones puedan ejecutarse y funcionar. El principal tipo de software de sistema es el sistema operativo. Algunos ejemplos de sistemas operativos son: Windows, Linux, Mac.
	El software de programación es el conjunto de herramientas que nos permiten desarrollar programas informáticos.

	<p>Algunos ejemplos son los editores de texto/código, compiladores, intérpretes, entornos de desarrollo integrados (IDE).</p>
	<p>Las aplicaciones informáticas son un conjunto de programas que tienen una finalidad más o menos concreta. Son ejemplos de aplicaciones los procesadores de textos, las hojas de cálculo, el software para reproducir música, los videojuegos, etc.</p>

En definitiva, un **programa** es un conjunto de instrucciones escritas en un lenguaje de programación, que indican a la máquina que operaciones realizar sobre unos determinados datos.

En este tema, nuestro interés se centra en ver como se desarrollan las aplicaciones informáticas.

A lo largo de esta primera unidad vas a aprender los conceptos fundamentales de software y las fases del llamado ciclo de vida de una aplicación informática.

También aprenderás a distinguir los diferentes lenguajes de programación y los procesos que ocurren hasta que el programa funciona y realiza la acción deseada.



Para saber más

En el siguiente enlace encontrarás más información de los tipos de software existente, así como ejemplos de cada uno que te ayudarán a profundizar sobre el tema.



[Ampliación sobre los tipos de software.](#)



Reflexiona

Hay varios sistemas operativos en el mercado: Linux, Windows, Mac OS X etc. El más conocido es Windows. A pesar de eso, ¿por qué utilizamos cada vez más Linux?

2.- Relación Hardware - Software.



Caso práctico

Después de saber ya diferenciar los distintos tipos de software, Ana se le plantea otra cuestión: El software, sea del tipo que sea, se ejecuta sobre los dispositivos físicos del ordenador. ¿Qué relación hay entre ellos?



Como sabemos, al conjunto de dispositivos físicos que conforman un ordenador se le denomina hardware.

Al hablar de un ordenador, la relación hardware-software es inseparable. El software se ejecuta sobre los dispositivos físicos y éstos precisan del software para proporcionar sus funciones.

La primera arquitectura hardware se estableció en 1946 por John Von Neumann, véase en la siguiente figura los principales bloques que la conforman.

En la actualidad, los equipos todavía se basan en esos mismos conceptos. Esta relación software-hardware la podemos poner de manifiesto desde dos puntos de vista:



[Descripción de la imagen](#)

Desde el punto de vista del sistema operativo

El sistema operativo es el encargado de coordinar al hardware durante el funcionamiento del ordenador, actuando como intermediario entre éste y las aplicaciones que están corriendo en un momento dado.

Todas las aplicaciones necesitan recursos hardware durante su ejecución (tiempo de CPU, espacio en memoria RAM, tratamiento de interrupciones, gestión de los dispositivos de Entrada/Salida, etc.). Será siempre el sistema operativo el encargado de controlar todos estos aspectos de manera "oculta" para las aplicaciones (y para el usuario).

Desde el punto de vista de las aplicaciones

Como ya sabemos, una aplicación no es otra cosa que un conjunto de programas y que éstos están escritos en algún lenguaje de programación que el hardware del equipo debe interpretar y ejecutar.



Hay multitud de lenguajes de programación diferentes (como ya veremos en su momento). Sin embargo, todos tienen algo en común: estar escritos con sentencias de un idioma que el ser humano puede aprender y usar fácilmente. Por otra parte, el hardware de un ordenador sólo es capaz de interpretar señales eléctricas (ausencias o presencias de tensión) que, en informática, se traducen en secuencias de 0 y 1 (código binario).

Esto nos hace plantearnos una cuestión: ¿Cómo será capaz el ordenador de "entender" algo escrito en un lenguaje que no es el suyo?.

Como veremos a lo largo de esta unidad, tendrá que pasar algo (un proceso de traducción de código) para que el ordenador ejecute las instrucciones escritas en un lenguaje de programación.



Autoevaluación

Para fabricar un programa informático que se ejecuta en una computadora:

- ☐ Hay que escribir las instrucciones en código binario para que las entienda el hardware.
- ☐ Sólo es necesario escribir el programa en algún lenguaje de programación y se ejecuta directamente.
- ☐ Hay que escribir el programa en algún Lenguaje de Programación y contar con herramientas software que lo traduzcan a código binario.
- ☐ Los programas informáticos no se pueden escribir: forman parte de los sistemas operativos.

Incorrecta, ya que el ser humano no tiene capacidad para escribir programas usando 1 y 0.

No es correcta porque el hardware no entiende ese lenguaje.

Muy bien. Esa es la idea...

No es cierta ninguna de las dos afirmaciones.

Solución

1. Incorrecto
2. Incorrecto
3. Opción correcta
4. Incorrecto

3.- Fases a seguir en el desarrollo del software.



Caso práctico


En BK programación ya están manos a la obra. Ada reúne a toda su plantilla para desarrollar el nuevo proyecto.

Ella sabe mejor que nadie que no será sencillo y que habrá que pasar por una serie de etapas. Ana no quiere perderse la reunión, quiere descubrir por qué hay que tomar tantas anotaciones y tantas molestias antes incluso de empezar.



En los puntos siguientes veremos como elegir un modelo de ciclo de vida para el desarrollo de nuestro software.

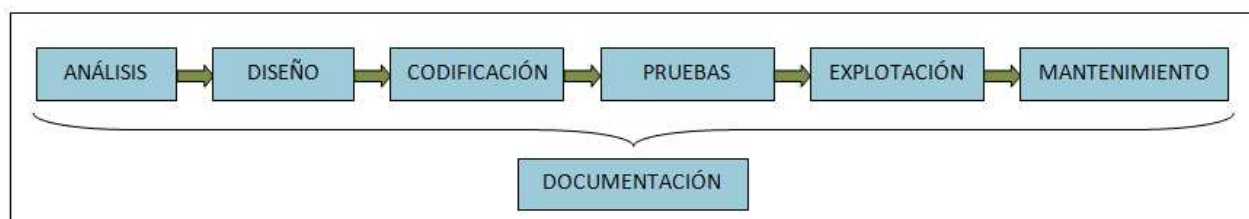
Independientemente del modelo elegido, siempre hay una serie de etapas que debemos seguir para construir software fiable y de calidad.

Entendemos por  Desarrollo de Software todo el proceso que ocurre desde que se concibe una idea hasta que un programa está implementado en el ordenador y funcionando.

El proceso de desarrollo, que en un principio puede parecer una tarea simple, consta de una serie de pasos de obligado cumplimiento, pues sólo así podremos garantizar que los programas creados son eficientes, fiables, seguros y responden a las necesidades de los usuarios finales (aquellos que van a utilizar el programa).

Es muy importante dedicar los recursos necesarios en las primeras etapas del desarrollo del software. Avanzar a las etapas finales sin un análisis y diseño libres de errores, implicará que se propaguen durante toda la vida del proyecto y como consecuencia el producto obtenido sea de mala calidad.

Estas etapas son:



Fases del desarrollo del software

Fase	Tareas
------	--------

Análisis 	<p>Analizar las necesidades de la aplicación a generar.</p> <p>Consensuar todo lo que se requiere del sistema, siendo el punto de partida para las siguientes etapas. Estos requisitos “deberían” ser cerrados para el resto del desarrollo.</p> <p>Se especifican los requisitos funcionales y no funcionales del sistema (ANÁLISIS DE REQUISITOS).</p>
Diseño 	<p>Se divide el sistema en partes y se determina la función de cada una.</p> <p>Realizar los algoritmos necesarios para el cumplimiento de los requisitos planteados en el proyecto.</p> <p>Determinar las herramientas a utilizar en la codificación.</p>
Codificación y compilación 	<p>Implementar el código fuente y obtener los ficheros en código máquina que es capaz de entender el ordenador.</p>
Pruebas 	<p>Los elementos, ya programados, se enlazan para componer el sistema y se comprueba que funciona correctamente y que cumple con los requisitos, antes de ser entregado al usuario final.</p>
Verificación en cliente / Explotación 	<p>Instalamos, configuramos y probamos la aplicación en los equipos del cliente.</p> <p>Es la fase donde el usuario final utiliza el sistema en un entorno de preproducción o pruebas. Si todo va correctamente, el producto estará listo para ser pasado a producción.</p>
Mantenimiento 	<p>Se mantiene el contacto con el cliente para actualizar y modificar la aplicación el futuro.</p> <p>Se trata de modificaciones al producto, generando nuevas versiones del mismo.</p>



Documentación



Las tareas de documentación están presentes a lo largo de todo el ciclo de vida del proyecto, por lo que muchos autores no la consideran una etapa en sí misma.

De todas las etapas, se documenta y guarda toda la información.



Reflexiona

La construcción de software es un proceso que puede llegar a ser muy complejo y que exige gran coordinación y disciplina del grupo de trabajo que lo desarrolle.



Autoevaluación

¿Crees que debemos esperar a tener completamente cerrada una etapa para pasar a la siguiente?

- ☐ Sí.
- ☐ No.

Incorrecto. Recuerda que hay que dejar siempre una "puerta abierta" para volver atrás e introducir modificaciones.

Muy bien, vas captando la idea.

Solución

1. Incorrecto

2. Opción correcta

3.1.- Análisis.




Caso práctico

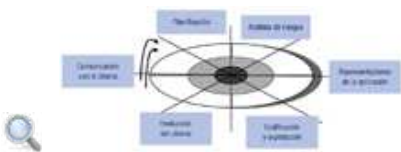
En la reunión de BK acerca del nuevo proyecto Ada, la supervisora, dejó bien claro que lo primero y más importante es tener claro qué queremos que haga el software y con qué herramientas contamos: lo demás vendría después, ya que si esto no está bien planteado, ese error se propagará a todas las fases del proyecto.



- ¿Por dónde empezamos? —pregunta Juan.
- ANÁLISIS de REQUISITOS —contesta Ada.

Esta es la primera fase del proyecto. Una vez finalizada, pasamos a la siguiente (diseño).

Es la fase de mayor importancia en el desarrollo del proyecto y todo lo demás dependerá de lo bien detallada que esté. También es la más complicada, ya que no está automatizada y depende en gran medida del  analista que la realice.



¿Qué se hace en esta fase?

En líneas generales, de esta fase obtendremos dos salidas:

- ✔ **Documento especificación de requisitos software**, que considerará tanto requisitos funcionales como no funcionales del sistema.
 1. **Funcionales**: qué funciones tendrá que realizar la aplicación. Qué respuesta dará la aplicación ante todas las entradas. Cómo se comportará la aplicación en situaciones inesperadas.
 2. **No funcionales**: tiempos de respuesta del programa, legislación aplicable, tratamiento ante la simultaneidad de peticiones, etc.

Considerando un programa para la gestión de ventas de una cadena de tiendas, podríamos considerar como ejemplo de requisitos:

Funcionales	No funcionales
Si se desea que la lectura de los productos se realice mediante códigos de barras.	Los PCs suministrados deberán ser de color azul por tratarse del color corporativo.

Si se van a detallar las facturas de compra y sus formatos.	La venta online tendrá que garantizar un servicio ininterrumpido a lo largo de año. La disponibilidad deberá ser 24x7.
Si los trabajadores de las tiendas trabajan a comisión, tener información de las ventas de cada uno.	Los materiales entregables deberán cumplir la normativa requerida por la comunidad europea en el sector del comercio.
Si se desea un control del stock en almacén.	La empresa debe hacer sus desarrollos de acuerdo a algún tipo de certificación.
La interfaz tiene que ser fácil de usar para usuarios con pocos conocimientos de informática.	

- ♥ **Documento de diseño de arquitectura**, que contiene la descripción de la estructura relacional global del sistema y la especificación de lo que debe hacer cada una de sus partes, así como la manera en que se combinan unas con otras. En ocasiones este documento se genera como una de las primeras tareas de la fase de diseño.

Es imprescindible una buena comunicación entre el analista y el cliente para que la aplicación que se va a desarrollar cumpla con sus expectativas. Y habrá que asegurar que se definen aspectos como los siguientes:

- ♥ La planificación de las reuniones que van a tener lugar.
- ♥ Relación de los objetivos del usuario cliente y del sistema.
- ♥ Relación de objetivos prioritarios y temporización.
- ♥ Mecanismos de actuación ante contingencias.
- ♥



Citas para pensar

Todo aquello que no se detecte, o resulte mal entendido en la etapa inicial provocará un fuerte impacto negativo en los requisitos, propagando esta corriente degradante a lo largo de todo el proceso de desarrollo e **incrementando su perjuicio cuanto más tardía sea su detección**
(Bell y Thayer 1976)(Davis 1993).

3.2.- Diseño.



Caso práctico

Juan está agobiado por el proyecto. Ya han mantenido comunicaciones con el cliente y saben perfectamente qué debe hacer la aplicación. También tiene una lista de las características hardware de los equipos de su cliente y todos los requisitos. Tiene tanta información que no sabe por dónde empezar.

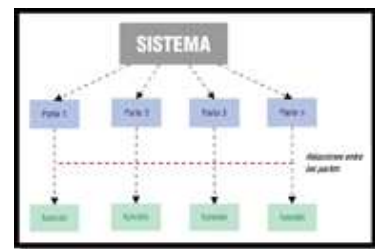


Decide hablar con Ada. Su supervisora, amable como siempre, le sugiere que empiece a dividir el problema en las partes implicadas.

—Vale, Ada, pero, ¿cómo lo divido?

En este punto, ya sabemos lo que hay que hacer, el análisis ya ha definido los requisitos y el documento de análisis arquitectónico identifica cómo dividir el programa para afrontar su desarrollo, pero ¿Cómo hacerlo?.

Se debe dividir el sistema en partes y establecer qué relaciones habrá entre ellas.



Decidir qué hará exactamente cada parte.

En definitiva, debemos crear un modelo funcional-estructural de los requerimientos del sistema global, para poder dividirlo y afrontar las partes por separado.

En este punto, se deben tomar decisiones importantes, tales como:

- ✓ Entidades y relaciones de las bases de datos.
- ✓ Selección del lenguaje de programación que se va a utilizar.
- ✓ Selección del Sistema Gestor de Base de Datos.
- ✓ Etc.

En líneas generales, de esta fase obtendremos dos salidas:

- ✓ **Documento de Diseño del Software**, que recoge información de los aspectos anteriormente mencionados.
- ✓ **Plan de pruebas** a utilizar en la fase de pruebas, sin entrar en detalles de las mismas.

Nota: en ocasiones, el diseño de arquitectura, que aquí ha sido tratado como una labor a realizar en la fase de análisis, es considerado como una primera tarea de la fase de diseño



Citas para pensar

Design is not just what it looks like and feels like. Design is how it works.

Steve Jobs ("El diseño no es sólo lo que parece y cómo parece. Diseño es cómo se trabaja").



Reflexiona

Según estimaciones, las organizaciones y empresas que crecen más son las que más dinero invierten en sus diseños.

3.3.- Codificación.



Caso práctico

En BK, ya tienen el proyecto dividido en partes.

Ahora llega una parte clave: codificar los pasos y acciones a seguir para que el ordenador los ejecute. En otras palabras, programar la aplicación. Saben que no será fácil, pero afortunadamente cuentan con herramientas CASE que les van a ser de gran ayuda. A Ana le gustaría participar, pero cuando se habla de "código fuente", "ejecutable", etc. sabe que no tiene ni idea y que no tendrá más remedio que estudiarlo si quiere colaborar en esta fase del proyecto.



Durante la fase de codificación se realiza el proceso de programación.

Consiste en elegir un determinado lenguaje de programación y una vez elegido, codificar toda la información anterior (es decir, indicar paso a paso usando un lenguaje de programación, las tareas que debe realizar el ordenador). Al realizar esto se obtiene lo que se llama **código fuente**.

Esta tarea la realiza el programador y tiene que cumplir exhaustivamente con todos los datos impuestos en el análisis y en el diseño de la aplicación.

Las características deseables de todo código son:

1. **Modularidad:** que esté dividido en trozos más pequeños.
2. **Corrección:** que haga lo que se le pide realmente.
3. **Fácil de leer:** para facilitar su desarrollo y mantenimiento futuro.
4. **Eficiencia:** que haga un buen uso de los recursos.
5. **Portabilidad:** que se pueda implementar en cualquier equipo.

```
#!/bin/bash
read b
if [ $a -eq $b ]
then
{
    echo $a es igual a $b
}
else
{
    if [ $a -lt $b ]
    then
    {
        echo $a es mayor q
```

3.4.- Compilación.

El **código fuente** es el conjunto de instrucciones que la computadora deberá realizar, escritas por los programadores en algún lenguaje de alto nivel.

Este conjunto de instrucciones no es directamente ejecutable por la máquina, sino que deberá ser traducido al lenguaje máquina, que la computadora será capaz de entender y ejecutar.

En esta fase se hace una traducción de todo el código fuente con el objetivo de pasarlo a lenguaje máquina. Esta traducción es absolutamente necesaria debido a que es el lenguaje que entiende el ordenador.

El proceso de traducción de código fuente a código objeto puede realizarse de dos formas:

1. **Compilación:** El proceso de traducción se realiza sobre todo el código fuente, en un solo paso. Se crea código objeto que habrá que enlazar. El software responsable se llama **compilador**.
2. **Interpretación:** El proceso de traducción del código fuente se realiza línea a línea y se ejecuta simultáneamente. No existe código objeto intermedio. El software responsable se llama **intérprete**. El proceso de traducción es más lento que en el caso de la compilación, pero es recomendable cuando el programador es inexperto, ya que da la detección de errores es más detallada.

3.5.- Pruebas.



Caso práctico

María reúne todos los códigos diseñados y los prepara para implementarlos en el equipo del cliente.



Juan se percata de ello, y le recuerda a su amiga que aún no los han sometido a pruebas. Juan se acuerda bien de la vez que le pasó aquello: *hace dos años, cuando fue a presentar una aplicación a sus clientes, no paraba de dar errores de todo tipo... los clientes, por supuesto, no la aceptaron y Juan perdió un mes de duro trabajo y estuvo a punto de perder su empleo...*

“—No tan deprisa María, tenemos que PROBAR la aplicación”.



Una vez obtenido el software, la siguiente fase del ciclo de vida es la realización de pruebas.

Normalmente, éstas se realizan sobre un conjunto de datos de prueba, que consisten en un conjunto seleccionado y predefinido de datos límite a los que la aplicación es sometida.

La realización de pruebas es imprescindible para asegurar la  validación y  verificación del software construido.

Entre todas las pruebas que se efectúan sobre el software podemos distinguir básicamente:

PRUEBAS UNITARIAS.


Consisten en probar, una a una, las diferentes partes de software y comprobar su funcionamiento (por separado, de manera independiente).

Como resultado de las pruebas unitarias se genera un **documento de procedimiento de pruebas**, que tendrá como partida el plan de pruebas de la fase de diseño. Éste incluye los resultados obtenidos y deben ser comparados con los resultados esperados que se habrán determinado de antemano.

JUnit es el entorno de pruebas unitarias para Java.

PRUEBAS DE INTEGRACIÓN.

Consiste en la puesta en común de todos los programas desarrollados una vez pasadas las pruebas unitarias de cada uno de ellos. Para las pruebas de integración se genera un **documento de procedimiento de pruebas de integración**, que podrá partir de un plan de pruebas de integración si durante la fase de análisis fue generado. Al igual que en las pruebas unitarias los resultados esperados se compararán con los obtenidos.

En el siguiente  [enlace](#) encontrarás información interesante sobre los tipos de prueba que debemos hacer a nuestro software.



Autoevaluación

Si las pruebas unitarias se realizan con éxito, ¿es obligatorio realizar las de integración?

- ☐ Sí, si la aplicación está formada por más de cinco módulos diferentes.
- ☐ Sí, en cualquier caso.

Incorrecto. Una aplicación formada por dos módulos ya es susceptible de producir errores en su interrelación.

Muy bien, vas captando la idea.

Solución

1. Incorrecto
2. Opción correcta

3.6.- Explotación.



Caso práctico

Llega el día de la cita con la cadena hotelera. Ada y Juan se dirigen al hotel donde se va a instalar y configurar la aplicación. Si todo va bien, se irá implementando en los demás hoteles de la cadena.

Ada no quiere que se le pase ni un detalle: lleva consigo la guía de uso y la guía de instalación.

Después de todas las fases anteriores, una vez que las pruebas nos demuestran que el software es fiable, carece de errores y hemos documentado todas las fases, el siguiente paso es la explotación.

Aunque diversos autores consideran la explotación y el mantenimiento como la misma etapa, nosotros vamos a diferenciarlas en base al momento en que se realizan.

La explotación es la fase en que los usuarios finales conocen la aplicación y comienzan a utilizarla.

La explotación es la instalación, puesta a punto y funcionamiento de la aplicación en el equipo final del cliente.

En el proceso de instalación, los programas son transferidos al computador del usuario cliente y posteriormente configurados y verificados.

Es recomendable que los futuros clientes estén presentes en este momento e irles comentando cómo se va planteando la instalación.



En este momento, se suelen llevar a cabo las Beta Test, que son las últimas pruebas que se realizan en los propios equipos del cliente y bajo cargas normales de trabajo.

Una vez instalada, pasamos a la fase de configuración.

En ella, asignamos los parámetros de funcionamiento normal de la empresa y probamos que la aplicación es operativa. También puede ocurrir que la configuración la realicen los propios usuarios finales, siempre y cuando les hayamos dado previamente la guía de instalación. Y también, si la aplicación es más sencilla, podemos programar la configuración de manera que se realice automáticamente tras instalarla. (Si el software es "a medida", lo más aconsejable es que la hagan aquellos que la han fabricado).

Una vez se ha configurado, el siguiente y último paso es la fase de producción normal. La aplicación pasa a manos de los usuarios finales y se da comienzo a la explotación del software.

Es muy importante tenerlo todo preparado antes de presentarle el producto al cliente: será el momento crítico del proyecto.



Reflexiona

Realizas un proyecto software por vez primera y no te das cuenta de documentarlo. Consigues venderlo a buen precio a una empresa. Al cabo de un par de meses te piden que actualices algunas de las funciones, para tener mayor funcionalidad. Estás contento o contenta porque eso significa un ingreso extra. Te paras un momento...¿Dónde están los códigos? ¿Qué hacía exactamente la aplicación? ¿Cómo se diseñó? No lo recuerdas... Probablemente hayas perdido un ingreso extra y unos buenos clientes.

3.7.- Mantenimiento.



Caso práctico

Ada reúne por última vez durante estas semanas a su equipo. Todos celebran que el proyecto se ha implementado con éxito y que sus clientes han quedado satisfechos.

—Esto aún no ha terminado —comenta Ada—, nos quedan muchas cosas por hacer. Esta tarde me reúno con los clientes. ¿Cómo vamos a gestionar el mantenimiento de la aplicación?



Sería lógico pensar que con la entrega de nuestra aplicación (la instalación y configuración de nuestro proyecto en los equipos del cliente) hemos terminado nuestro trabajo.

En cualquier otro sector laboral esto es así, pero el caso de la construcción de software es muy diferente.

La etapa de mantenimiento es la más larga de todo el ciclo de vida del software.

Por su naturaleza, el software es cambiante y deberá actualizarse y evolucionar con el tiempo. Deberá ir adaptándose de forma paralela a las mejoras del hardware en el mercado y afrontar situaciones nuevas que no existían cuando el software se construyó.

Además, siempre surgen errores que habrá que ir corrigiendo y nuevas versiones del producto mejores que las anteriores.

Por todo ello, se pacta con el cliente un servicio de mantenimiento de la aplicación (que también tendrá un coste temporal y económico).

El mantenimiento se define como el proceso de control, mejora y optimización del software.

Su duración es la mayor en todo el ciclo de vida del software, ya que también comprende las actualizaciones y evoluciones futuras del mismo.

Los tipos de cambios que hacen necesario el mantenimiento del software son los siguientes:

- ✓ **Perfectivos:** Para mejorar la funcionalidad del software.
- ✓ **Evolutivos:** El cliente propone mejoras para el producto. Implica nuevos requisitos.
- ✓ **Adaptativos:** Modificaciones, actualizaciones... para adaptarse a las nuevas tendencias del mercado, a nuevos componentes hardware, nuevas condiciones especificadas por organismos reguladores, etc.
- ✓ **Correctivos:** Resolver errores detectados. Sería utópico pensar que esto no vaya a suceder.



Autoevaluación

¿Cuál es, en tu opinión, la etapa más importante del desarrollo de software?

- ☐ El análisis de requisitos.
- ☐ La codificación.
- ☐ Las pruebas y documentación.
- ☐ La explotación y el mantenimiento.

Efectivamente. Si esta etapa no está lograda, las demás tampoco lo estarán.

Incorrecto, no necesariamente. Hoy día contamos con ayudas automatizadas (CASE).

No es correcto. Es importante, pero si el análisis no es correcto, no nos servirán de mucho.

No es cierto porque si el software no hace lo que se le pide, éstas no tendrán sentido.

Solución

1. Opción correcta
2. Incorrecto
3. Incorrecto
4. Incorrecto

3.8.- Documentación.



Caso práctico

Ada ha quedado dentro de dos días con su cliente. Pregunta a María por todos los dossiers de documentación. La pálida expresión de la joven hace que Ada arda en desesperación: "—¿No habéis documentado las etapas? ¿Cómo voy a explicarle al cliente y sus empleados el funcionamiento del software? ¿Cómo vamos a realizar su mantenimiento?".



Todas las etapas en el desarrollo de software deben quedar perfectamente documentadas.

En realidad, la documentación no debe ser considerada como una etapa más en el desarrollo del proyecto, la elaboración de documentos es constante durante todo su ciclo de vida.

Documentar un proyecto se hace necesario para dar toda la información a los usuarios de nuestro software y para poder acometer futuras revisiones.

Una correcta documentación permitirá pasar de una etapa a otra de forma clara y definida. También se hace imprescindible para la reutilización de parte de los programas en otras aplicaciones, siempre y cuando se desarrollen con diseño modular.

De acuerdo al destinatario final de los documentos, podemos distinguir tres tipos:

Guías técnicas. Dirigidos a personal técnico en informática (analistas y programadores).


Aspectos que quedan reflejados:	El diseño de la aplicación. La codificación de los programas. Las pruebas realizadas.
¿Cuál es su objetivo?	Facilitar un correcto desarrollo, realizar correcciones en los programas y permitir un mantenimiento futuro.

Guías de uso. Dirigidos a usuarios que van a usar la aplicación(clientes).

Aspectos que quedan reflejados:	Descripción de la funcionalidad de la aplicación. Forma de comenzar a ejecutar la aplicación. Ejemplos de uso del programa. Requerimientos software de la aplicación. Solución de los posibles problemas que se pueden
---------------------------------	--

	presentar.
¿Cuál es su objetivo?	Dar a los usuarios finales toda la información necesaria para utilizar la aplicación.

Guías de instalación. Dirigidos al personal informático responsable de la instalación.	
Aspectos que quedan reflejados:	Puesta en marcha. Explotación. Seguridad del sistema.
¿Cuál es su objetivo?	Dar toda la información necesaria para garantizar que la implantación de la aplicación se realice de forma segura, confiable y precisa.



Reflexiona

Según estimaciones, el 26% de los grandes proyectos de software fracasan, el 48% deben modificarse drásticamente y sólo el 26% tienen rotundo éxito. La principal causa del fracaso de un proyecto es la falta de una buena planificación de las etapas y mala gestión de los pasos a seguir. ¿Por qué el porcentaje de fracaso es tan grande? ¿Por qué piensas que estas causas son tan determinantes?

Mostrar retroalimentación

Porque los errores en la planificación inicial se propagarán en cascada al resto de etapas del desarrollo.

4.- Ciclos de vida del software.

Se puede definir el **ciclo de vida de un proyecto (ciclo de vida del software)** como: conjunto de fases o etapas, procesos y actividades requeridas para ofertar, desarrollar, probar, integrar, explotar y mantener un producto software.

Al principio, el desarrollo de una aplicación era un proceso individualizado, carente de planificación donde únicamente se hacía una codificación y prueba/depuración del código a desarrollar. Pero pronto se detectaron muchos inconvenientes:

- ✓ Dependencia total de la persona que programó.
- ✓ Se desconoce el progreso y la calidad del proyecto.
- ✓ Falta de flexibilidad ante cambios.
- ✓ Posible incremento del coste o incluso imposibilidad de completarlo.
- ✓ Puede no reflejar las necesidades del cliente.

De ahí surgió la necesidad de hacer desarrollos más estructurados, aportando valor añadido y calidad al producto final, apareciendo el concepto de **ciclo de vida del software**.

Algunas ventajas que se consiguen son:

- ✓ En las primeras fases, aunque no haya líneas de código, invertir en pensar el diseño es avanzar en la construcción del sistema, pues facilitará la codificación.
- ✓ Asegura un desarrollo progresivo, con controles sistemáticos, que permite detectar defectos con mucha antelación.
- ✓ El seguimiento del proceso permite controlar los plazos de entrega retrasados y los costes excesivos.
- ✓ La documentación se realiza de manera formal y estandarizada simultáneamente al desarrollo, lo que facilita la comunicación interna entre el equipo de desarrollo y la de éste con los usuarios.
- ✓ Aumenta la visibilidad y la posibilidad de control para la gestión del proyecto.
- ✓ Supone una guía para el personal de desarrollo, marcando las tareas a realizar en cada momento.
- ✓ Minimiza la necesidad de rehacer trabajo y los problemas de puesta a punto.

Diversos autores han planteado distintos modelos de ciclos de vida, pero los más conocidos y utilizados son los citados a continuación:

1. Modelo en Cascada

Es el modelo de vida clásico del software.

Se pasa de unas etapas a otras sin retorno posible, cualquier error en las fases iniciales ya no será subsanable aunque sea detectado más adelante.

Este escaso margen de error lo hace prácticamente imposible de utilizar. Sólo es aplicable en pequeños desarrollos.

Todos los requisitos son planteados para hacer un único recorrido del proyecto.

Características:

Requiere conocimiento previo y absoluto de los requerimientos del sistema.



No hay retornos en las etapas: si hay algún error durante el proceso hay que empezar desde el principio.

No permite modificaciones ni actualizaciones del software.

Es un modelo utópico.

2. Modelo en Cascada con Realimentación

Es uno de los modelos más utilizados. Proviene del modelo anterior, pero se introduce una realimentación entre etapas, de forma que podamos volver atrás en cualquier momento para corregir, modificar o depurar algún aspecto. No obstante, si se prevén muchos cambios durante el desarrollo no es el modelo más idóneo.

Es el modelo perfecto si el proyecto es rígido (pocos cambios, poco evolutivo) y los requisitos están claros.

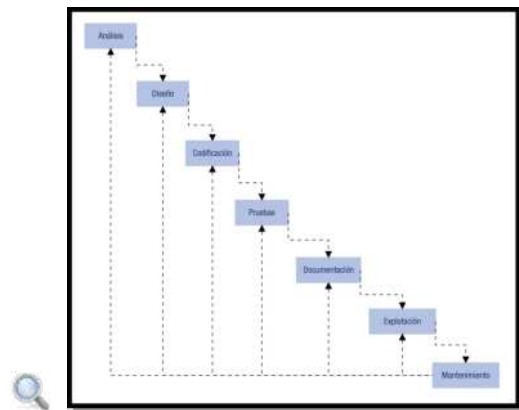
Características:

Es uno de los modelos más utilizados.

Se puede retornar a etapas anteriores para introducir modificaciones o depurar errores.

Idóneo para proyectos más o menos rígidos y con requisitos claros.

Los errores detectados una vez concluido el proyecto pueden provocar que haya que empezar desde cero.



3. Modelos Evolutivos

Son más modernos que los anteriores. Tienen en cuenta la naturaleza cambiante y evolutiva del software.

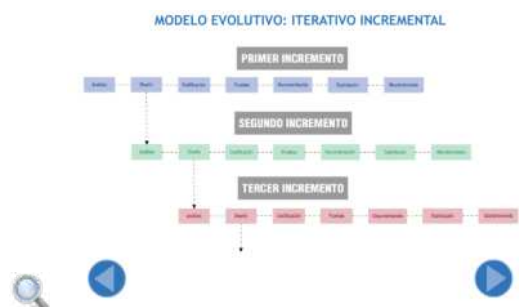
Distinguimos dos variantes:

1. Modelo Iterativo Incremental.

Está basado en el modelo en cascada con realimentación, donde las fases se repiten y refinan, y van propagando su mejora a las fases siguientes.

Cada iteración cubre una parte de los requisitos requeridos, generando versiones parciales y crecientes para el producto software en desarrollo. Cada versión obtenida será el punto de partida para la siguiente iteración.

Los incrementos a considerar en cada vuelta ya vienen establecidos desde las etapas iniciales.



Características:

Actualmente, no se ponen en el mercado los productos completos, sino versiones.

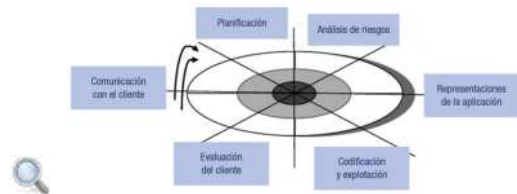
Permite una evolución temporal.

Vemos que se trata de varios ciclos en cascada que se repiten y se refinan en cada incremento.

Las sucesivas versiones del producto son cada vez más completas hasta llegar al producto final.

2. Modelo en Espiral.

Es una combinación del modelo anterior con el modelo en cascada. En él, el software se va construyendo repetidamente en forma de versiones que son cada vez mejores, debido a que incrementan la funcionalidad en cada versión. Es un modelo bastante complejo.




Características:


Se divide en 6 zonas llamadas regiones de tareas: Comunicación con el cliente, Planificación, Análisis de riesgos, Representación de la aplicación, Codificación y explotación y Evaluación del cliente.

Se adapta completamente a la naturaleza evolutiva del software.

Reduce los riesgos antes de que sean problemáticos.

3. Modelos ágiles.

Se trata de modelos que están ganando gran presencia en los desarrollos software. Muy centrados en la satisfacción del cliente, muestran gran flexibilidad a la aparición de nuevos requisitos, incluso durante el desarrollo de la aplicación. Al tratarse de metodologías evolutivas, el desarrollo es incremental, pero los incrementos son cortos y están abiertos al solapado de unas fases con otras. La comunicación entre los integrantes del equipo de trabajo y de éstos con el cliente son constantes. Un ejemplo de metodología ágil es  [Scrum](#).

 [Descargar en formato PDF](#)



Autoevaluación

Si queremos construir una aplicación pequeña, y se prevé que no sufrirá grandes cambios durante su vida, ¿sería el modelo de ciclo de vida en espiral el más recomendable?

- ☐ Sí.
- ☐ No.

Incorrecto, si la aplicación no sufrirá grandes cambios, este modelo no es recomendable.

Efectivamente, por las características de esta aplicación, pensaríamos mejor en el modelo en cascada con realimentación.

Solución

1. Incorrecto
2. Opción correcta

5.- Herramientas de apoyo al desarrollo del software.

En la práctica, para llevar a cabo varias de las etapas vistas en el punto anterior contamos con herramientas informáticas, cuya finalidad principal es automatizar las tareas y ganar fiabilidad y tiempo.



Esto nos va a permitir centrarnos en los requerimientos del sistema y el análisis del mismo, que son las causas principales de los fallos del software.

Las herramientas **CASE (Computer Aided Software Engineering)** son un conjunto de aplicaciones que se utilizan en el desarrollo de software con el objetivo de automatizar las fases del desarrollo y reducir tanto costes como tiempo del proceso. Como consecuencia, se consigue mejorar la productividad, la calidad del proceso y el resultado final.

¿En qué fases del proceso nos pueden ayudar? En el diseño del proyecto, en la codificación de nuestro diseño a partir de su apariencia visual, detección de errores...

En concreto, estas herramientas permiten:


- ✓ Mejorar la planificación del proyecto.
- ✓ Darle agilidad al proceso.
- ✓ Poder reutilizar partes del software en proyectos futuros.
- ✓ Hacer que las aplicaciones respondan a estándares.
- ✓ Mejorar la tarea del mantenimiento de los programas.
- ✓ Mejorar el proceso de desarrollo, al permitir visualizar las fases de forma gráfica.

CLASIFICACIÓN

Normalmente, las herramientas CASE se clasifican en función de las fases del ciclo de vida del software en la que ofrecen ayuda:

- ✓ **U-CASE**: ofrece ayuda en las fases de planificación y análisis de requisitos.
- ✓ **M-CASE**: ofrece ayuda en análisis y diseño.
- ✓ **L-CASE**: ayuda en la programación del software, detección de errores del código, depuración de programas y pruebas y en la generación de la documentación del proyecto.


Ejemplos de herramientas CASE libres son: ArgoUML, Use Case Maker, ObjectBuilder...

Pulsa este  [enlace](#) para ver una ampliación de los tipos y ayudas concretas de la herramientas CASE.




Para saber más

En el siguiente enlace se presenta una ampliación de los tipos y ayudas concretas de la herramientas CASE.

 [Ayudas concretas de CASE.](#)



6.- Frameworks.

Un  **Framework** es una estructura de ayuda al programador, en base a la cual podemos desarrollar proyectos sin partir desde cero.

Se trata de una plataforma software donde están definidos programas soporte, bibliotecas, lenguaje interpretado, etc., que ayuda a desarrollar y unir los diferentes módulos o partes de un proyecto.



Con el uso de framework podemos pasar más tiempo analizando los requerimientos del sistema y las especificaciones técnicas de nuestra aplicación, ya que la tarea laboriosa de los detalles de programación queda resuelta.

✓ Ventajas de utilizar un framework:

- ✦ **Desarrollo rápido** de software.
- ✦ **Reutilización** de partes de código para otras aplicaciones.
- ✦ **Diseño** uniforme del software.
- ✦ **Portabilidad** de aplicaciones de un computador a otro, ya que los bytecodes que se generan a partir del lenguaje fuente podrán ser ejecutados sobre cualquier máquina virtual.

✓ Inconvenientes:

- ✦ Gran dependencia del código respecto al framework utilizado (sin cambios de framework, habrá que reescribir gran parte de la aplicación).
- ✦ La instalación e implementación del framework en nuestro equipo consume bastantes recursos del sistema.



Para saber más

El uso creciente de frameworks hace que tengamos que estar reciclándonos constantemente. En el siguiente enlace, hay un documento muy interesante de sus principales características, ventajas y formas de uso:



[Características de frameworks.](#)



Debes conocer

Ejemplos de Frameworks:

- ✓ **.NET** es un framework para desarrollar aplicaciones sobre Windows. Ofrece el "Visual Studio .net" que nos da facilidades para construir aplicaciones y su motor es el ".Net framework" que permite ejecutar dichas aplicaciones.
- ✓ **Spring de Java**. Es un conjunto de bibliotecas (API's) para el desarrollo y

ejecución de aplicaciones Java.

🦋 **Qt.** Framework multiplataforma para el lenguaje C++. Admite adaptaciones para ser utilizado en otros lenguajes.

🦋 **Angular.** Framework de Javascript para aplicaciones web.

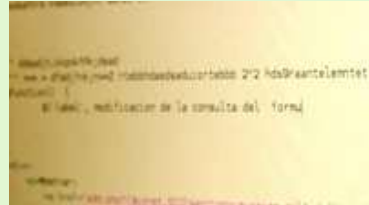
7.- Lenguajes de programación.



Caso práctico

Una de los aspectos del proyecto que más preocupa a Ada es la elección del lenguaje de programación a utilizar.

Necesita tener muy claros los requerimientos del cliente para enfocar correctamente la elección, pues según sean éstos unos lenguajes serán más efectivos que otros.



Ya dijimos anteriormente que los programas informáticos están escritos usando algún lenguaje de programación. Por tanto, podemos definir un Lenguaje de Programación como un idioma creado de forma artificial, formado por un conjunto de símbolos y normas que se aplican sobre un alfabeto para obtener un código, que el hardware de la computadora pueda entender y ejecutar. Es decir, un lenguaje de programación es el conjunto de:

- ✓ **Alfabeto:** conjunto de símbolos permitidos.
- ✓ **Sintaxis:** normas de construcción permitidas de los símbolos del lenguaje.
- ✓ **Semántica:** significado de las construcciones para hacer acciones válidas.

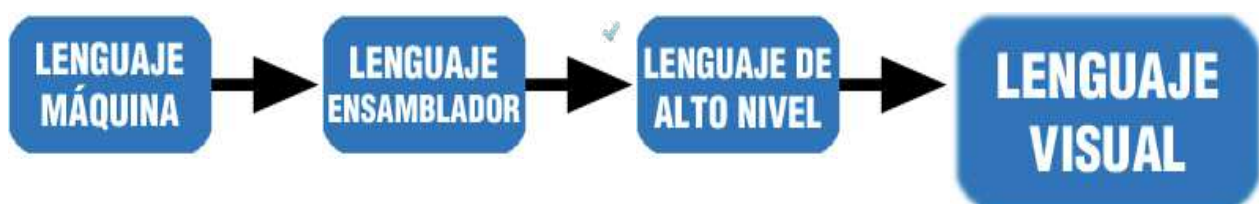
Hay multitud de lenguajes de programación, cada uno con unos símbolos y unas estructuras diferentes. Además, cada lenguaje está enfocado a la programación de tareas o áreas determinadas. Por ello, la elección del lenguaje a utilizar en un proyecto es una cuestión de extrema importancia.

Como ya sabemos, es en la etapa de diseño cuando típicamente se elige el lenguaje de programación a utilizar y en la fase de desarrollo cuando se hace uso de ellos.

Los lenguajes de programación son los que nos permiten comunicarnos con el hardware del ordenador.

En otras palabras, es muy importante tener muy clara la función de los lenguajes de programación. Son los instrumentos que tenemos para que el ordenador realice las tareas que necesitamos.

Los lenguajes de programación han sufrido su propia evolución, como se puede apreciar en la figura siguiente:





Para saber más

En el siguiente enlace, verás la evolución entre los distintos tipos de Lenguajes de Programación en la historia.



[Evolución de los Lenguajes de Programación](#)

7.1.- Características de los lenguajes de programación

Características de los Lenguajes de Programación:

✓ Lenguaje máquina:

- Sus instrucciones son combinaciones de unos y ceros.
- Es el único lenguaje que entiende directamente el ordenador. (No necesita traducción).
- Fue el primer lenguaje utilizado.
- Es único para cada procesador (no es portable de un equipo a otro).
- Hoy día nadie programa en este lenguaje.

✓ Lenguaje ensamblador:

- Sustituyó al lenguaje máquina para facilitar la labor de programación.
- En lugar de unos y ceros se programa usando mnemotécnicos (instrucciones complejas).
- Necesita traducción al lenguaje máquina para poder ejecutarse.
- Sus instrucciones son sentencias que hacen referencia a la ubicación física de los archivos en el equipo.
- Es difícil de utilizar.

✓ Lenguaje de alto nivel basados en código:

- Sustituyeron al lenguaje ensamblador para facilitar más la labor de programación.
- En lugar de mnemotécnicos, se utilizan sentencias y órdenes derivadas del idioma inglés. (Necesita traducción al lenguaje máquina).
- Son más cercanos al razonamiento humano.
- Necesita traducción al lenguaje máquina.
- Son utilizados hoy día, aunque la tendencia es que cada vez menos.

✓ Lenguajes visuales:

- Están sustituyendo a los lenguajes de alto nivel basados en código.
- En lugar de sentencias escritas, se programa gráficamente usando el ratón y diseñando directamente la apariencia del software.
- Su correspondiente código se genera automáticamente.
- Necesitan traducción al lenguaje máquina.
- Son completamente portables de un equipo a otro.

7.2.- Clasificación de los lenguajes de programación



Caso práctico

Juan y María ya han decidido el Lenguajes de Programación que van a utilizar.

Saben que el programa que realicen pasará por varias fases antes de ser implementado en los equipos del cliente. Todas esas fases van a producir transformaciones en el código. ¿Qué características irá adoptando el código a medida que avanza por el proceso de codificación?



Ya sabemos que los lenguajes de programación han evolucionado, y siguen haciéndolo, siempre hacia la mayor usabilidad de los mismos (que el mayor número posible de usuarios lo utilicen y exploten).

La elección del lenguaje de programación para codificar un programa dependerá de las características del problema a resolver.

Podemos clasificar los distintos tipos de Lenguajes de Programación en base a distintos criterios:

Según lo cerca que esté del lenguaje humano

- ✓ **Lenguajes de programación de alto nivel.** Los lenguajes de programación de alto nivel se caracterizan por traducir los algoritmos a un lenguaje mucho más fácil de entender para el programador. Estos lenguajes no están orientados a la máquina (donde se va a ejecutar) lo cual hace que los programas escritos con este tipo de lenguaje se puedan ejecutar en cualquier tipo de ordenador. Tienen el inconveniente de que este conjunto de instrucciones no es directamente ejecutable por la máquina, sino que deberá ser traducido al lenguaje máquina, que la computadora será capaz de entender y ejecutar.
- ✓ **Lenguajes de programación de bajo nivel.** Los lenguajes de bajo nivel son lenguajes que aprovechan al máximo los recursos del ordenador donde se van a ejecutar. Los lenguajes de más bajo nivel son los lenguajes máquinas. A este nivel le sigue el lenguaje ensamblador.
 - ✦ **Lenguaje máquina.** Se programan sus registros directamente con 0s y 1s. Difícil de programar y resolver errores.

```
package calculadora;  
import junit.framework.TestCase;  
/**  
 * @author usuario123  
 */  
public class CalculandoTest extends TestCase {  
    public CalculandoTest(String testName) {  
        super(testName);  
    }  
    @Override  
    protected void setUp() throws Exception {  
        super.setUp();  
    }  
}
```

- ✦ **Lenguaje ensamblador.** Se trata de un primer nivel de abstracción respecto al lenguaje máquina, aunque conceptualmente está mucho más cercano al equipo que al razonamiento humano.

Estos lenguajes tienen importantes inconvenientes, por ejemplo:

- ✦ Su adaptación a la máquina (donde se va a ejecutar) hace que no se pueda utilizar en otro ordenador al menos que tenga las mismas características que el ordenador (para el que se hizo).
- ✦ Dificultad en el manejo de dicho lenguaje.

Según su forma de ejecución

Un ordenador realizará cada una de las instrucciones indicadas en un programa. Dicho programa podrá estar escrito en muchos lenguajes de programación pero solo hay uno que entiende el ordenador: el lenguaje máquina.

Si el programa no está escrito en dicho lenguaje habrá que hacer una traducción para que el ordenador lo entienda.

Según como se realice dicha traducción, se pueden dividir en dos grupos:

- ✦ **Lenguajes compilados:** Son aquellos lenguajes que se traducen a través de un programa traductor llamado compilador. Dicho programa traduce todo el programa al lenguaje máquina, generando un nuevo fichero ejecutable, que contiene la misma información que el fichero original (código fuente) pero escrito en lenguaje máquina. Si no hay cambios en el fichero original, no hará falta volver a realizar, de nuevo, dicha traducción. Por ejemplo: C, Pascal, etc.
- ✦ **Lenguajes interpretados:** Un programa escrito en un lenguaje interpretado supone el tener que traducir el programa cada vez que se quiera ejecutar. A los programas, que realizan dicha traducción, se les llama intérpretes. Dichos programas no generan un fichero ejecutable, por ello, es necesario realizar la traducción cada una de las veces que se ejecute el programa. Por ejemplo: Basic, etc.

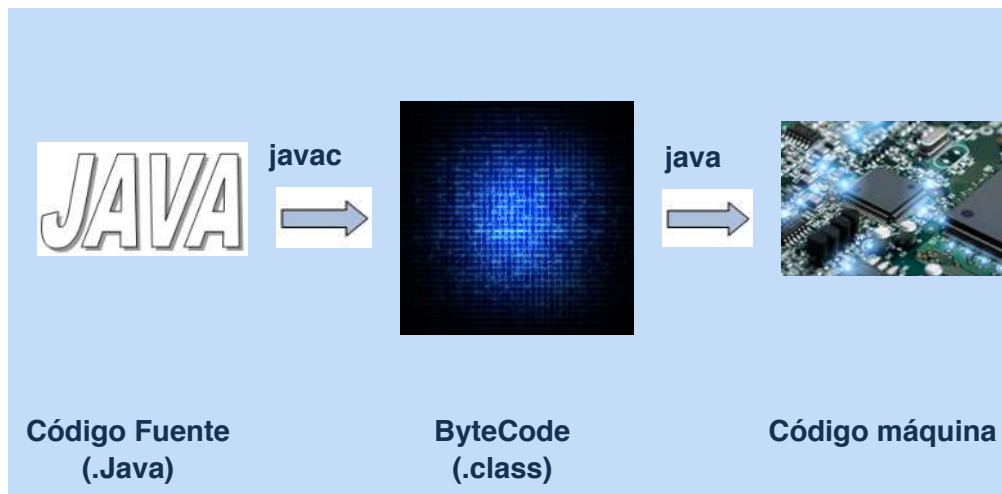
La ventaja de los lenguajes compilados frente a los interpretados es que no hace falta traducir el programa cada vez; sin embargo, su inconveniente es que no podremos ejecutar el programa hasta que no haya ningún error. Los intérpretes van traduciendo y ejecutando instrucción a instrucción lo cual hace que se pueda ejecutar el programa (aunque haya errores). En el momento que llegue a dicha instrucción errónea, se parará el programa.

Para obtener las ventajas de ambos tipos de lenguajes, algunos combinan estas dos tareas:

1. Primero, el programa original (en Java, los ficheros donde se guarda el programa fuente tienen extensión “java”) se hace una primera traducción pero no al lenguaje máquina sino a un lenguaje intermedio (en Java es bytecodes). De dicha traducción se obtiene un fichero (en Java tienen extensión “class”). Esto equivaldría a la fase de compilación.
2. En una segunda fase, dicho archivo es traducido (interpretado) en cada ejecución.

Esto es lo que realiza, por ejemplo, Java. A estos lenguajes se les llama **lenguajes intermediarios**.

Durante todo este proceso (que se realizan en los lenguajes intermedios), el código pasa por diferentes estados. La siguiente figura muestra el proceso de transformación del código fuente a máquina para el lenguaje Java.



Por lo tanto tendremos:

Tipos de código

Código Fuente



Es el escrito por los programadores en algún editor de texto. Se escribe usando algún lenguaje de programación de alto nivel y contiene el conjunto de instrucciones necesarias que debe seguir el ordenador para implementar el algoritmo.

Se trata de información conceptualmente más cercana al programador que a la máquina, que no es capaz de ejecutarlo directamente.

Un aspecto muy importante en esta fase es la elaboración previa de un algoritmo, que lo definimos como un conjunto de pasos a seguir para obtener la solución del problema. El algoritmo lo diseñamos en 📝 pseudocódigo y con él, la codificación posterior a algún Lenguaje de Programación determinado será más rápida y directa.

Para obtener el código fuente de una aplicación informática:

1. Se debe partir de las etapas anteriores de análisis y diseño.
2. Se diseñará un 📝 algoritmo que simbolice los pasos a seguir para la resolución del problema.
3. Se elegirá una Lenguajes de Programación de alto nivel apropiado para las características del software que se quiere codificar.
4. Se procederá a la codificación del algoritmo antes diseñado.

La culminación de la obtención de código fuente es un documento con la codificación de todos los 📁 módulos, 📄 funciones, bibliotecas y 📋 procedimientos necesarios para codificar la aplicación.

Puesto que, como hemos dicho antes, este código no es inteligible por la máquina, habrá que TRADUCIRLO, obteniendo así un código equivalente pero ya traducido a código binario que se llama código objeto. Que no será directamente ejecutable por la computadora si éste ha sido compilado.

Un aspecto importante a tener en cuenta es su licencia. Así, en base a ella, podemos distinguir dos tipos de código fuente:

- 🔓 Código fuente abierto. Es aquel que está disponible para que cualquier usuario pueda estudiarlo, modificarlo o reutilizarlo.
- 🔒 Código fuente cerrado. Es aquel que no tenemos permiso para editarlo.

Código Objeto



Se trata de un código intermedio para los lenguajes intermedios, por lo tanto no es directamente ejecutable por el equipo. Para que la máquina pueda ejecutar el programa es necesario transformarlo al lenguaje que ésta maneja con un intérprete. El código objeto se obtiene mediante el uso de un compilador.

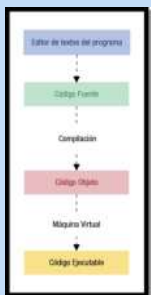
El código objeto no es directamente inteligible por el ser humano, pero tampoco por la computadora. Es un código intermedio entre el código fuente y el ejecutable.

En definitiva, es el resultado de traducir código fuente a un código equivalente formado por unos y ceros que aún no puede ser ejecutado directamente por la computadora.

Consiste en un bytecode (código binario) que está distribuido en varios archivos, cada uno de los cuales corresponde a cada programa fuente compilado.

Sólo se genera código objeto una vez que el código fuente está libre de errores sintácticos y semánticos.

Código Ejecutable



Es el código máquina resultante de enlazar los archivos de código objeto con ciertas rutinas y bibliotecas necesarias. El sistema operativo será el encargado de cargar el código ejecutable en memoria RAM y de proceder a su ejecución. Los programas compilados no producen código objeto. El proceso de compilación ya realiza el paso de fuente a ejecutable directamente.

Para obtener un sólo archivo ejecutable, habrá que enlazar todos los archivos de código objeto, a través de un software llamado linker (enlazador) y obtener así un único archivo que ya sí es ejecutable directamente por la computadora. También es conocido como código máquina y ya sí es directamente inteligible por la computadora.

Generación de código ejecutable.

En el esquema de generación de código ejecutable, vemos el proceso completo para la generación de ejecutables. A partir de un editor, escribimos el lenguaje fuente con algún Lenguaje de programación. (En el ejemplo, se usa Java). A continuación, el código fuente se compila obteniendo código objeto o bytecode. Ese bytecode, a través de la máquina virtual (se verá en el siguiente punto), pasa a código máquina, ya directamente ejecutable por la computadora.

Lenguajes imperativos vs declarativos

- ✓ En la **programación imperativa** se describe paso a paso un conjunto de instrucciones que deben ejecutarse para variar el estado del programa y hallar la solución, es decir, un algoritmo en el que se describen los pasos necesarios para solucionar el problema. P.e. **Basic, C, C++, Fortran, Pascal, Perl, PHP, Java**, lenguajes ensamblador.
- ✓ En la **programación declarativa** las sentencias que se utilizan describen el problema que se quiere solucionar, pero no las instrucciones necesarias para hacerlo. Algunos ejemplos son:
 - ✦ Lógicos: **prolog**.
 - ✦ Algebraicos: **SQL**.
 - ✦ Funcionales: **Haskell**.

Según la técnica de programación utilizada

- ✓ **Lenguajes de programación estructurados.** Usan la técnica de programación estructurada. Ejemplos: **Pascal, C**, etc.
- ✓ **Lenguajes de programación orientados a objetos (POO).** Usan la técnica de programación orientada a objetos. Ejemplos: **C++** (C plus plus), **Java, Ada, C#, .Net**, etc.

- ✔ **Lenguajes de programación visuales.** Basados en las técnicas anteriores, permiten programar gráficamente, posteriormente se obtiene un código equivalente de forma automática. Ejemplos: **Visual Basic.Net, C#, .Net**, etc.

A pesar de la inmensa cantidad de lenguajes de programación existentes, Java, C, C++, PHP y Visual Basic concentran alrededor del 60% del interés de la comunidad informática mundial.



Autoevaluación

Para obtener código fuente a partir de toda la información necesaria del problema:

- ❌ Se elige el Lenguaje de Programación más adecuado y se codifica directamente.
- ❌ Se codifica y después se elige el Lenguaje de Programación más adecuado.
- ❌ Se elige el Lenguaje de Programación más adecuado, se diseña un algoritmo y se codifica.

Incorrecta. La codificación directa nos llevará mucho tiempo y tendremos demasiados errores.

No es correcta. Antes de programar tenemos que saber qué Lenguajes de Programación vamos a utilizar.

Muy bien. El diseño del algoritmo (los pasos a seguir) nos ayudará a que la codificación posterior se realice más rápidamente y tenga menos errores.

Solución

1. Incorrecto
2. Incorrecto
3. Opción correcta



Autoevaluación

Relaciona los tipos de código con su característica más relevante, escribiendo el número asociado a la característica en el hueco correspondiente.

Ejercicio de relacionar

Tipo de código.	Relación.	Características.
Código Fuente	<input type="text"/>	1. Escrito en Lenguaje Máquina pero no ejecutable.
Código Objeto	<input type="text"/>	2. Escrito en algún Lenguaje de Programación de alto nivel, pero no ejecutable.
Código Ejecutable	<input type="text"/>	3. Escrito en Lenguaje Máquina y directamente ejecutable.

Enviar

El código fuente escrito en algún lenguaje de programación de alto nivel, el objeto escrito en lenguaje máquina sin ser ejecutable y el código ejecutable, escrito también en lenguaje máquina y ya sí ejecutable por el ordenador, son las distintas fases por donde pasan nuestros programas.

7.2.1.- Lenguajes de programación estructurados.

Aunque los requerimientos actuales de software son bastante más complejos de lo que la técnica de programación estructurada es capaz, es necesario por lo menos conocer las bases de los Lenguajes de Programación estructurados, ya que a partir de ellos se evolucionó hasta otros lenguajes y técnicas más completas (orientada a eventos u objetos) que son las que se usan actualmente.



La **programación estructurada** se define como una técnica para escribir lenguajes de programación que permite sólo el uso de tres tipos de sentencias o estructuras de control:

- ✔ **Sentencias secuenciales.**
- ✔ **Sentencias selectivas (condicionales).**
- ✔ **Sentencias repetitivas (iteraciones o bucles).**

Los lenguajes de programación que se basan en la programación estructurada reciben el nombre de lenguajes de programación estructurados.

La programación estructurada fue de gran éxito por su sencillez a la hora de construir y leer programas.

La Programación estructurada evolucionó hacia la Programación modular, que divide el programa en trozos de código llamados módulos (siguiendo la conocida técnica "divide y vencerás") con una funcionalidad concreta, que podrán ser reutilizables. A su vez, luego triunfaron los lenguajes orientados a objetos y de ahí a la programación visual (siempre es más sencillo programar gráficamente que en código, ¿no crees?).

Ventajas e inconvenientes de la programación estructurada

VENTAJAS	INCONVENIENTES
<ul style="list-style-type: none"> ✔ Los programas son fáciles de leer, sencillos y rápidos. ✔ El mantenimiento de los programas es sencillo. ✔ La estructura del programa es sencilla y clara. 	<ul style="list-style-type: none"> ✔ Todo el programa se concentra en un único bloque (si se hace demasiado grande es difícil manejarlo). ✔ No permite reutilización eficaz de código, ya que todo va "en uno". Es por esto que a la programación estructurada le sustituyó la programación modular, donde los programas se codifican por módulos y bloques, permitiendo mayor funcionalidad.



Debes conocer

La Programación estructurada evolucionó hacia la Programación modular, que divide el programa en trozos de código llamados módulos con una funcionalidad concreta, que podrán ser reutilizables.

7.2.2.- Lenguajes de programación orientados a objetos.

Después de comprender que la programación estructurada no es útil cuando los programas se hacen muy largos, es necesaria otra técnica de programación que solucione este inconveniente. Nace así la Programación Orientada a Objetos (en adelante, P.O.O.).

Los lenguajes de programación orientados a objetos tratan a los programas no como un conjunto ordenado de instrucciones (tal como sucedía en la programación estructurada) sino como un conjunto de objetos que colaboran entre ellos para realizar acciones.

En la P.O.O. los programas se componen de objetos independientes entre sí que colaboran para realizar acciones.

Su primera desventaja es clara: no es una programación tan intuitiva como la estructurada.

Algunas de sus características principales son:

- ✓ Se define clase como una colección de objetos con características similares. Si en nuestro programa interactúan un conjunto de personas, la clase a implementar será Persona y los objetos serán personas en particular (Noa, Valeria, Mila ...).
- ✓ Las clases definen una serie de atributos, que caracterizan a cada objeto persona. Por ejemplo, el atributo edad tendrá el valor 4 para Noa, 7 para Valeria
- ✓ Las clases definen una serie de métodos que corresponden a las acciones que pueden llevar a cabo los objetos. Por ejemplo, en la clase Persona se puede definir el método Saludar, que puede ser utilizado por Mila para dar los buenos días. Estos métodos contendrán el código que da respuesta a las acciones que implementan.
- ✓ Mediante llamadas a los métodos, unos objetos se comunican con otros produciéndose un cambio de estado de los mismos. Esto se denomina envío de mensajes.
- ✓ El código es más reutilizable.
- ✓ Si hay algún error, es más fácil de localizar y depurar en una clase que en un programa entero.



Algunos términos relativos a la programación orientada a objetos son:

Términos relativos a la programación orientada a objetos



- ✓ Clase
- ✓ Objeto
- ✓ Mensaje
- ✓ Método
- ✓ Evento (sistema)
- ✓ Propiedad o atributo
- ✓ Estado interno
- ✓ Abstracción

- ✓ Encapsulamiento
- ✓ Poliformismo
- ✓ Herencia


Algunos lenguajes orientados a objetos son: **Ada**, **C++** (C plus plus), **VB.NET**, **C#**, **Java**, **PowerBuilder**, etc.

Nota: otros módulos del ciclo te permitirán conocer la programación orientada a objetos en profundidad.

8.- Máquinas virtuales.

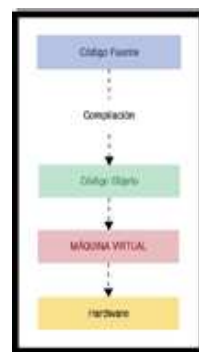
Una **máquina virtual** es un tipo especial de software cuya misión es separar el funcionamiento del ordenador de los componentes hardware instalados.

Esta capa de software desempeña un papel muy importante en el funcionamiento de los lenguajes de programación, tanto compilados como interpretados.

Con el uso de máquinas virtuales podremos desarrollar y ejecutar una aplicación sobre cualquier equipo, independientemente de las características concretas de los componentes físicos instalados. Esto garantiza la  portabilidad de las aplicaciones.

Las funciones principales de una máquina virtual son las siguientes:

- ✓ Conseguir que las aplicaciones sean portables.
- ✓ Reservar memoria para los objetos que se crean y liberar la memoria no utilizada.
- ✓ Comunicarse con el sistema donde se instala la aplicación (huésped), para el control de los dispositivos hardware implicados en los procesos.
- ✓ Cumplimiento de las normas de seguridad de las aplicaciones.



CARACTERÍSTICAS DE LA MÁQUINA VIRTUAL

- ✓ Cuando el código fuente se compila se obtiene código objeto (bytecode, código intermedio).
- ✓ Para ejecutarlo en cualquier máquina se requiere tener independencia respecto al hardware concreto que se vaya a utilizar.
- ✓ Para ello, la máquina virtual aísla la aplicación de los detalles físicos del equipo en cuestión.
- ✓ Funciona como una capa de software de bajo nivel y actúa como puente entre el bytecode de la aplicación y los dispositivos físicos del sistema.
- ✓ La Máquina Virtual verifica todo el bytecode antes de ejecutarlo.
- ✓ La Máquina Virtual protege direcciones de memoria.

La máquina virtual actúa de puente entre la aplicación y el hardware concreto del equipo donde se instale.

8.1.- Entornos de ejecución.

Un **entorno de ejecución** es un servicio de máquina virtual que sirve como base software para la ejecución de programas. En ocasiones pertenece al propio sistema operativo, pero también se puede instalar como software independiente que funcionará por debajo de la aplicación.

Es decir, es un conjunto de utilidades que permiten la ejecución de programas.

Se denomina runtime al tiempo que tarda un programa en ejecutarse en la computadora.

Durante la ejecución, los entornos se encargarán de:

- ✓ Configurar la memoria principal disponible en el sistema.
- ✓ Enlazar los archivos del programa con las bibliotecas existentes y con los subprogramas creados. Considerando que las bibliotecas son el conjunto de subprogramas que sirven para desarrollar o comunicar componentes software pero que ya existen previamente y los subprogramas serán aquellos que hemos creado a propósito para el programa.
- ✓ Depurar los programas: comprobar la existencia (o no existencia) de errores semánticos del lenguaje (los sintácticos ya se detectaron en la compilación).



Funcionamiento del entorno de ejecución:

El Entorno de Ejecución está formado por la máquina virtual y los API's (bibliotecas de clases estándar, necesarias para que la aplicación, escrita en algún Lenguaje de Programación pueda ser ejecutada). Estos dos componentes se suelen distribuir conjuntamente, porque necesitan ser compatibles entre sí.

El entorno funciona como intermediario entre el lenguaje fuente y el sistema operativo, y consigue ejecutar aplicaciones.

Sin embargo, si lo que queremos es desarrollar nuevas aplicaciones, no es suficiente con el entorno de ejecución.

Adelantándonos a lo que veremos en la próxima unidad, para desarrollar aplicaciones necesitamos algo más. Ese "algo más" se llama entorno de desarrollo.



Autoevaluación

Señala la afirmación falsa respecto de los entornos de ejecución:

- ☐ Su principal utilidad es la de permitir el desarrollo rápido de aplicaciones.
- ☐ Actúa como mediador entre el sistema operativo y el código fuente.



Es el conjunto de la máquina virtual y bibliotecas necesarias para la ejecución.

Muy bien, lo has entendido perfectamente.

Incorrecto, eso es verdad.

No es correcto, eso es verdad.

Solución

1. Opción correcta
2. Incorrecto
3. Incorrecto

8.2.- Java runtime environment.

En esta sección se va a explicar el funcionamiento, instalación, configuración y primeros pasos del Runtime Environment del lenguaje Java (se hace extensible a los demás lenguajes de programación).

Concepto.

Se denomina JRE al Java Runtime Environment (entorno en tiempo de ejecución Java).

El JRE se compone de un conjunto de utilidades que permitirá la ejecución de programas Java sobre cualquier tipo de plataforma.

Componentes.

JRE está formado por:

- ✓ Una Máquina virtual Java (JVM o JVM si consideramos las siglas en inglés), que es el programa que interpreta el código de la aplicación escrito en Java.
- ✓ Bibliotecas de clase estándar que implementan el API de Java.
- ✓ Las dos: JVM y API de Java son consistentes entre sí, por ello son distribuidas conjuntamente.

Lo primero es descargarnos el programa JRE. (Java2 Runtime Environment JRE 1.6.0.21). Java es software libre, por lo que podemos descargarnos la aplicación libremente.

Una vez descargado, comienza el proceso de instalación, siguiendo los pasos del asistente.



Debes conocer

El proceso de descarga, instalación y configuración del entorno de ejecución de programas. En el siguiente enlace, se explican los pasos para hacerlo bajo el sistema operativo Linux.



[Instalación y configuración del JRE de Java.](#)

Anexo I.- Sentencias de control de la programación estructurada.

SENTENCIAS SECUENCIALES

Las sentencias secuenciales son aquellas que se ejecutan una detrás de la otra, según el orden en que hayan sido escritas.

Ejemplo en lenguaje C:

```
printf ("declaración de variables");  
int numero_entero;  
espacio=espacio_inicio + veloc*tiempo;
```

SENTENCIAS SELECTIVAS (CONDICIONALES)

Son aquellas en las que se evalúa una condición. Si el resultado de la condición es verdad es ejecutan una serie de acción o acciones y si es falso se ejecutan otras.

if → señala la condición que se va a evaluar

then → Todas las acciones que se encuentren tras esta palabra reservada se ejecutarán si la condición del **if** es cierta (en C, se omite esta palabra).

else → Todas las acciones que se encuentren tras esta otra palabra reservada se ejecutarán si la condición de **if** es falsa.

Ejemplo en lenguaje C:

```
if (a >= b)  
c= a-b;  
else  
c=a+b;
```

SENTENCIAS REPETITIVAS (ITERACIONES O BUCLES)

Un bucle iterativo de una serie de acciones harán que éstas se repitan mientras o hasta que una determinada condición sea falsa (o verdadera).

while → marca el comienzo del bucle y va seguido de la condición de parada del mismo.

do → a partir de esta palabra reservada, se encontrarán todas las acciones a ejecutar mientras se ejecute el bucle (en C, se omite esta palabra).

done → marca el fin de las acciones que se van a repetir mientras estemos dentro del bucle (en C, se omite esta palabra).









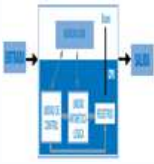


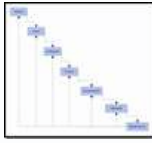


Ejemplo en lenguaje C:










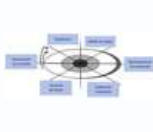

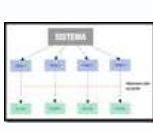

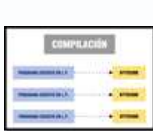


```
int num;
```












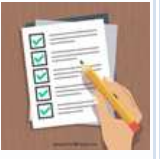



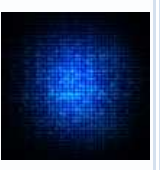
```
num = 0;
while (num<=10) { printf("Repetición numero %d\n", num);
num = num + 1;
};
```










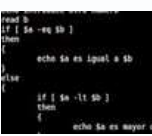


Anexo II.- Licencias de recursos.

Licencias de recursos utilizados en la Unidad de Trabajo.

Recurso (1)	Datos del recurso (1)	Recurso (2)	Datos del recurso (2)
	Autoría: Scott Schram. Licencia: CC by 2.0. Procedencia: http://www.flickr.com/photos/schram/21742249/		Autoría: fsse8info. Licencia: CC by -SA 2.0. Procedencia: http://www.flickr.com/photos/fsse-info/3276664015/
	Autoría: Ministerio de Educación. Licencia: Uso Educativo-nc. Procedencia: Elaboración propia.		Autoría: Ministerio de Educación. Licencia: Uso Educativo-nc. Procedencia: Elaboración propia.
	Autoría: Ministerio de Educación. Licencia: Uso Educativo-nc. Procedencia: Elaboración propia.		Autoría: Ministerio de Educación. Licencia: Uso Educativo-nc. Procedencia: Elaboración propia.
	Autoría: Ministerio de Educación. Licencia: Uso Educativo-nc. Procedencia: Elaboración propia.		Autoría: Ministerio de Educación. Licencia: Uso Educativo-nc. Procedencia: Elaboración propia.
	Autoría: Verónica Cabrerizo. Licencia: Uso Educativo-nc. Procedencia: Elaboración propia.		Autoría: Ministerio de Educación. Licencia: Uso Educativo-nc. Procedencia: Elaboración propia.
	Autoría: Ministerio de Educación. Licencia: Uso Educativo-nc. Procedencia: Elaboración propia.		Autoría: Ministerio de Educación. Licencia: Uso Educativo-nc. Procedencia: Elaboración propia.
	Autoría: Ministerio de Educación. Licencia: Uso Educativo-nc. Procedencia: Elaboración propia.		Autoría: Ministerio de Educación. Licencia: Uso Educativo-nc. Procedencia: Elaboración propia.

	Autoría: Ministerio de Educación. Licencia: Uso Educativo-nc. Procedencia: Elaboración propia.		Autoría: Ministerio de Educación. Licencia: Uso Educativo-nc. Procedencia: Elaboración propia.
	Autoría: Ministerio de Educación. Licencia: Uso Educativo-nc. Procedencia: Elaboración propia.		Autoría: Ministerio de Educación. Licencia: Uso Educativo-nc. Procedencia: Elaboración propia.
	Autoría: Ministerio de Educación. Licencia: Uso Educativo-nc. Procedencia: Elaboración propia.		Autoría: Ministerio de Educación. Licencia: Uso Educativo-nc. Procedencia: Elaboración propia.
	Autoría: Verónica Cabrerizo. Licencia: Uso Educativo-nc. Procedencia: Elaboración propia.		Autoría: Ministerio de Educación. Licencia: Uso Educativo-nc. Procedencia: Elaboración propia.
	Autoría: Ministerio de Educación. Licencia: Uso Educativo-nc. Procedencia: Elaboración propia.		Autoría: Ministerio de Educación. Licencia: Uso Educativo-nc. Procedencia: Elaboración propia.
	Autoría: Ministerio de Educación. Licencia: Uso Educativo-nc. Procedencia: Elaboración propia.		Autoría: Ministerio de Educación. Licencia: Uso Educativo-nc. Procedencia: Elaboración propia.
	Autoría: Ministerio de Educación. Licencia: Uso Educativo-nc. Procedencia: Elaboración propia.		Autoría: Ministerio de Educación. Licencia: Uso Educativo-nc. Procedencia: Elaboración propia.
	Autoría: Ministerio de Educación. Licencia: Uso Educativo-nc. Procedencia: Elaboración propia.		Autoría: Ministerio de Educación. Licencia: Uso Educativo-nc. Procedencia: Elaboración propia.

	Autoría: Ministerio de Educación. Licencia: Uso Educativo-nc. Procedencia: Elaboración propia.		Autoría: Ministerio de Educación. Licencia: Uso Educativo-nc. Procedencia: Elaboración propia.
	Autoría: Ministerio de Educación. Licencia: Uso Educativo-nc. Procedencia: Elaboración propia.		Autoría: Ministerio de Educación. Licencia: Uso Educativo-nc. Procedencia: Elaboración propia.
	Autoría: Ministerio de Educación. Licencia: Uso Educativo-nc. Procedencia: Elaboración propia.		Autoría: Ministerio de Educación. Licencia: Uso Educativo-nc. Procedencia: Elaboración propia.
	Autoría: Ministerio de Educación. Licencia: Uso Educativo-nc. Procedencia: Elaboración propia.		Autoría: Ministerio de Educación. Licencia: Uso Educativo-nc. Procedencia: Elaboración propia.
	Autoría: Ministerio de Educación. Licencia: Uso Educativo-nc. Procedencia: Elaboración propia.		Autoría: Ministerio de Educación. Licencia: Uso Educativo-nc. Procedencia: Elaboración propia.
	Autoría: Ministerio de Educación. Licencia: Uso Educativo-nc. Procedencia: Elaboración propia.		Autoría: Ministerio de Educación. Licencia: Uso Educativo-nc. Procedencia: Elaboración propia.
	Autoría: Ministerio de Educación. Licencia: Uso Educativo-nc. Procedencia: Elaboración propia.		Autoría: Ministerio de Educación. Licencia: Uso Educativo-nc. Procedencia: Elaboración propia.
	Autoría: Ministerio de Educación. Licencia: Uso Educativo-nc. Procedencia: Elaboración propia.		Autoría: Ministerio de Educación. Licencia: Uso Educativo-nc. Procedencia: Elaboración propia.

	<p>Autoría: Ministerio de Educación.</p> <p>Licencia: Uso Educativo-nc.</p> <p>Procedencia: Elaboración propia.</p>		<p>Autoría: Ministerio de Educación.</p> <p>Licencia: Uso Educativo-nc.</p> <p>Procedencia: Elaboración propia.</p>
	<p>Autoría: Ministerio de Educación.</p> <p>Licencia: Uso Educativo-nc.</p> <p>Procedencia: Elaboración propia.</p>		<p>Autoría: Ministerio de Educación.</p> <p>Licencia: Uso Educativo-nc.</p> <p>Procedencia: Elaboración propia.</p>
	<p>Autoría: Ministerio de Educación.</p> <p>Licencia: Uso Educativo-nc.</p> <p>Procedencia: Elaboración propia.</p>		<p>Autoría: Ministerio de Educación.</p> <p>Licencia: Uso Educativo-nc.</p> <p>Procedencia: Elaboración propia.</p>
	<p>Autoría: Ministerio de Educación.</p> <p>Licencia: Uso Educativo-nc.</p> <p>Procedencia: Elaboración propia.</p>		<p>Autoría: Ministerio de Educación.</p> <p>Licencia: Uso Educativo-nc.</p> <p>Procedencia: Elaboración propia.</p>
	<p>Autoría: Francisco Palacios.</p> <p>Licencia: CC by -NC-ND 2.0.</p> <p>Procedencia: http://www.flickr.com/photos/wizard_/3303810302/</p>		<p>Autoría: Verónica Cabrerizo.</p> <p>Licencia: Uso Educativo-nc.</p> <p>Procedencia: Elaboración propia.</p>
	<p>Autoría: Ministerio de Educación.</p> <p>Licencia: Uso Educativo-nc.</p> <p>Procedencia: Elaboración propia.</p>		<p>Autoría: Ministerio de Educación.</p> <p>Licencia: Uso Educativo-nc.</p> <p>Procedencia: Elaboración propia.</p>