MÓDULO: Entornos de Desarrollo

Unidad 1

Desarrollo de Software

Unión Europea

Fondo Social Europeo

"El FSE invierte en tu futuro"

Índice de contenido

1 Software, programa y tipos de software

Un ordenador se compone de dos partes bien diferenciadas: **Hardware y Software**.

- **El hardware** hace referencia a todos los componentes físicos de un ordenador, lo que podemos tocar, como por ejemplo el monitor, el teclado, una memoria usb, ...
- **El software** es el conjunto de programas informáticos que actúan sobre el hardware para ejecutar lo que el usuario desee. Por tanto el software es el equipamiento lógico necesario para que el ordenador realice tareas específicas.

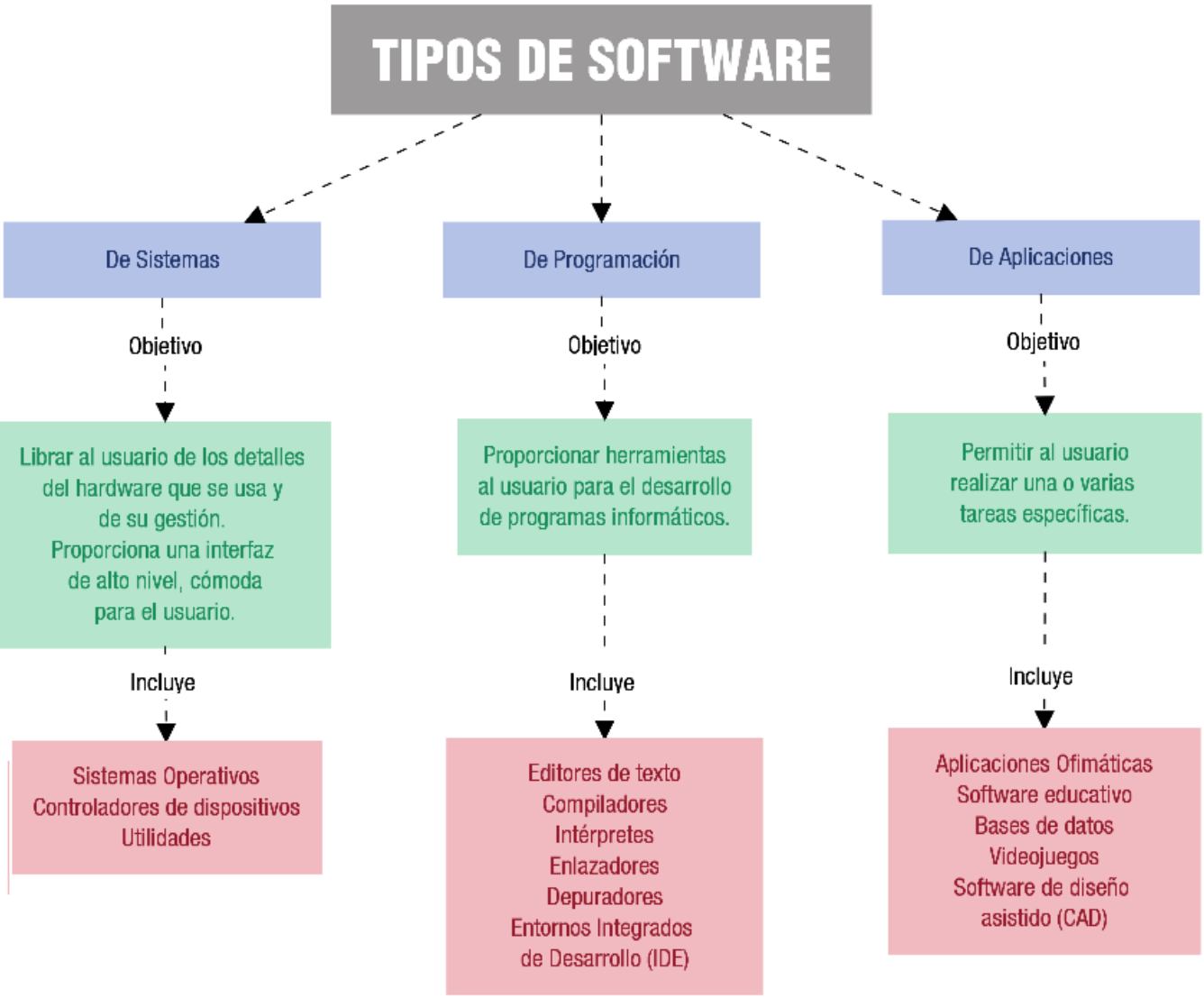
Pensemos en un CD que contiene música, el CD, ese trozo de plástico que podemos tener entre las manos, se puede ver y tocar, es el hardware, mientras que la música en el contenida, equivaldría al software, no se puede tocar pero está ahí y la podemos escuchar si ponemos el CD en un reproductor; la música equivale al software.

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

- **El sistema operativo** es el software base, ha de estar instalado y configurado en nuestro ordenador para que el resto de aplicaciones puedan ejecutarse.
  Facilita la interacción entre los componentes físicos (hardware) y el resto de las aplicaciones, librando a estas de la tarea de conocer con precisión el hardware instalado. También proporciona una interfaz con el usuario para que este pueda realizar operaciones con ficheros y dispositivos.
  Ejemplos de sistemas operativos: Windows, Linux, Mac OS X, FreeBSD, Android…..

- **El software de programación** es el conjunto de herramientas que permite desarrollar programas. Son los programas que permiten crear otros programas. Un programa es un conjunto de instrucciones escritas en un lenguaje de programación, que una vez ejecutadas realizan unas tareas en un ordenador.
  Ejemplos de software de programación: un compilador, un editor de textos, …

- **Las aplicaciones** informáticas son un conjunto de programas que tienen una finalidad en distintos campos y están dirigidas a los usuarios en general.
  Son ejemplos de aplicaciones: un procesador de textos, una hoja de cálculo, el software para reproducir música, un videojuego, un programa de retoque de fotografías, etc.



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.

El Software (wikipedia).

2 Relación hardware – software

Al conjunto de dispositivos físicos que conforman un ordenador se le denomina hardware. Pero un hardware sin software no es nada, el ordenador sería una máquina totalmente inútil. El software necesita ser instalado en el hardware para que juntos un ordenador pueda realizar una o infinitas funciones. Es el software el que transforma un ordenador un una máquina universal capaz de hacer casi cualquier cosa siempre que se le añadan los dispositivos adecuados y los programas de control de los mismos.

Por tanto, el software se ejecutará sobre un hardware. **El software que interactúa directamente con el hardware es el llamado sistema operativo (SO)**. El resto de aplicaciones acceden al hardware por medio de el. De esta forma los programadores de aplicaciones de usuario se libran de tener que preocuparse del hardware concreto de cada equipo. Si no hubiese SO, un programa procesador de textos tendría que saber imprimir en una impresora de cierta marca, si cambiamos de impresora, al programa procesador de textos le habría que añadir otra funcionalidad más para esa nueva impresora, y esto se pasaría con cada programa. Sería una locura que cada programa supiese manejar cualquier dispositivo, por eso es tan esencial un SO, ya que libra al resto de programas la interacción directa con el hardware. Si un procesador de textos quiere imprimir, simplemente le envía esa orden al SO y este se encarga de manejar la impresora.

Existe otro tipo de software base conocido como **firmware** que también actúa directamente con el hardware, pero que es más propio de otros dispositivos electrónicos más sencillos que un ordenador, como por ejemplo un router, una grabadora de discos, la BIOS de un ordenador, etc. En la actualidad, cualquier aparato un poco sofisticado lleva un firmware que permite programarlo de alguna manera.

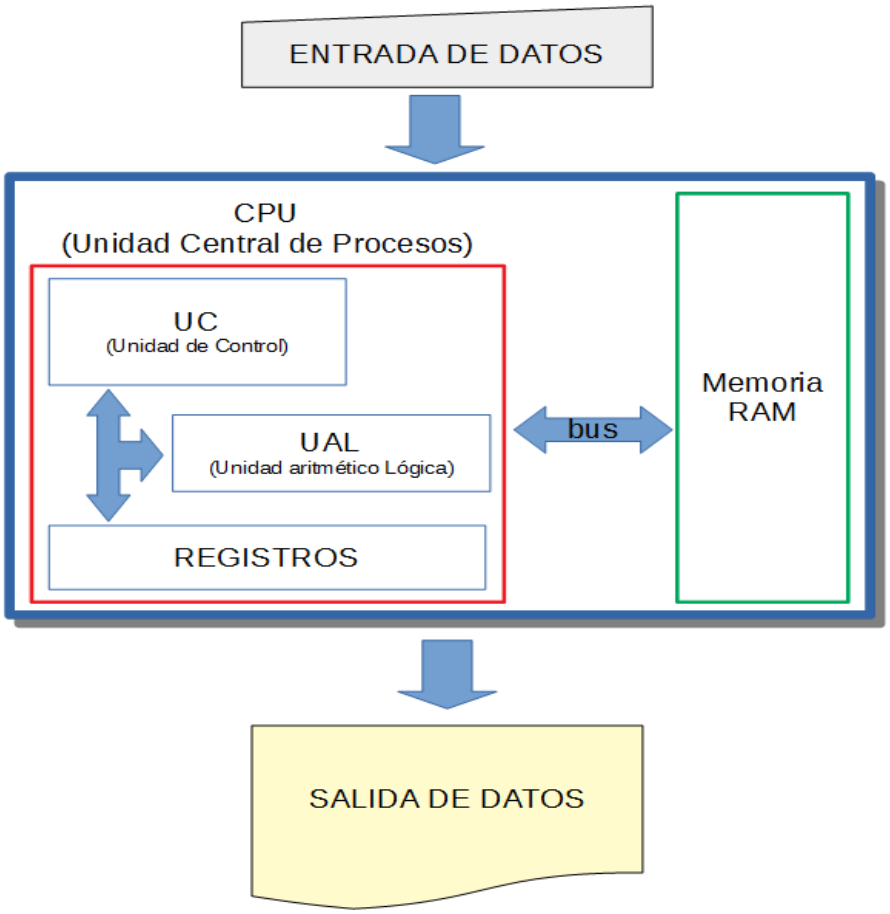
Los elementos que más destacan en un ordenador son el procesador y la memoria RAM. El procesador (CPU) es el encargado de ejecutar los programas, cuanto más rápido lo haga, mejor. Por su lado la memoria RAM almacena los programas que se están ejecutando en ese instante, por tanto, cuanta más capacidad tenga, más programas se pueden ejecutar a la vez.

La primera arquitectura hardware con programa almacenado se estableció en 1946 por **John Von Neumann**. Actualmente los ordenadores siguen manteniendo básicamente la misma arquitectura y modo de funcionamiento. El Sistema Operativo carga un programa en la memoria RAM para que se ejecute. El programa necesita unos datos de entrada (input) y ofrece tras la ejecución unos datos de salida (output). Este es el funcionamiento básico de cualquier programa. Por ejemplo, ejecutamos el programa procesador de textos LibreOffice Writer, por tanto el Sistema Operativo lo carga en memoria RAM. Empezamos a escribir una novela, es decir, estamos introduciendo datos al programa

mediante un teclado. Cuando la novela está terminada la imprimimos, estamos recibiendo unos datos de salida en la impresora conectada al PC.

> **RECUERDA**: Un programa cuando se está ejecutando está cargado en la memoria RAM, no vale con que esté almacenado en el disco duro.

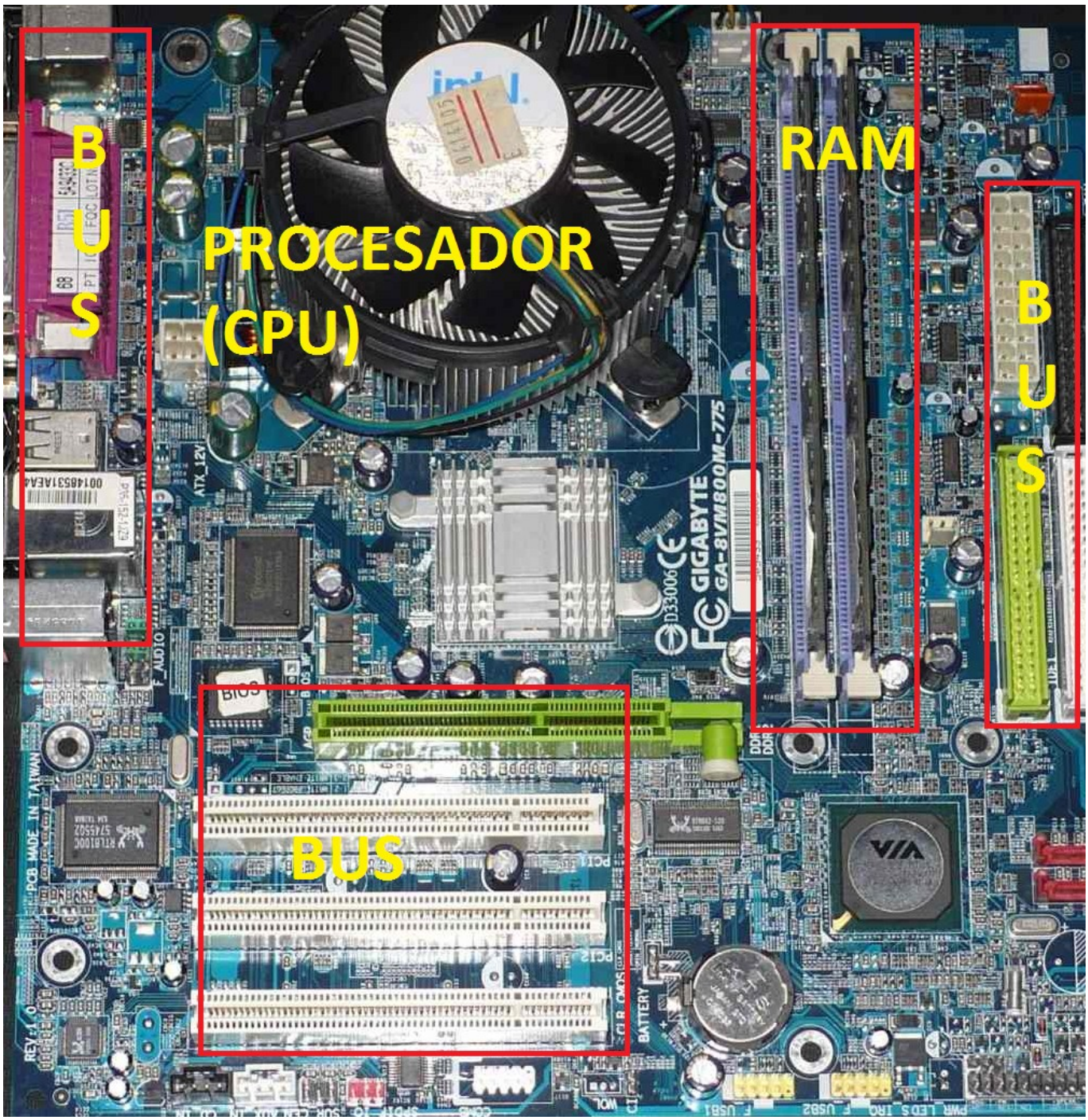
Para que los programas de los usuarios “no se tengan que pelear” con el hardware existe ese software especial denominado Sistema Operativo, el cual ya está cargado en parte en la memoria RAM del equipo. Por tanto, la relación entre la máquina (hardware) y el software de los usuarios (videojuego, calculadora, editor fotográfico,...), se hace a través del SO. El SO no deja de ser un programa, software.



Por tanto debe quedar claro que el SO es el encargado de coordinar al hardware durante el funcionamiento del ordenador, actuando como intermediario entre éste y las aplicaciones que están corriendo (ejecutándose) 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 SO el encargado de controlar todos estos aspectos de manera "**oculta**" para las aplicaciones y para el usuario. Por tanto, una aplicación de un usuario que se está ejecutando en un momento dado, por ejemplo, un

programa de retoque fotográfico como GIMP, si quiere mostrar una imagen en pantalla, mandará esa orden al SO, el cual se encargará de “negociar” con la tarjeta gráfica y el monitor.

En la siguiente imagen, se puede ver una foto real de la placa base de un ordenador tipo PC. Una placa base (motherboard) es un circuito impreso sobre el cual están soldados muchos elementos electrónicos, destacando principalmente el procesador (CPU) y la memoria RAM. En la imagen el procesador, está debajo del ventilador, ya que debido a las temperaturas que alcanza necesita normalmente de una refrigeración adicional. También destacan los BUS de comunicación. Estos son canales por donde fluyen los datos que se mueven entre distintos componentes de la computadora. En un ordenador también pueden insertar tarjetas que extienden las posibilidades de comunicación hacia el exterior. Si a un PC le conectamos una controladora de un robot industrial, mediante el software adecuado podremos manejar el robot . De esta forma, un ordenador mediante hardware y software es capaz de controlar cualquier otra máquina.



Vamos a utilizar un símil para entender bien los conceptos de hardware, software, lenguaje de programación, programa, programador y programación. Un músico (programador) lo es por que es capaz de escribir (programar) música (programas). Lo hace utilizando un lenguaje musical (lenguaje de programación), notas sobre un pentagrama. Pero ese pentagrama con sus notas (software, programa) en realidad no sirve de nada si alguien no lo interpreta sobre algún instrumento (hardware).

Ya hemos dicho que 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. Al igual que hay muchos idiomas, existen muchos de lenguajes de programación diferentes. Pero llegados a este punto hay que recordar

que 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 0s y 1s (código binario). Ese único lenguaje de programación que conoce el ordenador, se denomina código máquina y está muy alejado de la forma de trabajar las personas. Esto nos hace plantearnos una cuestión: ¿Cómo será capaz el ordenador de "entender" un programa escrito en un lenguaje que realmente no es el suyo?

Ejemplo de programa en lenguaje C que suma 3 números. No puede ser interpretado directamente por el hardware (ordenador), se necesitará otro programa que lo traduzca a código máquina:

```
#include<stdio.h>

#include<conio.h>

int main()
{
      int num1,num2,num3;

      printf("introduce 3 números: ");
      scanf("%d",&num1);      scanf("%d",&num2);      scanf("%d",&num3);

      printf("\n La suma es:%d",num1+num2+num3); getch();
      return 0;
}
```

Si no tienes experiencia programando, posiblemente el programa anterior te parezca muy difícil de entender, pero piensa que te pasó lo mismo la primera vez que viste un pentagrama con sus notas.

Los lenguajes de programación modernos se crearon para facilitar el entendimiento por parte de las personas, la otra alternativa sería escribir directamente en código máquina; si lo anterior te asustó, mira lo siguiente.

Ejemplo de programa codificado directamente en binario. Puede ser interpretado directamente por el hardware, pero muy difícil de entender para las personas. ¡No te asustes! Es muy raro enfrentarse a este tipo de codificación:

```
11000001    01000101    01011010    10101101    01010101    01011010    10101010
10101010    10101010    01100010    11111000    01100101    01010101    01010110
11101110    11010010    01011010    01101010    10110101    01010110    10101010
11111111    10110101    01010110    10101010    10101101    01011010    10101011
01010101    01001010    01011010    10101101    01010101    01011010    10101010
```

Ejemplo de programa en código máquina pero codificado en hexadecimal (base 16). Las secuencias numéricas son más cortas y se pueden llevar fácilmente a código binario. Si un día tienes que codificar en binario, posiblemente lo hagas usando hexadecimal.
45, 2A, 03, A5, FF, 23, CC, 23, 04, A5, B8, 2D, 44, A3, 98, 28, AA, 35, CD, 78, C2, CC, 23, 4, A5, B8, 2D, 44, A3, A3, 00, 00, 00, 00, 00, 02, AA, 35, CD, 78, 25, AA, BB, A2, 3D, 11, 19, 1A, 0D

¿Se utiliza esta codificación? Si, pero en casos muy especiales, por ejemplo para crear controladores (drivers) para hardware, para interpretar el funcionamiento del malware (software malicioso como los virus) y en pocas cosas más; la mayoría de los programadores no lo harán en su vida.

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 que no es el suyo (código máquina).

3 Desarrollo de Software.

Entendemos por Desarrollo de Software a 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, consta de una serie de pasos de obligado cumplimiento, pues solo así podremos garantizar que los programas creados sean eficientes, fiables, seguros y respondan a las necesidades de los usuarios finales (aquellos que van a utilizar el programa).

Genéricamente, estos pasos son los siguientes:

Según el orden y la forma en que se lleven a cabo las etapas anteriores hablaremos de los diferentes ciclos de vida del software.

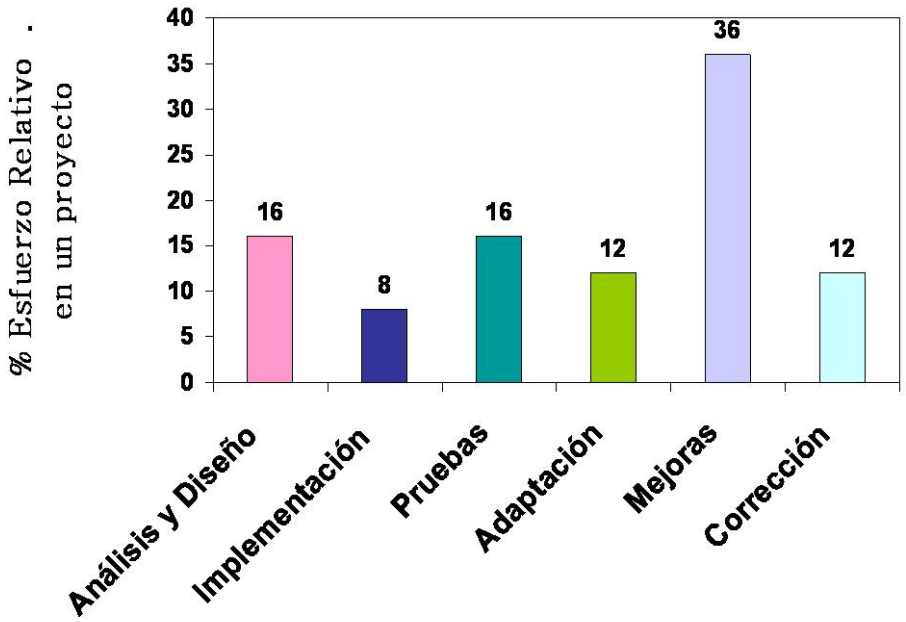
La construcción de software es un proceso que puede llegar a ser muy complejo y que exige gran coordinación y disciplina del grupo de trabajo que lo desarrolle.

3.1 Ciclos de vida del Software.

Podemos definir el Ciclo de Vida del Software como el conjunto de fases por las que pasa el sistema que se está desarrollando desde que nace la idea inicial hasta que el software es retirado o reemplazado por otro más adecuado. Es muy común el que software tenga actualizaciones a lo largo de su ciclo de vida.

Durante el ciclo de vida del software se realiza un reparto del esfuerzo de desarrollo del mismo en cada una de las fases que lo componen. La tabla siguiente muestra cuales son esas fases, y el gráfico que le sigue muestra el porcentaje de esfuerzo y por tanto de coste que supone cada fase sobre el total de un proyecto.

Ciclo de vida del Software	
Análisis y diseño.	Estudio del problema y planteamiento de soluciones.
Implementación (programación, codificación).	Confección de la solución elegida.
Pruebas.	Proceso para comprobar la calidad del producto.
Correcciones.	Solución de errores o ajustes para evitar problemas.
Adaptación.	Instalación al cliente para que pueda usarlo.
Mejoras.	Retoques que permiten hacer más útil el programa.



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

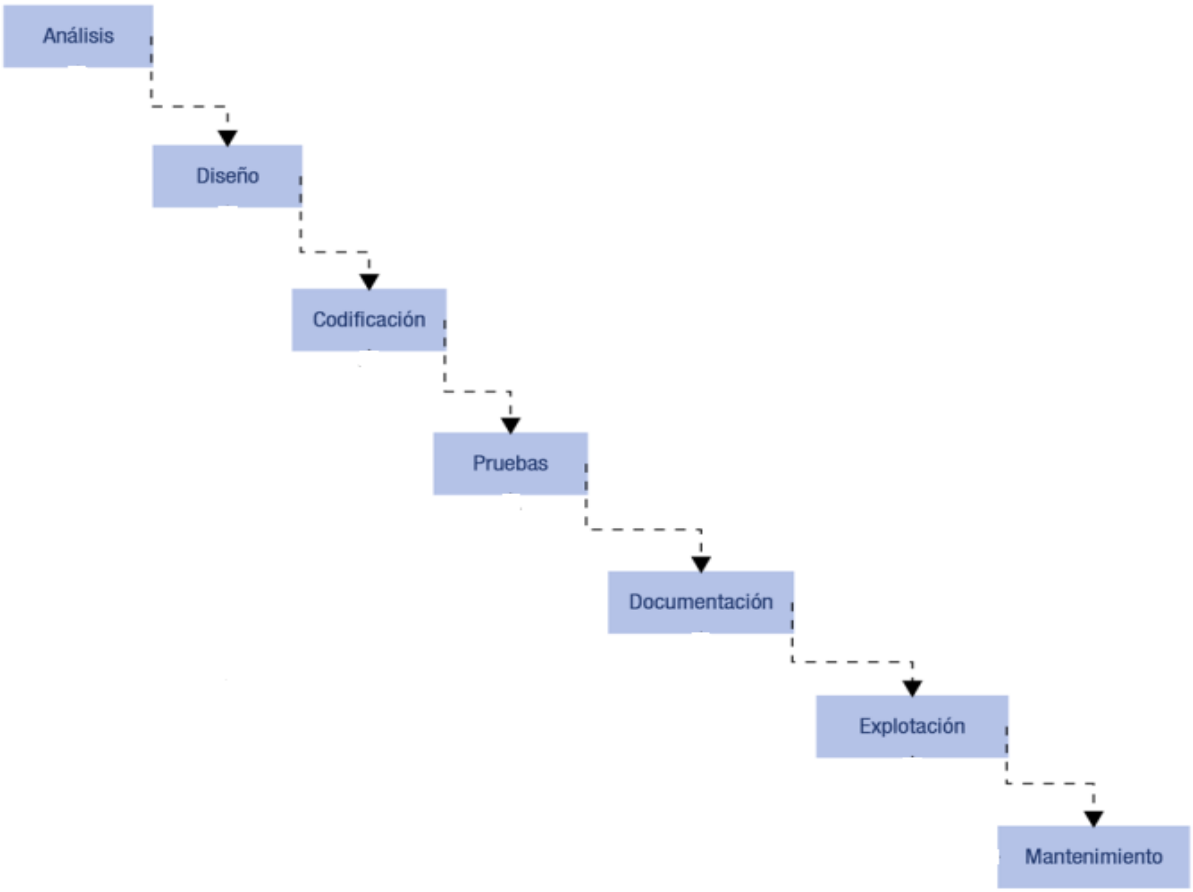
- **Modelo en Cascada.**

Es el modelo de vida clásico del software. Era la época de los primeros ordenadores, equipos grandes, lentos y con poca memoria, lo que se traducía en que los programas eran pequeños, básicamente algoritmos que realizaban un cálculo muy concreto.

Las fases se desarrollan una detrás de otra, el inicio de una fase no comienza hasta que termina la fase anterior. Es prácticamente imposible que se pueda utilizar actualmente, ya que requiere conocer de

antemano todos los requisitos del sistema. Sólo es aplicable a pequeños desarrollos, ya que las etapas pasan de una a otra sin retorno posible. (Se presupone que no habrá variaciones del software).

Esta forma de trabajar es la que se utiliza cuando una persona está aprendiendo programación, resolver programas de pocas líneas que básicamente constan de un solo algoritmo.



- **Modelo en Cascada con Realimentación.**

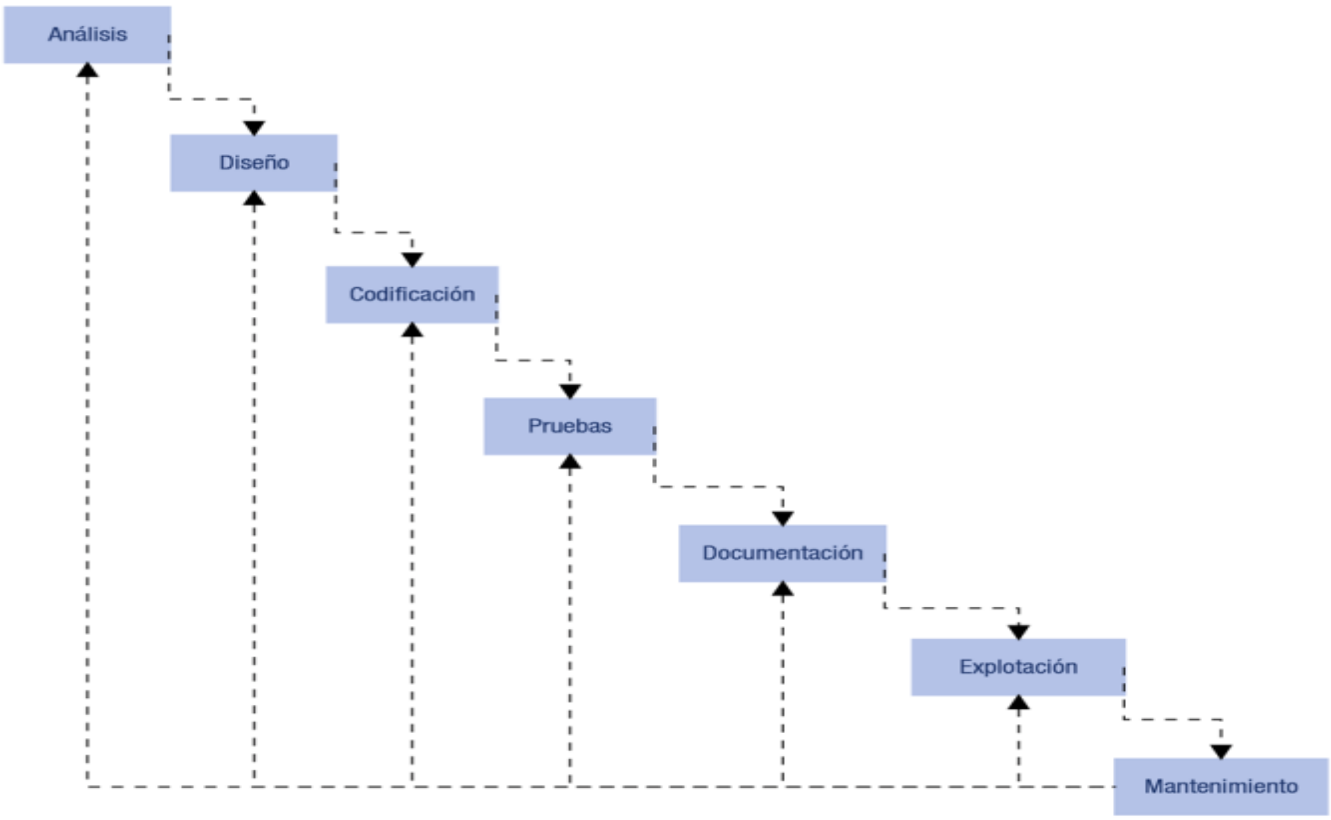
Es uno de los modelos más utilizados cuando se trabaja con programas no muy grandes.

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 algo que no funciona como se espera.

Si se prevén muchos cambios durante el desarrollo no es el modelo más idóneo, ya que estos pasos atrás suponen tiempo y dinero.

No se debe tomar el volver atrás como algo bueno, es más bien una necesidad. Si se hace a la ligera, el proyecto se eternizará y puede no llegar a culminar.

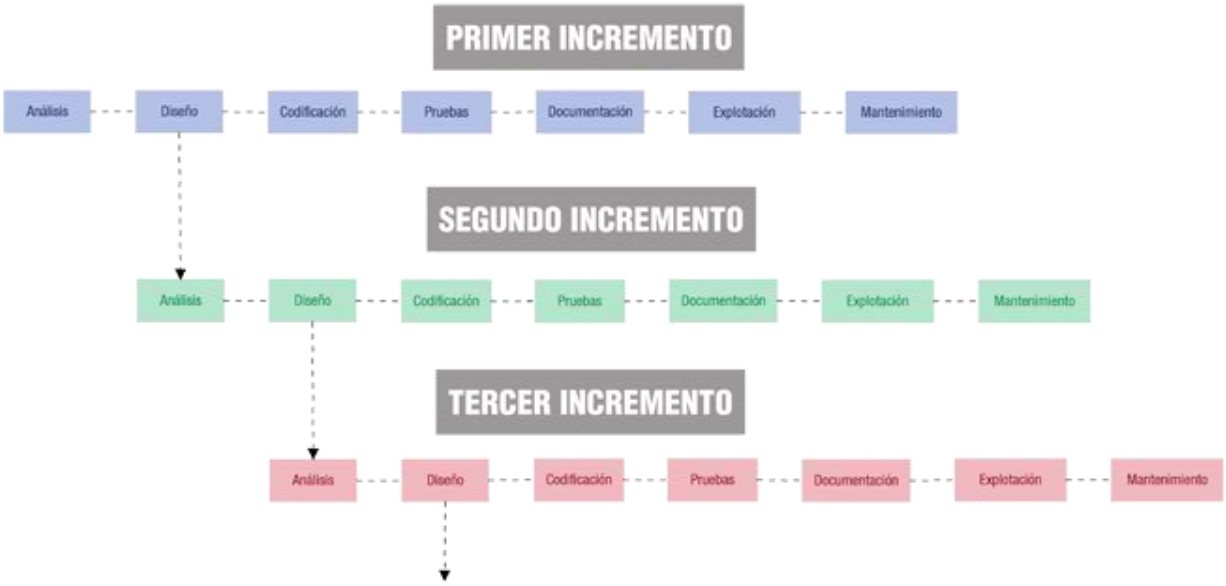
Es un modelo adecuado si el proyecto es rígido (pocos cambios, poco evolutivo) y los requisitos están claros.



- **Modelos Evolutivos**

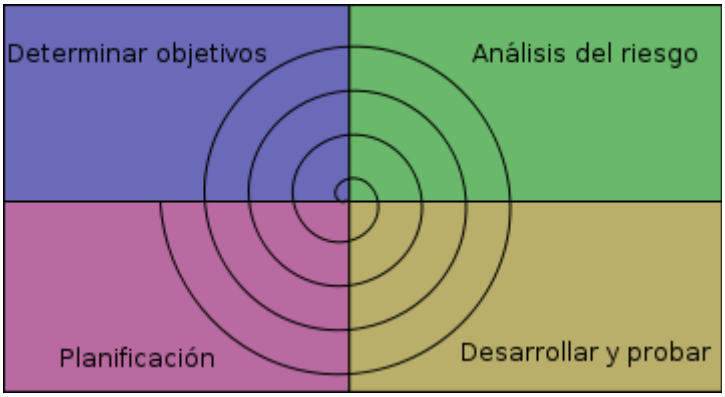
Son más modernos que los anteriores. Tienen en cuenta la mayor complejidad del software, su naturaleza cambiante y evolutiva. Distinguimos dos variantes.

- **Modelo Iterativo Incremental:** Se van liberando partes del producto (prototipos) periódicamente. En cada iteración (nueva versión), normalmente se aumenta la funcionalidad y se mejora la calidad respecto a la anterior.



Está basado en el modelo en cascada con realimentación, donde las fases se repiten, refinan, y van propagando sus mejoras a las fases siguientes. El resultado es que con cada incremento el software obtiene mejoras.
Una manera primordial de dirigir el proceso iterativo incremental es la de priorizar los objetivos y requerimientos en función del valor que ofrecen al cliente.

- **Modelo en Espiral:** Es una combinación del modelo anterior con el modelo en cascada. Se aúna la construcción iterativa de prototipos junto al hacer controlado y sistemático del modelo en cascada. Se suele interpretar como que dentro de cada ciclo de la espiral se sigue el modelo en cascada, pero no necesariamente debe ser así, ya que esto es difícil de conseguir.



Las actividades de este modelo se conforman en una espiral en la que cada bucle o iteración representa un conjunto de actividades. Las actividades no están fijadas a ninguna prioridad (al contrario que el modelo iterativo incremental), sino que las fases siguientes se eligen en función del análisis de riesgo, comenzando por el bucle interior.

Tiene muy en cuenta el riesgo que aparece a la hora de desarrollar software. Para ello, se comienza mirando las posibles alternativas de desarrollo, se opta por la de riesgo más asumible y se hace un ciclo de la espiral. Si el cliente quiere seguir haciendo mejoras en el software, se vuelve a evaluar las distintas nuevas alternativas y riesgos y se realiza otra vuelta de la espiral, así hasta que llegue un momento en el que el producto software desarrollado sea aceptado y no necesite seguir mejorándose con otro nuevo ciclo.

El software se va construyendo repetidamente en forma de versiones que son cada vez mejores, debido a que incrementan la funcionalidad en cada nueva versión.

En cada vuelta o iteración hay que tener en cuenta:

- **Objetivos**: Qué necesidades debe cubrir el producto.

- **Alternativas**: Las diferentes formas de conseguir los objetivos de forma exitosa, desde diferentes puntos de vista como pueden ser:
  Características: experiencia del personal, requisitos a cumplir, etc.
  Formas de gestión del sistema.
  Riesgo asumido con cada alternativa.

- **Desarrollo y Verificación**: Programar y probar el software.

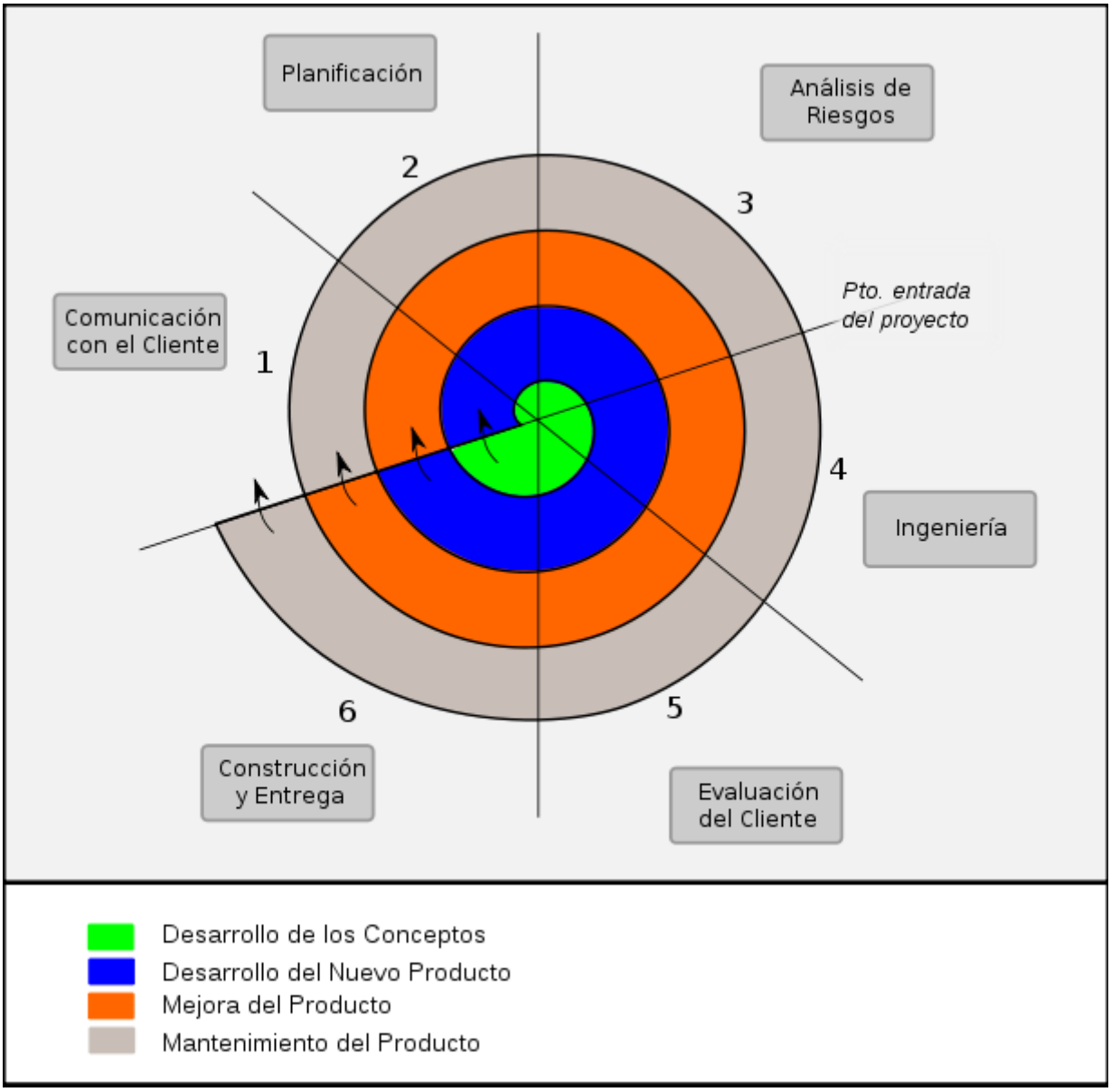
Si el resultado no es el adecuado o se necesita implementar mejoras o funcionalidades, se planificaran los siguientes pasos y se comienza un nuevo ciclo de la espiral.

La espiral se dice que mantiene dos dimensiones, la radial y la angular:

- **Angular**: Indica el avance del proyecto del software dentro de un ciclo.

- **Radial**: Indica el aumento del coste del proyecto, ya que con cada nueva iteración se pasa más tiempo desarrollando.

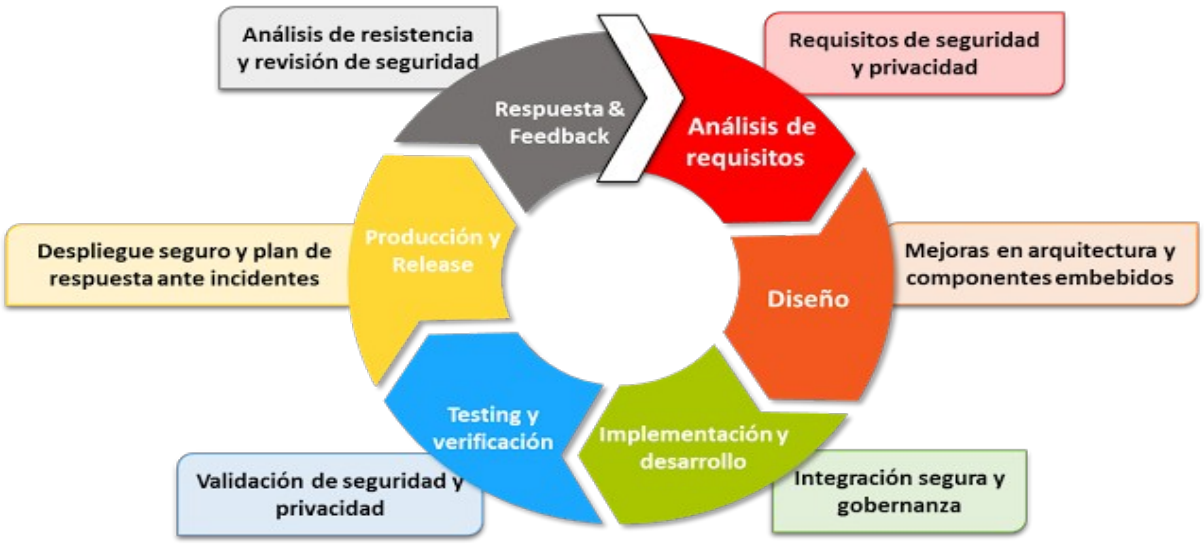
  Este sistema es muy utilizado en proyectos grandes y complejos como puede ser, por ejemplo, la creación de un Sistema Operativo.



3.2 Desarrollo y seguridad

En la actualidad, son muchas las aplicaciones que de una manera u otra están conectadas a Internet. La Seguridad y Privacidad por Diseño (SPD, o Security and Privacy by design) se define como el establecimiento de un marco de trabajo que de manera sistémica, metódica y formal introduce medidas de seguridad y privacidad a lo largo de todo el ciclo de vida del desarrollo software.

El problema está en que clientes, analistas y programadores tienden a centrarse en las funcionalidades del programa, olvidándose los temas de seguridad. Esto puede poner en grave peligro a las organizaciones que utilicen ese software ya que puede haber agujeros por los que se pueden colar cibercriminales.



3.3 Herramientas de apoyo al desarrollo del software.

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

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

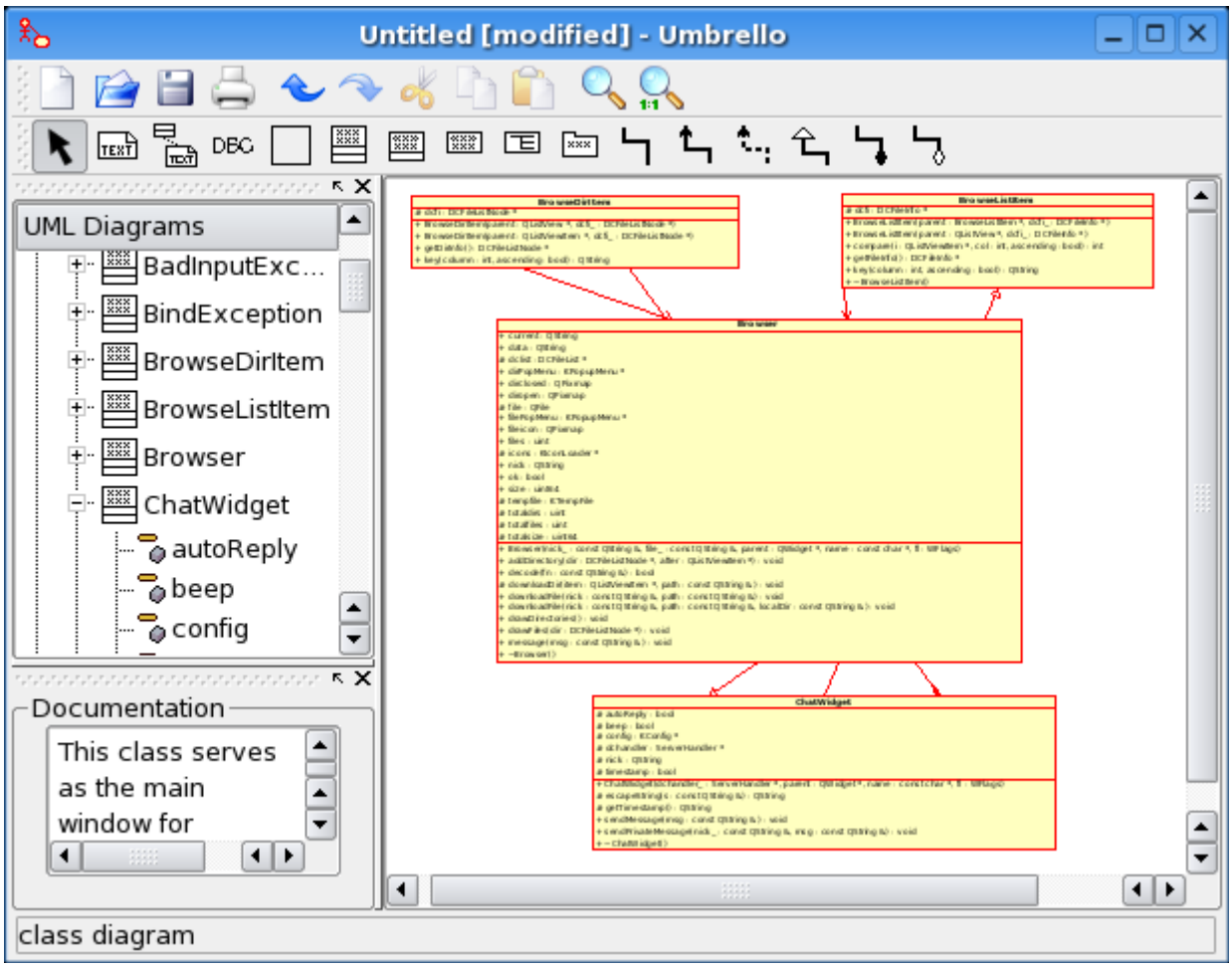
El denominado “desarrollo rápido de aplicaciones”, RAD (Rapid Application Development) es un proceso de desarrollo de software que comprende el desarrollo interactivo, la construcción de prototipos y el uso de herramientas CASE. Algunas de las plataformas más conocidas son Visual Studio, Lazarus, Gambas, Delphi, Foxpro , Anjuta, Game Maker, Velneo o Clarion.

Hoy en día se suele utilizar para referirnos al desarrollo rápido de interfaces gráficas de usuario o entornos de desarrollo integrados completos.

Las herramientas **CASE** (Computer Aided Software Engineering, Ingeniería de Software Asistida por Ordenador) son las herramientas de los desarrolladores de software. Son aplicaciones que se utilizan en el desarrollo de software con el objetivo de reducir costes y tiempo, mejorando por tanto la productividad del proceso.

En pequeños programas su uso puede no ser tan interesante, pero en grandes proyectos son imprescindibles. De cualquier forma es bueno que un programador comience a manejar estas herramientas lo más pronto posible y que vaya poco a poco eligiendo las más adecuadas para las tareas que suele realizar.

¿En qué fases del proceso nos pueden ayudar?
En todas las fases, en el diseño del proyecto, en la codificación, en la detección de errores…



4 Lenguajes de Programación.

Los programas informáticos están escritos usando algún lenguaje de programación. La programación directa en código máquina (el único lenguaje que realmente entiende una máquina) sólo se uso en los primeros ordenadores.

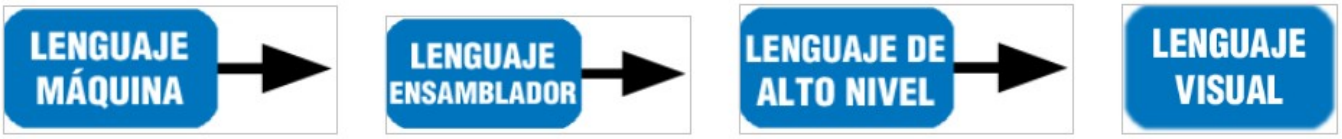
Pronto surgió el primer lenguaje de programación, ensamblador, en el cual es también muy difícil programar ya que básicamente es utilizar código máquina pero usando comandos textuales (mnemotécnicos). En la actualidad se usa muy poco, casi siempre relacionado con desarrollo de controladores (drivers) o en algunas tareas como puede ser el desarrollo de motores gráficos.

A continuación aparecieron los llamados **lenguajes de programación de alto nivel**, más cercanos a la forma de pensar y trabajar las personas.

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 una vez traducido a código máquina. Debe quedar claro, un programa escrito en un lenguaje de alto nivel, como Basic, C, o Java, no se puede ejecutar directamente, el ordenador no entiende ese tipo de código.

Los lenguajes de programación son los que nos permiten comunicarnos de forma más amable con el hardware del ordenador**.** 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."

Algunos lenguajes están enfocados o especializados en la programación de determinadas áreas (videojuegos, diseño gráfico, física, etc), mientras que la mayoría son más generalistas.



En la siguiente imagen histórica podemos observar los 10 lenguajes más utilizados en Octubre de 2012 según el índice TIOBE (comprueba que lenguajes de programación son los más utilizados, http://www.tiobe.com).

Position Oct 2012	Position Oct 2011	Delta in Position	Programming Language	Ratings Oct 2012	Delta Oct 2011	Status
1	2	↑	C	19.822%	+2.11%	A
2	1	↓	Java	17.193%	-0.72%	A
3	6	↑↑↑	Objective-C	9.477%	+3.23%	A
4	3	↓	C++	9.260%	+0.19%	A
5	5	=	C#	6.530%	-0.19%	A
6	4	↓↓	PHP	5.669%	-1.15%	A
7	7	=	(Visual) Basic	5.120%	+0.57%	A
8	8	=	Python	3.895%	-0.05%	A
9	9	=	Perl	2.126%	-0.31%	A
10	11	↑	Ruby	1.802%	+0.28%	A

Hay muchos lenguajes de programación, más de 50, y en cualquier instante puede surgir uno nuevo que se adapte mejor a las tecnologías y necesidades actuales. Fijándonos en el índice TIOBE se comprueba que no hay un lenguaje que gane claramente.

La elección del lenguaje a utilizar en un proyecto es una cuestión de importancia y vendrá dada por el tipo de proyecto y por la experiencia de los programadores.
Los lenguajes de programación han sufrido su propia evolución, acercándose cada vez más a la forma de comunicarse las personas y alejándose más del funcionamiento interno del hardware. Destaca una tendencia hacia los lenguajes más visuales donde se utilizan muchas herramientas gráficas.

Hoy los lenguajes siguen evolucionando, unos se adaptan, otros aparecen y otros quedan de desuso. En el año 2018, este es el índice TIOBE y la evolución de uso de los lenguajes de programación:

Sep 2018	Sep 2017	Change	Programming Language	Ratings	Change
1	1		Java	17.436%	+4.75%
2	2		C	15.447%	+8.06%
3	5	▲	Python	7.653%	+4.67%
4	3	▼	C++	7.394%	+1.83%
5	8	▲	Visual Basic .NET	5.308%	+3.33%
6	4	▼	C#	3.295%	-1.48%
7	6	▼	PHP	2.775%	+0.57%
8	7	▼	JavaScript	2.131%	+0.11%
9	-	⇈	SQL	2.062%	+2.06%
10	18	⇈	Objective-C	1.509%	+0.00%

Evolución de los Lenguajes de Programación:

- **Lenguaje máquina**
  - Fue el primer lenguaje utilizado.
  - Sus instrucciones (opcode) son combinaciones de unos y ceros, esto no quiere decir que haya que escribir en código binario, se suele utilizar mucho el hexadecimal y el decimal.
  - Es el único lenguaje que entiende directamente el procesador. (No necesita traducción).
  - Es único para la misma familia de procesadores, por tanto suele ser incompatible con otros procesadores.
  - Hoy en día nadie programa en este lenguaje, se utiliza el lenguaje ensamblador como aproximación a más bajo nivel.

- **Lenguaje ensamblador**
  - Sustituyó al lenguaje máquina para facilitar la labor de programación.
  - En lugar de los códigos numéricos representativos de cada comando, se programa usando mnemotécnicos (se da un nombre a cada secuencia numérica que recuerda su función).
  - Necesita traducción al lenguaje máquina para poder ejecutarse, pero es casi inmediata.
  - A pesar de ser más fácil que el código máquina, es muy difícil de utilizar y es particular para cada familia de procesadores.

- **Lenguajes de alto nivel**
  - Sustituyeron al lenguaje ensamblador (bajo nivel) para facilitar más la labor de programación. Se acerca más al lenguaje humano.
  - En lugar de mnemotécnicos, se utilizan textos (sentencias y órdenes) derivados del idioma inglés.
  - Son más cercanos al razonamiento humano.

- Necesitan una traducción más compleja al lenguaje máquina para poder ejecutarse.
- Son utilizados hoy día, aunque la tendencia es que cada vez se usen menos.
- Se pueden utilizar en distintas arquitecturas siempre que exista el traductor a código máquina.

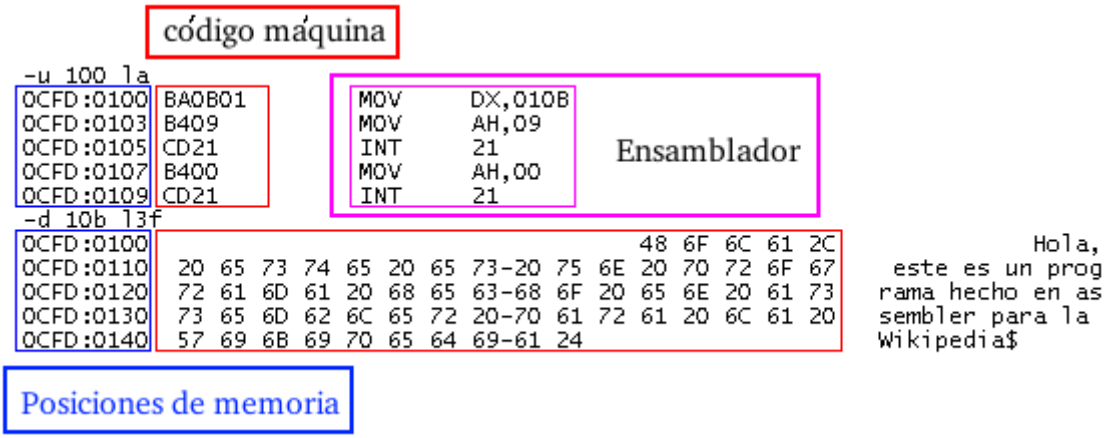
- **Lenguajes visuales**
  - Están sustituyendo a los lenguajes de alto nivel basados en código (texto).
  - Se programa gran parte del código gráficamente usando el ratón y diseñando directamente la apariencia del software. El código, en muchos casos, se genera automáticamente.
  - Necesitan traducción al lenguaje máquina.
  - Se pueden utilizar en distintas arquitecturas siempre que exista el traductor.

Vamos a ver que aspecto tiene un programa con varios ejemplos realizados en distintos lenguajes.

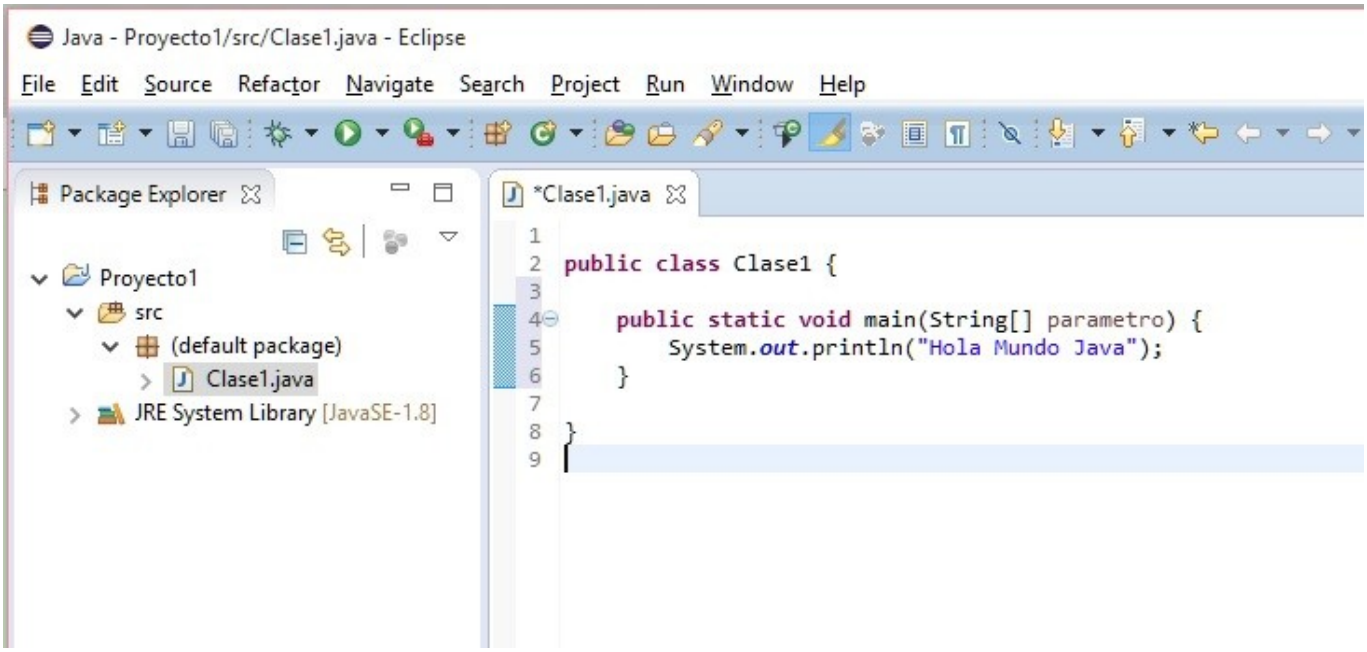
Ejemplo de programa que muestra el mensaje “Hola, este es un programa hecho en assembler para la Wikipedia”.

El programa está codificado en ensamblador para un ordenador tipo PC.

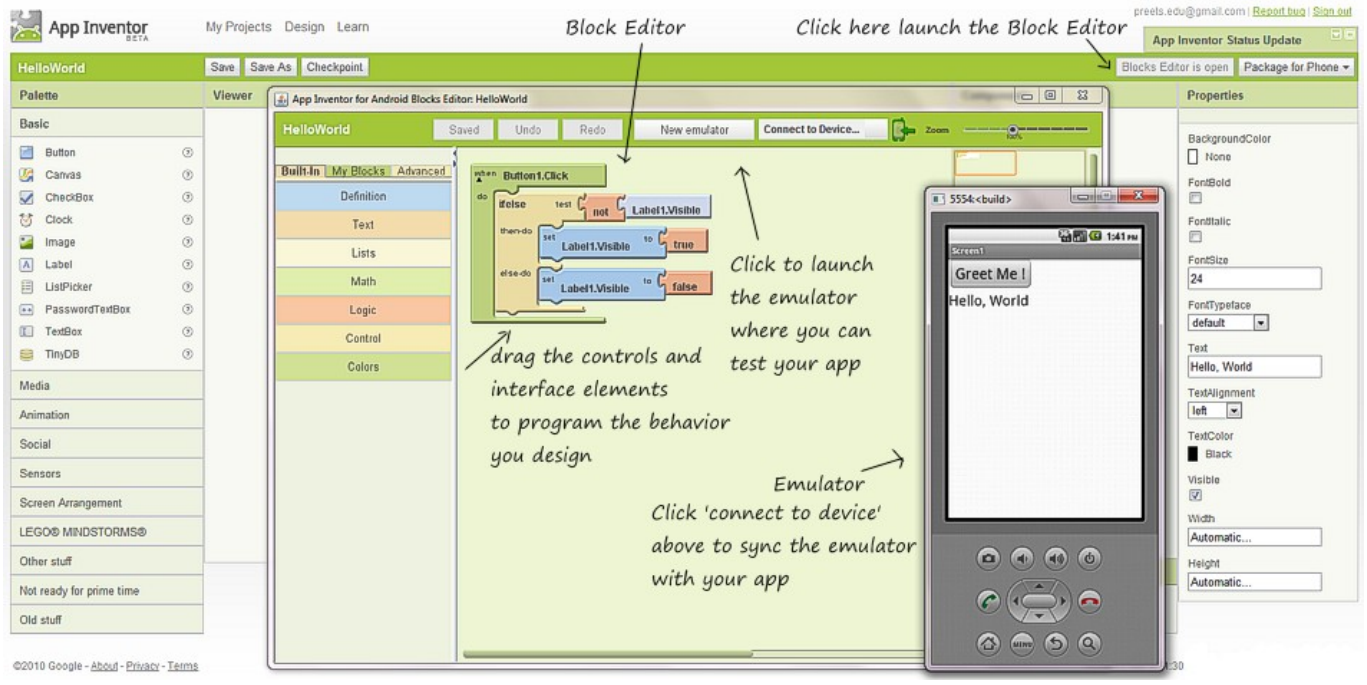
En la primera columna están las posiciones de memoria RAM en que se encuentra el código, en la segunda el código máquina, en la tercera el código ensamblador. Fijarse que se utiliza hexadecimal en la numeración.



Ejemplo de programa realizado en Java que muestra el mensaje “Hola Mundo Java”:



Ejemplo de programa para teléfono móvil que muestra el famoso mensaje “Hola mundo”, realizado con App Inventor, un entorno visual:



Para saber más

En el siguiente enlace, verás la cronología de los Lenguajes de Programación.

Cronología de los lenguajes de programación

4.1 Características y clasificación de los lenguajes de programación.

La elección del lenguaje de programación para codificar un programa dependerá de las características del problema a resolver. Pero esto no quiere decir que un programador tenga que conocer todos los lenguajes, normalmente sabe manejar uno muy bien y 2 o 3 más en los que se defiende.

Un lenguaje de programación, al igual que un idioma tiene unas normas y características:

- **Alfabeto (o léxico)**: conjunto de símbolos y palabras permitidas (vocabulario del lenguaje).
- **Sintaxis**: normas de construcción permitidas con los símbolos del lenguaje para formar instrucciones o sentencias.
- **Semántica**: significado de las construcciones para hacer acciones válidas.

La gran diferencia en el uso de un lenguaje de programación con respecto al idioma, es que el compilador de un ordenador va a ser muy exigente a la hora de respetar las formas del lenguaje, es decir, no vale escribir programas como habla Yoda, un humano puede entender algo como «Muchuo que apRender todaBía tenes.», pero un traductor de un lenguaje de programación es mucho más exigente con las normas.

Vamos a analizar una sentencia escrita en lenguaje C:

```
total = total + 1;
```

Podemos observar que aparecen caracteres, números y los símbolos de suma, igualdad y “punto y coma”. La sintaxis es correcta. La semántica, el significado, es que a la variable “total” se le suma 1.

Si la sentencia fuese:

```
niño = niño + 1;
```

Sería un error léxico, ya que no se permite el uso de la letra “ñ” en el nombre de una variable (no pertenece a los símbolos permitidos en el lenguaje). La mayoría de los lenguajes de programación usan el vocabulario inglés.

Mejor usar algo como:

```
ninyo ninyo +1;
```

Si la sentencia fuese:

```
total_final + 1 = total;
```

Esto sería erróneo, ya que la sintaxis no permite realizar una operación de suma en la parte izquierda de un “igual”. Lo correcto sería:

```
total_final = total + 1;
```

Si el código fuente fuese:

```
char iva[]=”21%”;
float total;
```

```
total = total + total*iva;
```

No sería correcto, no tiene sentido (error en la semántica) ya que "iva" está declarado como una cadena de caracteres, mientras que total es un número en coma flotante (real). No se puede sumar un número a una cadena.

Clasificación de los lenguajes de programación:
Podemos realizar diferentes clasificaciones de los Lenguajes de Programación en base a distintas características:

- **Según el nivel de abstracción**, o sea, según lo cerca que esté del lenguaje humano:
  - **Lenguajes de Programación de Alto nivel**: por su esencia, están más próximos al razonamiento humano. (Java, Pascal, Basic, C++,…)
  - **Lenguajes de Programación de Bajo nivel**: están más próximos al funcionamiento interno de la computadora:
    - Lenguaje Ensamblador.
    - Lenguaje Máquina (código máquina).
- **Según el propósito**, es decir, el tipo de problemas a tratar con ellos:
  - **Lenguajes de propósito general**: Aptos para todo tipo de tareas, por ejemplo Java.
  - **Lenguajes de propósito específico**: Hechos para un objetivo muy concreto, por ejemplo el lenguaje CSound está pensado para la creación de música.
  - **Lenguajes de programación de sistemas**: Diseñados para realizar sistemas operativos o drivers, por ejemplo el C.
  - **Lenguajes de script**: para realizar tareas de control y auxiliares. También llamados lenguajes de procesamiento por lotes (batch) o JCL("Job Control Languajes").
- **Según la evolución histórica**:
  - **Lenguajes de primera generación (1GL)**: Código máquina.
  - **Lenguajes de segunda generación (2GL)**: Lenguajes ensamblador.
  - **Lenguajes de tercera generación (3GL)**: La mayoría de los lenguajes modernos, diseñados para facilitar la programación a los humanos. Ejemplos C, Java.
  - **Lenguajes de cuarta generación (4GL)**: Diseñados para reducir e esfuerzo de programación y el tiempo que se tarda en desarrollar software. Ejemplos: NATURAL, Mathematica.
  - **Lenguajes de quinta generación (5GL)**: La intención es que el programador establezca qué problema ha de ser resuelto y las condiciones a reunir, y la máquina lo resuelve. Se usan en inteligencia artificial. Ejemplo: Prolog.
- **Según la manera de traducirse**:
  - **Lenguajes compilados**: Un programa traductor traduce el código del programa (código fuente) en código máquina (código objeto). Otro programa, el enlazador, unirá los ficheros de código objeto del programa principal con los de las librerías para producir el programa ejecutable. Ejemplo el C.
  - **Lenguajes interpretados**: Un programa traductor denominado interprete, ejecuta las instrucciones del programa de manera directa. Ejemplo el Lisp, Basic, Python.
  - **Lenguajes Mixtos**, como Java, que primero pasan por una fase de compilación y luego es interpretado.
- **Según la manera de abordar la tarea a realizar**:

- **Lenguajes imperativos**: Indican cómo hay que hacer la tarea, es decir, expresan los pasos a realizar. Ejemplo el PASCAL, C, Basic.

- **Lenguajes declarativos**: Indican que hay que hacer. Ejemplos: Lisp, Prolog. Otros ejemplos de lenguajes declarativos, pero que no son lenguajes de programación, son HTML (para describir páginas Web) o SQL (para consultar Bases de datos).

- **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**: Usan la técnica de programación orientada a objetos. Ejemplos: C++, Java, C#, Ada, Delphi, etc.

  - **Lenguajes de Programación Visuales**: Basados en las técnicas anteriores, permiten programar gráficamente, siendo el código correspondiente generado en parte de forma automática. Ejemplos: Visual Basic.Net, Delphi, etc.

Para saber más

En la página web siguiente encontrarás un resumen de las características de los Lenguajes de Programación más utilizados en la actualidad.

Características de los Principales Lenguajes de Programación

4.2 Programación estructurada y modular.

Aunque los requerimientos actuales de software son bastante más complejos de lo que la técnica de programación estructurada es capaz de abordar, 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 más se usan actualmente.

Los lenguajes de programación que utilizan la programación estructurada permiten 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.

Ejemplo de Programa en C que escribe los números del 4 al 15. Aparecen los 3 tipos de sentencias:

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
int main(void)
{
    int min,max; int i; //declaración de variables
    min=4; //sentencia secuencial
    max=15; //sentencia secuencial
    if (min>max) //sentencia selectiva
    {
                printf("Valores incorrectos"); getch(); exit(1);  // 3 sentencias
secuenciales
    }
    for (i=min;i<=max;i++) //sentencia repetitiva (bucle)
        printf("%d \n",i);
    getch(); //sentencia secuencial
    exit(0); //sentencia secuencial
}
```

La programación estructurada fue de gran éxito por su sencillez a la hora de construir algoritmos que resolvían problemas. Un **algoritmo** es un conjunto de instrucciones bien definidas, ordenadas y finitas que permite resolver un problema mediante pasos sucesivos que no generen dudas y llevan a la resolución final del problema. En un programa pueden aparecer muchos algoritmos.

Propiedades de un algoritmo:

- **Precisión**: No puede tener ambigüedades en cada paso a seguir.
- **Exactitud**: Ante los mismos datos los resultados deben ser los mismos y correctos.
- **Finito**: Su ejecución debe concluir en algún momento, aunque aquí podríamos tener excepciones, por ejemplo, si queremos representar un reloj, su funcionamiento podría ser infinito. Al hablar de finito nos referimos a que si en el procesamiento de los datos hay que alcanzar un resultado final, en algún momento hay que acabar para representar ese resultado.

A medida que los programas se fueron haciendo más grandes, apareció la **programación modular**, que permitía dividir los programas grandes en trozos más pequeños llamados módulos o subprogramas, siguiendo la conocida técnica "divide y vencerás". Esto permite trabajar a más programadores sobre un mismo proyecto, cada uno en un subprograma. Se puede probar independientemente cada módulo, con lo que es más fácil encontrar los errores. En cada subprograma se utiliza la programación estructurada para resolver una parte concreta del programa. La suma de todos los subprogramas resuelven todas las necesidades planteadas.

Más tarde con la evolución del hardware de los ordenadores, aparecieron los entornos gráficos, lo que propició un nuevo paradigma, **la programación orientada a objetos** donde la reutilización del código cobró gran importancia. Tras el triunfo de los lenguajes orientados a objetos, el siguiente paso fue la **programación visual** con el objetivo de facilitar más aún la programación. Las técnicas de programación estructurada y modular se siguen utilizando en la programación orientada a objetos y visual.

- **Ventajas de la programación estructurada en aquella primera época**:
  - Los programas son fáciles de leer, sencillos y rápidos (desde el punto de vista pasado, en que los programas eran pequeños debido a las características del hardware de la época).
  - Los realiza un solo programador.
  - El mantenimiento de los programas es sencillo.
  - La estructura del programa es sencilla y clara.
- **Inconvenientes**:
  - Todo el programa se concentra en un único bloque, si se hace demasiado grande es difícil de manejar y comienzan a aparecer muchos errores.
  - No permite una reutilización eficaz del código, ya que todo va "en un bloque".

Ejemplos de lenguajes estructurados: Pascal, C, Fortran, Cobol.

4.3 Lenguajes de programación orientados a objetos.

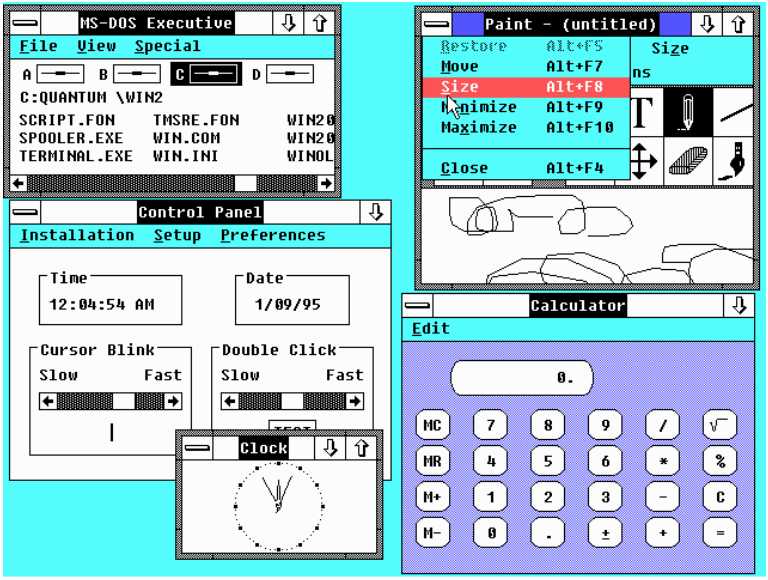
A medida que pasó el tiempo, el hardware de los equipos se hizo más potente (más capacidad en los discos duros, más RAM, procesadores más rápidos) por lo que el software también evolucionó para resolver problemas más complejos. Además aparecieron las pantallas a color, las tarjetas gráficas y el ratón. Todo esto llevo al uso de interfaces gráficas que facilitan el uso del Sistema Operativo y de los programas.

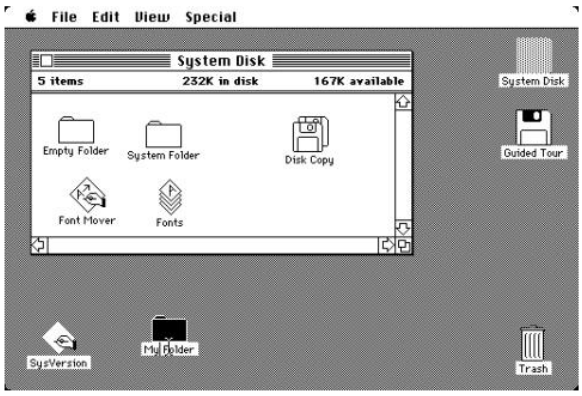
El resultado es que los programas son cada vez más grandes, tienen muchísimas más líneas de código, lo que conlleva más tiempo para su creación y por todo ello es más fácil cometer errores.
Se necesitaba una evolución en los lenguajes de programación, la programación estructurada y modular no son suficientes. Nace así la Programación Orientada a Objetos (POO), una nueva forma de realizar programas que se suma a las técnicas anteriores.

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 pueden colaborar entre ellos para realizar acciones.

Simulando la vida real, los programas representan objetos. Por ejemplo el “botón imprimir” de un programa simula un botón real de una impresora que al pulsarlo permite imprimir un documento.

En la siguiente imagen podemos ver las primeras interfaces gráficas utilizadas en los Sistemas Operativos de Windows y Mac. La orientación a objetos facilitó mucho su desarrollo.





PRIMERAS INTERFACES GRÁFICAS UTILIZADAS EN LOS SISTEMAS OPERATIVOS

Cada objeto tiene unas características y es capaz de realizar ciertas funciones. La idea es acercarse a la realidad, donde interactuamos continuamente con objetos.

Por ejemplo, un reloj es un objeto real que muestra la hora, lo podemos poner en hora, tiene un tamaño, una forma, un color, etc. Todo eso lo llevamos al mundo de la programación y estaríamos hablando del objeto reloj que realiza las mismas funciones.

Características:

- Se define “clase” como una colección de objetos con características similares. Una clase es una abstracción de algo, la idea general que tenemos de cualquier cosa. Pensemos en la idea de lo que es un coche, ese pensamiento es la abstracción, lo que es un coche en general. Mira por la ventana y busca un coche, ese es un objeto concreto, es de color rojo, mide 4 metros de largo, una potencia de 100 CV, etc.

- Los objetos de un programa tendrán una serie de atributos (color, tamaño, etc) que pueden tener un valor concreto en cada uno.
- Mediante los llamados métodos, los objetos realizan cosas y se pueden comunicar con otros produciéndose un cambio de estado de los mismos.

- Los objetos son, pues, como unidades individuales e indivisibles que forman la base de este tipo de programación.

Principales ventajas del uso de objetos:

- Los objetos son reutilizables en otros programas.

- Si hay algún error, es más fácil de localizar y de depurar en un objeto que en todo un programa.

Algunos lenguajes orientados a objetos: Ada, C++, Python, Delphi, Java, PowerBuilder, etc.

5 Fases en el desarrollo y ejecución del software.

Hemos visto que debemos elegir un modelo de ciclo de vida para el desarrollo de nuestro software. Independientemente del modelo elegido, siempre hay una serie de fases o etapas que debemos seguir para construir software fiable y de calidad.

Estas fases son:

1. **Análisis**: Se especifican los requisitos funcionales y no funcionales del sistema.
2. **Diseño**: Se divide el sistema en partes y se determina la función de cada una.
3. **Codificación** (programación): Se elige un Lenguajes de Programación y se codifican los programas.
4. **Pruebas**: Se prueban los programas para detectar errores y se depuran.
5. **Documentación**: De todas las etapas, se documenta y guarda toda la información.
6. **Explotación**: Instalamos, configuramos y probamos la aplicación en los equipos del cliente.
7. **Mantenimiento**: Se mantiene el contacto con el cliente para actualizar y modificar la aplicación el futuro.

5.1 Análisis.

Esta es la primera fase del proyecto. Si un cliente nos solicita hacer un programa, lo primero es saber que tiene que hacer ese programa. Mediante conversaciones con los clientes y futuros usuarios del programa, se tratará de conocer exactamente que funciones tiene que realizar la aplicación. Es una fase muy complicada, de mucho diálogo y la que más depende de la capacidad y pericia del analista para tratar con el cliente y poder extraer toda la información necesaria. Si Hay algún tipo de malentendido, o alguna información que no se recogió, a partir de ahí todo el proyecto irá mal.

Para darse una idea, voy a un arquitecto para que diseñe la casa de mis sueños y no comento que la quiero de 3 plantas. Si el arquitecto supone que es de una sola planta y así realiza los diseños, cuando se construya la casa....¡sorpresa!

Una vez finalizada la fase de análisis, pasamos a la fase diseño. Un arquitecto pasaría a dibujar los planos.

Todas las fases son importantes en el desarrollo de un proyecto, pero esta al ser la primera y teniendo en cuenta que las demás dependen de ella, puede que sea la fase de mayor transcendencia en el desarrollo del proyecto, especialmente si nos vemos sometidos a los gustos de un cliente. Es diferente si somos nosotros mismos los que desarrollamos un proyecto a nuestro gusto. Todas las demás fases dependerán de lo bien detallada que esté la fase de análisis.

¡Como vamos a hacer un buen programa si no queda claro que tiene que hacer ese programa!

Otra complicación de esta fase es que **está poco automatizada**, no tenemos buenas herramientas que nos permitan saber exactamente que es lo que quiere un cliente que haga su programa. Muchas veces los clientes olvidan contar cosas, otras veces suponen que los programadores conocen su negocio y dejan de contar detalles importantes. A esto se suma que el cliente tampoco suele tener del todo claro que tiene que hacer el programa. Por todo esto, un buen análisis **depende en gran medida de la pericia del analista**

que lo realice. Lo fundamental es la buena comunicación entre el analista y los clientes para que la aplicación que se va a desarrollar cumpla con sus expectativas.

¿Cuál es el resultado de esta fase? Se obtienen los requisitos funcionales y no funcionales del sistema.

- **Requisitos Funcionales**: Funciones a realizar por la aplicación, qué respuesta dará la aplicación ante todas las entradas de datos, procesos a realizar con los datos, cómo se comportará la aplicación en situaciones inesperadas. Por ejemplo algunas funciones típicas de un programa de retoque de fotografía permiten cambiar un color, borrar zonas de la imagen, añadir texto, etc.
- **Requisitos no funcionales**: Son restricciones o elementos a tener en cuenta pero que no tienen que ver con lo que tiene que hacer la aplicación. Estos requisitos están relacionados con el rendimiento del programa, tiempos válidos de respuesta ante ciertas consultas, legislación aplicable, estándares de calidad, accesibilidad, tratamiento ante la simultaneidad de peticiones, portabilidad, etc.
  Ejemplos: Un programa de ventas puede requerir el uso de un lector de código de barras para saber los productos que se compran. Un programa que debe de ocupar menos de 50 MB en memoria. Un programa que debe permitir cambios en los ficheros de datos por más de un usuario a la vez.

Ejemplo de algunos **requisitos funcionales** para una aplicación de una tienda de cosmética:

- Para los trabajadores que cobran por comisión hay que llevar un seguimiento individual de las ventas realizadas.
- El programa puede emitir facturas y albaranes.
- Se llevará por separado las ventas con pagos al contado y con tarjeta.
- Se realizará un control de stock en el almacén.
- Se desea que la lectura de los productos se realice mediante códigos de barras.

Ejemplo de algunos **requisitos no funcionales**:

- El programa debe funcionar en distintas plataformas (PC y móvil).
- El tipo de lector de códigos de barras a utilizar para vender productos.
- Las leyes de comercio electrónico que hay que cumplir.
- El tiempo de respuesta admisible ante una consulta vía web.
- El programa debe estar codificado en Java.

La culminación de esta fase de análisis es el documento ERS (Especificación de Requisitos Software). En este documento quedan especificados:

- La planificación de las reuniones que van a tener lugar.
- Relación de los objetivos del usuario cliente y del sistema.
- Relación de los requisitos funcionales y no funcionales del sistema.
- Relación de objetivos prioritarios y temporalización.
- Reconocimiento de requisitos mal planteados o que conllevan contradicciones, etc.

Ya sabemos lo que el usuario quiere, ahora, antes de hacer nada más, el usuario querrá saber cuanto le va a costar. Esta no es una tarea nada fácil. La elaboración de software es una actividad muy compleja donde no se depende tanto de maquinaria sino de trabajo intelectual. Además cada proyecto software fácilmente tendrá poco que ver con otro, por lo que las experiencias anteriores no son tan fácilmente aprovechables como en otras ingenierías. Por todo ello realizar un análisis técnico y económico del sistema es una tarea compleja pero que hay que intentar realizar con la mayor precisión posible. Entre otras decisiones a tomar, habrá que evaluar la viabilidad del sistema y establecer restricciones de costo y tiempo.

5.2 Diseño.

En esta fase ya sabemos lo que tiene que hacer la aplicación, el siguiente paso es ¿Cómo hacerlo?

Un arquitecto ya sabe como es la casa de nuestros sueños, ahora pasará a diseñar los planos.

Esta fase se la puede considerar **la primera en lo que se refiere al desarrollo del producto**, aunque el producto realmente aquí no se va a empezar a elaborar. Si pensamos en la fabricación de un coche, en esta fase tendremos sobre el papel, sobre el ordenador, o sobre arcilla, modelos en 3D que ya permiten entrever como tiene que ser el producto final. El objetivo del diseñador es producir un modelo o representación de una entidad que será construida mas adelante.

Esta etapa se suele dividir en 4 fases:

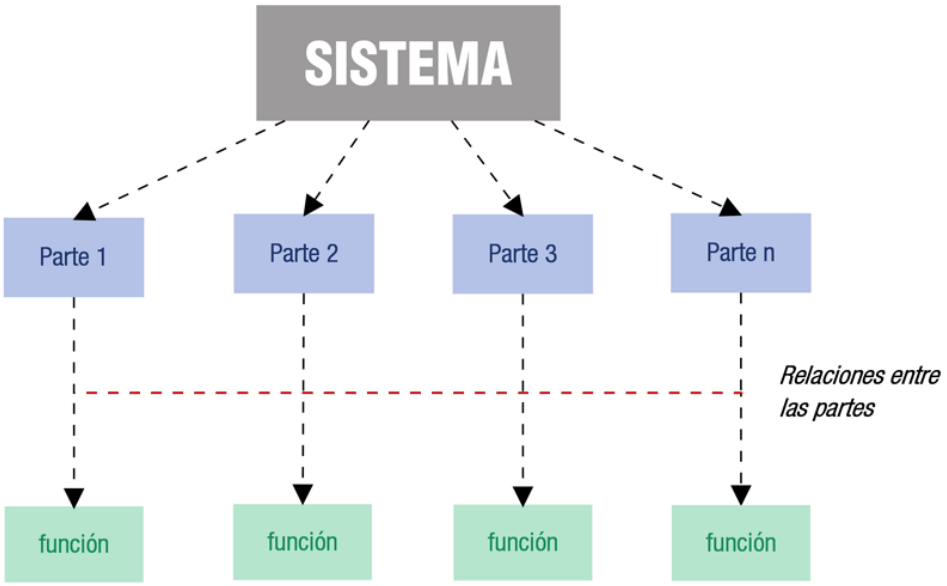
1. Diseño de datos. Donde se especifican todos los datos, sus tipos y las estructuras que los contienen (por ejemplo las bases de datos).
2. Diseño arquitectónico. Define las clases.
3. Diseño de la interfaces entre las clases.
4. Diseño Procedimental. Donde se crean los algoritmos que resuelven cada problema.

En aplicaciones grandes se debe dividir el sistema en partes y establecer qué relaciones habrá entre ellas. Hay que 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:

- Selección del Sistema Gestor de Base de Datos.
- Entidades y relaciones de las bases de datos.

- Selección de los lenguaje de programación que se van a utilizar.
- Diseño de las interfaces para la entrada y salida de datos. Etc.



5.3 Codificación.

Durante la fase de codificación se realiza el proceso de programación, escribir el programa.
Consiste en elegir el lenguaje de programación (uno o varios) y codificar toda la información anterior obteniendo el llamado código fuente del programa.

RECUERDA: El programa escrito en un lenguaje de programación diferente del código máquina, se denomina código fuente. Si somos programadores de Java, el programa escrito en Java es el código fuente. El programa listo para ser ejecutado en un ordenador directamente es el código máquina.

Esta tarea de programación o codificación la realizan los programadores y tiene que cumplir exhaustivamente con todos los requisitos impuestos en el análisis y en el diseño de la aplicación.

Las características deseables de todo código fuente son:

- **Modularidad**: El programa está dividido en trozos más o menos pequeños donde se resuelvan problemas concretos.
- **Corrección**: que haga lo que se le pide realmente.
- **Fácil de leer**: para facilitar su desarrollo entre varios programadores y su mantenimiento futuro. Hay que pensar que los cambios los pueden tener que realizar otros programadores.
- **Eficiencia**: que haga un buen uso de los recursos (ocupación en memoria RAM, espacio en disco, etc).
- **Portabilidad**: que se pueda ejecutar en cualquier equipo.

5.3.1 Fases para la obtención de código ejecutable.

Sabemos que un programa escrito en un lenguaje de programación distinto al código máquina no puede ser ejecutado directamente por el ordenador. Un programa pasa por diferentes estados antes de estar listo para su ejecución. Estos estados dependen del tipo de traductor a código máquina utilizado. Los traductores a código máquina son los compiladores y los intérpretes.

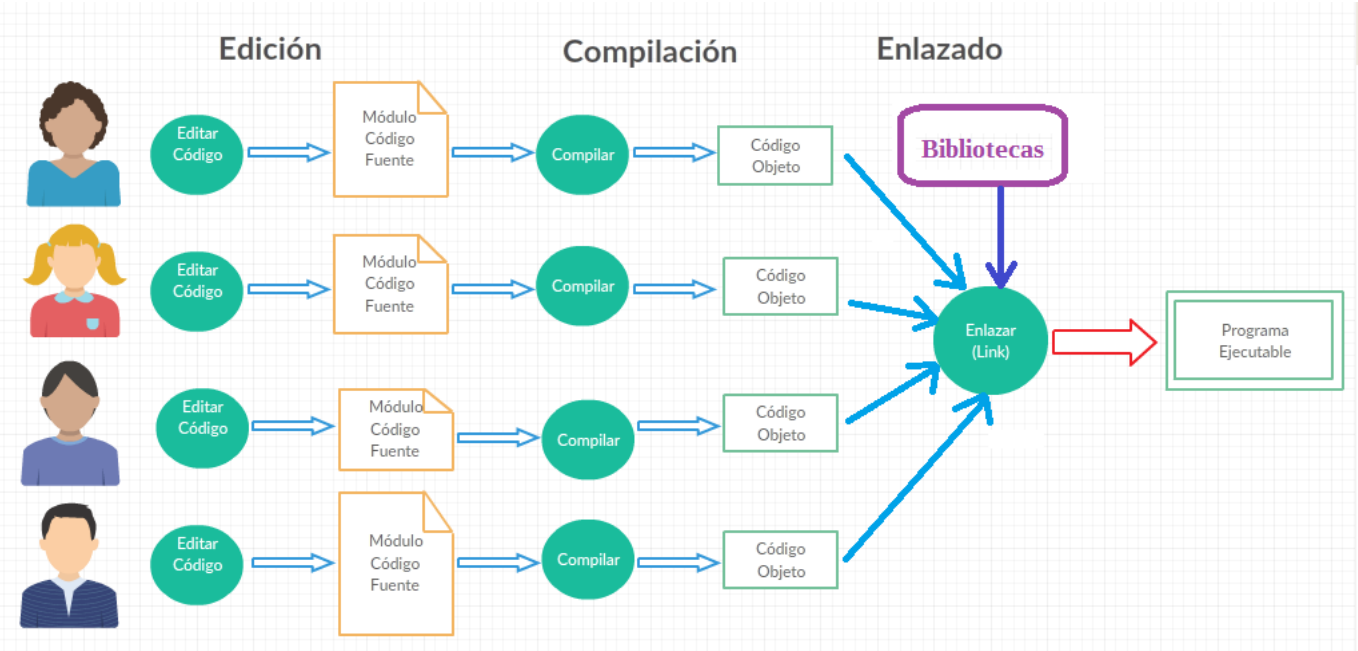
En el caso de utilizar un compilador para traducir nuestro programa a código máquina, nuestro código pasa por las siguientes fases:

Código Fuente: es el programa escrito por los programadores con la ayuda de un editor de texto. Se escribe usando un lenguaje de programación (por ejemplo lenguaje C) y contiene el conjunto de instrucciones necesarias para resolver todas las tareas planteadas por el cliente. Normalmente un programa está dividido en varios módulos subprogramas. Cada programador trabaja con una parte del código fuente, con un módulo.

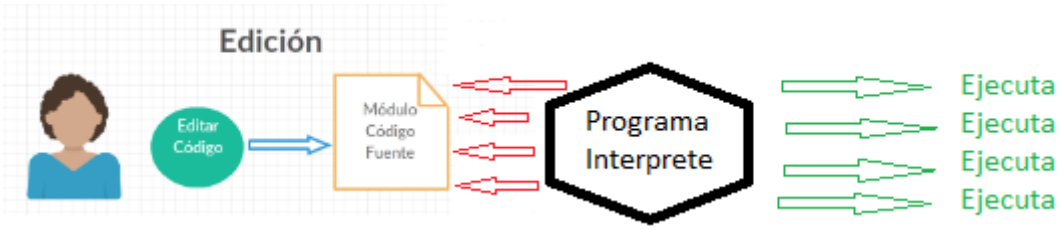
Código Objeto: es el código binario resultado de compilar cada módulo del código fuente. Consiste en lenguaje máquina o "bytecode" y se distribuye en varios archivos que corresponden a cada parte del código fuente compilado. Para obtener un programa ejecutable se han de **enlazar todos los archivos de código objeto** con un programa llamado enlazador (linker), obteniendo el programa ya ejecutable.

Código Ejecutable: Es el código binario resultante de enlazar los archivos de código objeto con ciertas rutinas y bibliotecas (o librerías) necesarias. Es el programa escrito en código máquina y es directamente inteligible por la computadora. Cuando un usuario manda ejecutar un programa, es el sistema operativo el encargado de cargar el código ejecutable en memoria RAM y proceder a ejecutarlo.

Una vez se obtiene el programa en código máquina, ya no se necesita el código fuente ni ninguna otra herramienta para ejecutarlo, por tanto, se puede entregar directamente al cliente.



En el caso de utilizar un intérprete para traducir nuestro programa a código máquina, el paso de fuente a ejecutable es directo mediante el uso del programa interprete. El intérprete traduce y ejecuta directamente cada línea del programa fuente. Al ordenar ejecutar el programa, el programa interprete se carga en memoria RAM junto al código fuente y comienza la tarea de traducir y ejecutar cada línea de código. El inconveniente de este método es que siempre que se quiera ejecutar el programa se necesitará el código fuente (a muchos programadores no les gusta la idea de entregarlo) y el programa traductor (el interprete).



Vamos a analizar de forma más profunda cada una de las fases por las que pasa un programa antes de poder ejecutarse en un ordenador.

5.3.2 Código Fuente.

El código fuente es el programa escrito por el programador, normalmente en algún lenguaje de alto nivel, por ejemplo Java. Está formado por un conjunto de instrucciones que la computadora deberá realizar una vez traducidas a código máquina.

En muy raras ocasiones un programador escribe directamente un programa en ensamblador y menos aún en código máquina. En este caso, el programa en ensamblador o en código máquina sería el código fuente, pero esa práctica no es la normal.

Si hablamos de lenguaje de programación de alto nivel, el 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.

Un aspecto muy importante en esta fase de programación, es la elaboración de un algoritmo.

Algoritmo: conjunto de pasos a seguir para obtener la solución a un problema. Si el algoritmo lo diseñamos en pseudocódigo (un lenguaje de descripción de los pasos a seguir para resolver un problema utilizando una notación muy cercana al lenguaje humano), la codificación posterior a algún lenguaje de programación concreto será más rápida y directa.

Ejemplo: Programa que solicite la edad de una persona y nos diga si es o no adulta.

Pseudocódigo	Java

	import java.util.Scanner;
	public class Main {
	public static void main(String[] args) {

Escribir “Introduce tu edad:”
leer (edad)
 	
Si edad > 17 entonces
  escribir(“Eres adulto.”)
sino
  escribir(“No eres adulto.”)
finSi
	Scanner sc = new Scanner(System.in);
	int edad;
	
	System.out.print("Introduce tu edad: ");
	edad = sc.nextInt();
	
	if((edad > 17)) {
	  System.out.println("Eres adulto.");
	} else {
	  System.out.println("No eres adulto.");
	}
	}
	}

Para obtener el código fuente de una aplicación informática:

1. Se debe partir de las etapas anteriores de análisis y diseño. Es decir, hay que saber que hacer y como hacerlo.
2. Se diseñarán los algoritmos que indican los pasos a seguir para la resolución del problema. Se pueden escribir en pseudocódigo o directamente en el lenguaje de programación que se vaya a utilizar.
3. Se elegirá un lenguaje de programación apropiado para las características del software que se quiere codificar. Hay que recordar que ciertos lenguajes se adaptan mejor a ciertas tareas.
4. Se procederá a la codificación (programación) del algoritmo, en el lenguaje elegido.

La culminación es la obtención del código fuente con la codificación de todos los módulos, funciones, bibliotecas y procedimientos necesarios para crear la aplicación. Recordar que una aplicación suele estar formada por varios módulos que pueden haber sido escritos por distintos programadores.

Como el código fuente no es inteligible por la máquina (salvo que se hubiese programado directamente en código máquina), habrá que traducirlo mediante un traductor (un compilador o un intérprete).

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

- **Código fuente abierto (Open Source)**. Es aquel que está disponible para que cualquier usuario pueda estudiarlo, y en algunos casos incluso modificarlo o reutilizarlo.
- **Código fuente cerrado o privativo**. No tenemos permiso ni siquiera para verlo.

No hay que confundir **código abierto (open source) y software libre (free software)**, son términos relacionados y suelen usarse indistintamente pero son bastante diferentes como se indica en la FSF (Free Software Foundation), organización creada por Richard Stallman en el año 1985. En ambos casos se tiene acceso al código fuente y el programa así licenciado **puede ser gratuito o no**. El código abierto es un movimiento más pragmático, se enfoca más en los beneficios prácticos del acceso al código fuente que en aspectos éticos o de libertad que son tan relevantes en el Software Libre.

Un programa es **software libre** si los usuarios tienen las cuatro libertades esenciales:

1. La libertad de ejecutar el programa como se desea, con cualquier propósito.
2. La libertad de estudiar cómo funciona el programa y cambiarlo si se desea. El acceso al código fuente es una condición necesaria para ello.
3. La libertad de redistribuir copias para ayudar a tu prójimo.
4. La libertad de distribuir copias de sus versiones modificadas a terceros. Esto permite ofrecer a toda la comunidad la oportunidad de beneficiarse de las modificaciones.

Recordar de nuevo que el Software Libre es muy utilizado en los sistemas GNU/Linux, pero también puede incluir código abierto y propietario, dependiendo de la distribución.

Open Source o código abierto, es la expresión con la que se conoce al software distribuido junto al código fuente.

Para pasar de un lenguaje de programación a código máquina se necesita un **programa traductor.** Si yo escribo un programa en lenguaje C++ y no tengo un traductor, no lo podre ejecutar, es como no tener nada, ya que el ordenador es incapaz de reconocer esas instrucciones.

5.3.3 Traductores.

Existen 2 tipos de programas que traducen a código máquina, el único realmente ejecutable por el ordenador:

- **Compiladores:** Pasan todo el código fuente escrito en un lenguaje de programación a otro lenguaje de programación, usualmente a lenguaje máquina, en una sola operación. El resultado es código objeto que tras una última operación de enlazado se obtiene el programa ejecutable, ya no se precisan más traducciones.
- **Interpretes:** Cada línea del código fuente es interpretada a código máquina y ejecutada inmediatamente. Siempre que se quiera ejecutar el programa hay que realizar este proceso de traducción y ejecución.

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

- **Compilación**: La realiza un programa compilador. El proceso de traducción se realiza sobre el código fuente, en un solo paso. El resultado es el código objeto que habrá que enlazar. Hecho esto ya tendremos un programa autónomo totalmente ejecutable para una plataforma concreta.
- **Interpretación**: La realiza un programa interprete. El proceso de traducción del código fuente se realiza línea a línea y se ejecuta simultáneamente. No se crea un código objeto. El proceso de traducción y ejecución hace que el programa se ejecute de forma más lenta, los errores aparecen en el momento en que se ejecute la instrucción problemática. Siempre que quiera ejecutar el

programa necesitaré el código fuente más el interprete, es decir, no tenemos un programa autónomo ejecutable por si solo.

5.3.4 Código objeto.

Es el código resultante de la compilación del código fuente. Recordemos que la compilación es el proceso de traducción que se realiza sobre todo el código fuente, en un solo paso obteniendo el código objeto.

El código objeto puede estar en lenguaje máquina para poder ser ejecutado en un procesador de una plataforma concreta o en bytecode.

Recordar que un programa grande puede constar de varios módulos, cada uno con su código fuente. Cada módulo se puede compilar por separado. Por tanto el código objeto se distribuye en varios archivos que corresponden a cada código fuente compilado.

El caso de **Java** es un poco especial, hablamos de **bytecode,** un código intermedio más abstracto que el código máquina. Habitualmente es tratado como un archivo binario que contiene un programa ejecutable similar a un módulo objeto. La diferencia está en que el módulo objeto si contiene código máquina de la plataforma sobre la que se va a ejecutar, mientras que el bytecode es una especie de código objeto pero para ser usado por una máquina virtual, la cual a su vez interpretará y ejecutará ese código en la máquina real donde está instalada.

¿Qué ventajas tiene el bytecode? Imagina que tenemos un programa escrito en C en un ordenar PC con Windows, se compila, se monta el programa y obtenemos el ejecutable que sólo funcionará en un PC con Windows. Con Java, al compilar obtenemos el bytecode el cual no está diseñado para ninguna plataforma concreta. Si tenemos una máquina virtual Java en un PC con Windows o con GNU/Linux o con Mac, nuestro programa Java funcionará en todos las plataformas. De ahí el conocido lema de Java: "Write once, run anywhere” (escríbelo una vez, ejecútalo en cualquier parte)

El bytecode recibe su nombre porque usualmente cada código de operación tiene una longitud de un byte. Por tanto, cada instrucción tiene un código de operación entre 0 y 255 seguido de parámetros tales como los registros o las direcciones de memoria. Esta sería la descripción de un caso típico, si bien la especificación del bytecode depende ampliamente del lenguaje.

Con todo esto, nos debe quedar claro que el bytecode no es código máquina, pero se le acerca bastante. Pero si no es código máquina, y los ordenadores sólo entienden código máquina, ¿quien traduce al bytecode a código máquina para que se ejecute el programa?

El bytecode es un código intermedio pensado para reducir la dependencia respecto del hardware real de un ordenador. Los programas en bytecode suelen ser interpretados por un intérprete de bytecode (en general llamado máquina virtual, dado que es análogo a un ordenador simulado). Su ventaja es su portabilidad, ya que el mismo bytecode puede ser ejecutado en diferentes ordenadores con distintas plataformas y arquitecturas siempre que tengamos la máquina virtual.

Como el bytecode es en general menos abstracto, más compacto y más orientado a la máquina que un programa pensado para su modificación por humanos, su rendimiento suele ser mejor que el de los lenguajes interpretados, pero inferior a los compilados. A causa de esa mejora en el rendimiento, muchos lenguajes interpretados, de hecho, se compilan para convertirlos en bytecode y después son ejecutados por un intérprete de bytecode. Entre esos lenguajes se encuentran Perl, Gambas, PHP y Python. El código Java se suele trasmitir como bytecode a la máquina receptora, que utiliza un compilador just-in-time para traducir el bytecode en código máquina antes de su ejecución para mejorar la velocidad.

Son asimismo interesantes los denominados p-Codes, similares a bytecodes pero cuyos códigos de operación pueden constar de más de un byte y pueden ser variables en tamaño, como los opcodes de muchas CPUs. Estos códigos trabajan a muy alto nivel, incluyendo instrucciones del estilo de «imprime esta cadena» o «borra la pantalla». Por ejemplo, BASIC utiliza p-Code.

Retornando al código objeto, este aún no es un programa ejecutable aunque es código máquina. Para obtener el programa ejecutable se han de enlazar todos los archivos de código objeto de cada uno de los módulos compilados, junto con algunas librerías utilizadas por los programadores, mediante un programa llamado enlazador (linker).

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

El código objeto es código binario, pero aún no puede ser ejecutado directamente por la computadora.

El siguiente paso para alcanzar un ejecutable es el **enlazado** (link, vinculación), que lo realiza otro programa llamado “linker”, enlazador. Su función es juntar todos los módulos ya compilados obteniendo por fin el programa ejecutable.

Por ejemplo, si mi programa está dividido en 5 módulos, cada uno de ellos se compila por separado, el enlazador los unirá junto a otras librerías del sistema obteniendo finalmente el programa ejecutable. También es típico enlazar funciones que han sido programadas por otros programadores y que nosotros hacemos uso de ellas en nuestro programa. Por ejemplo, mi programa usa una melodía, yo no programé el

reproductor musical, pero lo compré a otro programador que creó una librería para reproducir música. En la fase de compilación yo compilo mi código, en la de enlace yo lo uno todo, lo mio y lo comprado, obteniendo el programa ejecutable.

Para saber más

En el siguiente enlace podrás visitar una página web, que te permitirá aprender más acerca de la generación de códigos objeto:

Generación de código objeto.

5.3.5 Ejecutable.

Si nos hemos decidido por un traductor compilador, tras el proceso de montaje (link) donde se añaden el resto de rutinas necesarias para ejecutar nuestro programa, tendremos el código ejecutable, el programa. En ocasiones consta de un único archivo que puede ser directamente ejecutado por la computadora. No necesita ninguna aplicación externa. Este archivo es ejecutado y controlado por el sistema operativo.

Recordar que en el caso de la compilación, para obtener el archivo ejecutable, habrá que enlazar todos los archivos de código objeto, a través de un software llamado linker (enlazador) y obtener así un archivo que ya sí es ejecutable directamente por la computadora. El código fuente ya no es necesario.

En el caso de usar un traductor intérprete, para ejecutar el programa siempre se necesita el código fuente y el propio interprete, ya que el programa es traducido y ejecutado línea a línea cada vez que se inicia. Se lee una línea del código fuente y se ejecuta, se lee la siguiente línea del código fuente y se ejecuta, ….. Así hasta que termina el programa, en medio del proceso puede aparecer un error de sintaxis o semántico que aborte la ejecución.

Evidentemente la ejecución del código interpretado es más lenta ya que una misma línea puede llegar a ser traducida muchas veces. Con la compilación se traduce una sola vez y tras el enlazado ya tenemos el ejecutable que no precisa de más traducciones y por tanto, no necesita del código fuente.

Muchos programadores o empresas de software venden (o mayoritariamente licencian) el programa ejecutable pero se reservan el código fuente, lo que impide que ese programa sea modificado por otros programadores posteriormente al no tener acceso a ese código fuente. A esto se denomina código fuente cerrado, privativo o propietario, lo contrario es la filosofía del software libre (free software) o la más pragmática del código abierto (open source).

En realidad sería posible modificar el programa ejecutable, pero habría que hacer la denominada ingeniería inversa, lo que lleva muchísimo esfuerzo y el trabajo con lenguaje ensamblador. Por ejemplo, para hacer trampas en un juego, se puede modificar el ejecutable para que no muera el personaje. Con el código fuente sería mucho más fácil, pero pocas veces se consigue el código fuente de los juegos más populares.

5.3.6 Frameworks

Es una estructura de ayuda al programador, en base a la cual podemos desarrollar proyectos sin partir desde cero. Imaginemos que cogemos a varios programadores para realizar un proyecto. Si cada uno utiliza sus propias herramientas, su propio IDE, su forma personal de programar, etc. al final, el resultado es que va a ser muy complicada la colaboración, la comunicación y el intercambio de información entre ellos. Pues bien, la solución es un Framework, un entorno común para todos, que permita un desarrollo de aplicaciones más rápido, fiable, homogéneo y donde la reutilización de material será más efectiva.

Se entiende por "framework" (marco de referencia, infraestructura) un conjunto estandarizado de conceptos, prácticas y criterios para enfocar un tipo de problemática particular que sirve como referencia, para enfrentar y resolver nuevos problemas de índole similar.

En el desarrollo de software, un framework o infraestructura digital, es una estructura conceptual y tecnológica de soporte definido, normalmente con artefactos o módulos de software concretos, que puede servir de base para la organización y desarrollo de software. Típicamente, puede incluir soporte de programas, bibliotecas, y un lenguaje interpretado, entre otras herramientas, para así ayudar a desarrollar y unir los diferentes componentes de un proyecto.

Con el uso de un framework la comunicación y la colaboración son más efectivas.

- **Ventajas** de utilizar un framework:
    - Desarrollo más rápido del software.
    - Reutilización de partes de código para otras aplicaciones.
    - Diseño uniforme del software.
    - Suele favorecer la portabilidad de aplicaciones si se trabaja con maquinas virtuales.
- **Inconvenientes**:
    - Gran dependencia del código respecto al framework utilizado (sin cambiamos de framework, es posible que haya que reescribir parte de la aplicación).
    - Se necesita un tiempo de adaptación.
    - Para algunos proyectos puede implicar ciertas limitaciones impuestas por el propio Framework.
    - Consume bastantes recursos del sistema.

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. Es un componente que se instala sobre el sistema operativo.
- **Spring MVC** de Java. Son conjuntos de bibliotecas (API's) para el desarrollo y ejecución de aplicaciones.

Para saber más

El uso creciente de frameworks hace que tengamos que estar reciclándonos constantemente.

Frameworks

5.3.7 Máquinas Virtuales.

Una máquina virtual es un tipo especial de software que simula el funcionamiento de un equipo (ordenador, teléfono, consola de videojuegos …) dentro de otro equipo.

Existen dos tipos de máquinas virtuales diferenciadas por su funcionalidad:

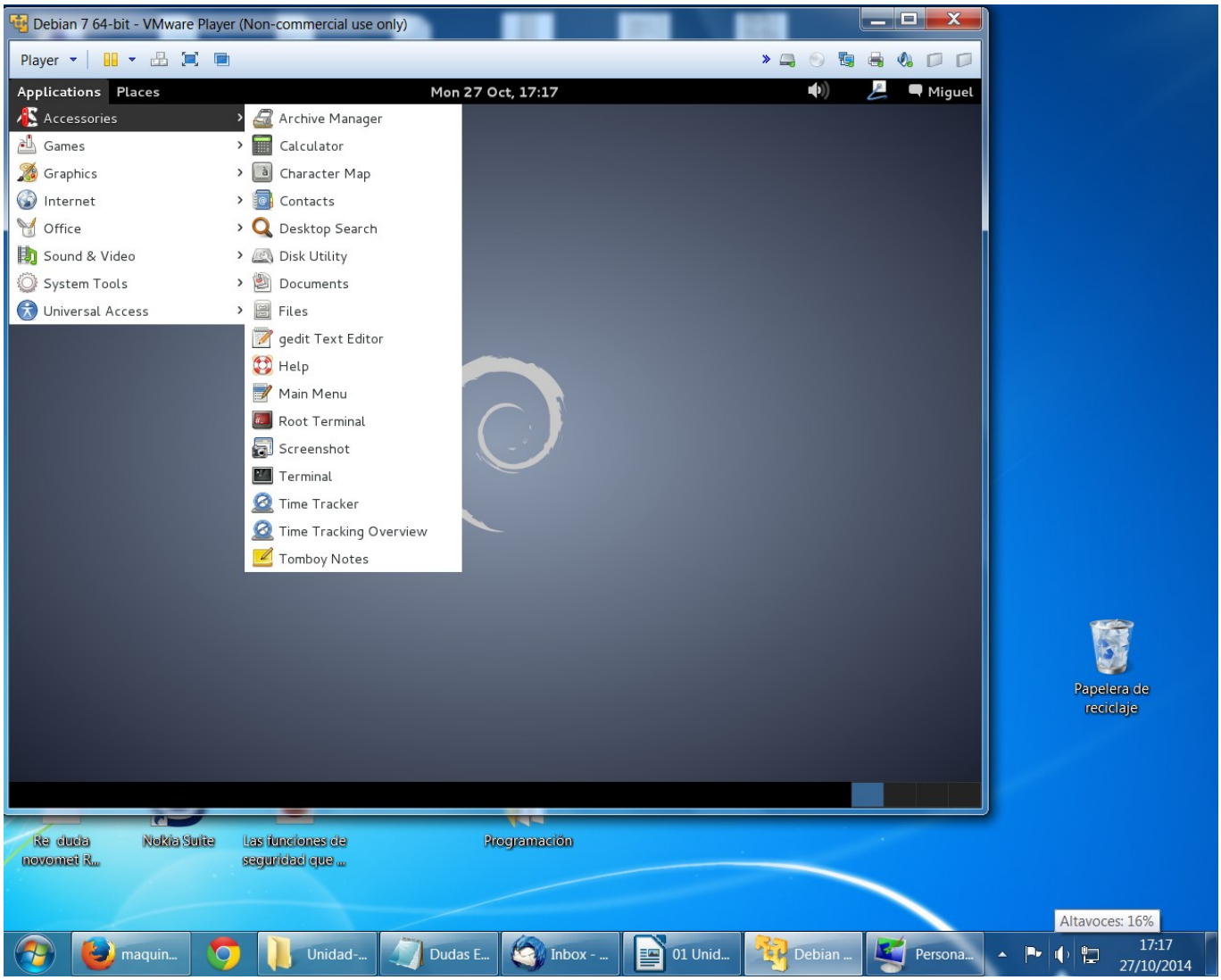
- **Las de sistema**: Emula a un ordenador completo. Es un software que puede hacerse pasar por otro dispositivo, como un ordenador PC, de tal modo que incluso puedes ejecutar otro sistema operativo totalmente distinto en su interior. La máquina virtual tiene sus componentes virtuales como por ejemplo discos duros, memoria RAM, tarjeta gráfica y demás componentes de hardware. Para el sistema operativo que se ejecuta dentro de la máquina virtual toda esta emulación es transparente e invisible, es decir, ese sistema operativo cree realmente estar en una ordenador de verdad. Programas como VMWare o Oracle VirtualBox permiten realizar esta tarea.
- **Las de proceso**: Son más simples, en vez de emular un ordenador completo, permiten ejecutar un programa en su entorno de ejecución. Esto es lo que pasa cada vez que ejecutas una aplicación basada en Java o en .NET Framework. Es de utilidad a la hora de desarrollar aplicaciones para varias plataformas, pues en vez de tener que programar específicamente para cada sistema, el entorno de ejecución (la máquina virtual) es el que se encarga de lidiar con el sistema operativo de la máquina real.

Las máquinas virtuales más populares en general son las de sistema, pero para los programadores las de proceso son muy utilizadas.

Una característica esencial de las máquinas virtuales es que los procesos (programas) que se ejecutan dentro de ellas, están limitados por los recursos y abstracciones proporcionados por las propias máquinas (además de los límites de la máquina real). Estos procesos no pueden escaparse de esa "computadora virtual". Por tanto es una buena plataforma para probar programas “sospechos” de contener virus, troyanos u otro tipos de malware, sin afectar a la máquina real.

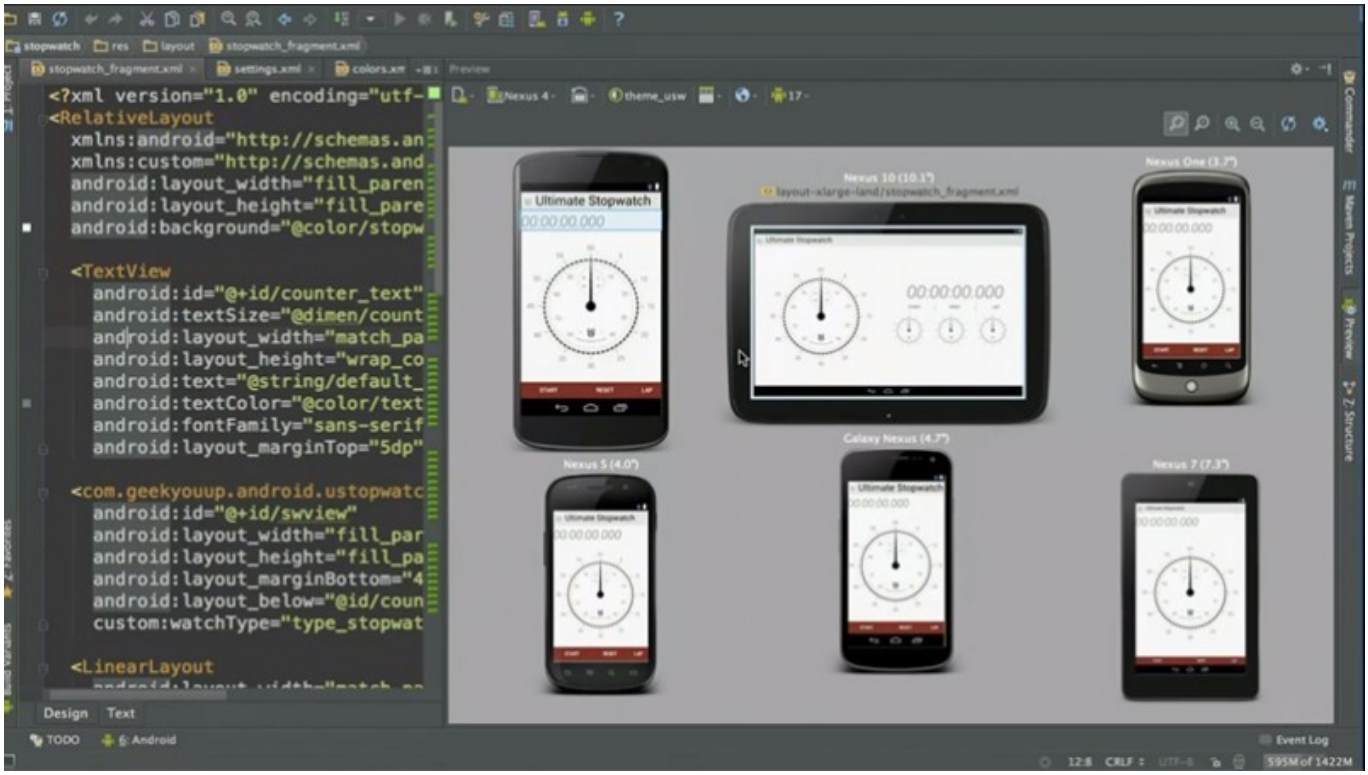
Con el uso de máquinas virtuales podremos desarrollar y ejecutar una aplicación sobre cualquier equipo (siempre que tenga instalada la máquina virtual), con cierta independencia de las características concretas de los componentes físicos instalados. Esto, en muchos casos, garantiza la potabilidad de las aplicaciones. Evidentemente no podemos tener en la máquina virtual más recursos que en la real.

En la siguiente imagen vemos un ordenador ejecutando un sistema operativo Windows. A la vez, dentro de una máquina virtual se está ejecutando el sistema operativo Debian Linux mediante el programa VMWare.



Hay muchos usos para las máquinas virtuales, por ejemplo, para hacer programas para una teléfono móvil, no se programa desde el teléfono, se usa un PC con un software simulador del teléfono (máquina virtual), desde el cual se hacen las pruebas del programa. Lo mismo ocurre para programar un juego para una videoconsola, todo el programa se desarrolla en un ordenador, el cual mediante una máquina virtual puede simular el funcionamiento de la videoconsola. Si el hardware de la videoconsola es demasiado complejo para simular, el resultado puede ser una ejecución demasiado lenta, en esos casos, una vez realizado el programa, se probaría directamente en la videoconsola.

En la siguiente imagen, se está programando desde un ordenador un programa para distintos tipos de teléfonos y tabletas usando máquinas virtuales de los mismos.

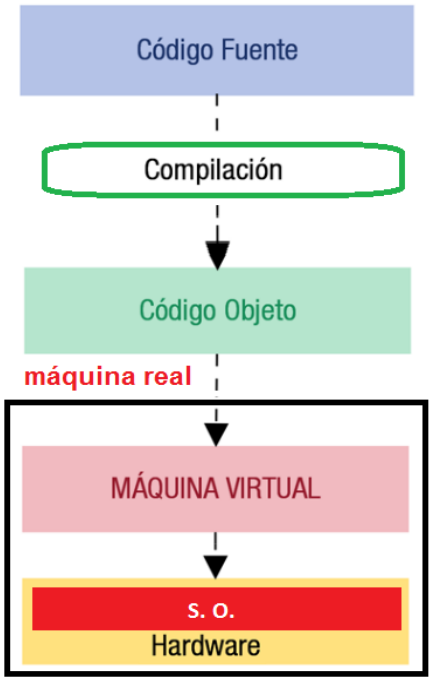


5.3.7.1 La máquina virtual de Java (Java Virtual Machine, JVM)

JVM es de proceso nativo, es decir, ejecutable en una plataforma específica (Windows, Android, etc) , capaz de interpretar y ejecutar instrucciones expresadas en un código binario especial (bytecode Java), el cual es generado por el compilador del lenguaje Java.ç

Los programas interpretados o compilados tienen ventajas e inconvenientes. En un intento de combinar lo mejor de ambos mundos, durante la década de los 90 surge con fuerza el enfoque de máquina virtual. Un intento por tener un sistema que permita hacer un programa y que funcione en cualquier tipo de ordenador y Sistema Operativo. Los principales lenguajes abanderados de esta tecnología son, por un lado, el lenguaje Java, y por otro, los lenguajes de la plataforma .NET, en especial C#.

La filosofía de la máquina virtual es la siguiente: el programa (código fuente) se compila y se genera una especie de ejecutable, un código máquina dirigido a una máquina imaginaria, con una CPU imaginaria. A esta especie de código máquina se le denomina código intermedio, o a veces también lenguaje intermedio, “p-code”, o “bytecode”.



Como esa máquina imaginaria no existe, para poder ejecutar ese ejecutable en una máquina real, se construye un intérprete. Este intérprete es capaz de leer cada una de las instrucciones de código máquina imaginario y ejecutarlas en la máquina real. A este intérprete se le denomina el intérprete de la máquina virtual.

El código binario generado por el compilador de Java (**bytecode**) es un verdadero código máquina de bajo nivel, que funciona en la **JVM (Java Virtual Machine)**. Podría ser incluso ejecutado en un microprocesador físico si algún día se fabricase.

La JVM es una de las piezas fundamentales de la plataforma Java. Se sitúa en un nivel superior al Sistema Operativo (SO) y el hardware del equipo real donde se ejecuta. Cuando se escribe una aplicación Java, se hace pensando que será ejecutada en una máquina virtual Java, siendo ésta la que en última instancia convierte el bytecode a código nativo del dispositivo final. La idea es programar una vez y ejecutar en cualquier sitio... siempre que tengamos la JVM.

La gran ventaja de la máquina virtual Java es aportar portabilidad al lenguaje. Su creador, Sun Microsystems (ahora pertenece a Oracle), ha implementado diferentes máquinas virtuales Java para varias arquitecturas. De esta forma un programa Java escrito en Windows puede ser interpretado en un entorno Linux o en un teléfono Android. Tan solo es necesario disponer de la máquina virtual (JVM). De ahí el famoso axioma de Java: **"escríbelo una vez, ejecútalo en cualquier parte" (“Write once, run anywhere")**.

La máquina virtual de Java (JVM) puede estar implementada en software, hardware, en una herramienta de desarrollo o incluso en un navegador Web.

Cuando un programa realizado en Java (código fuente) se compila, se obtiene código objeto (bytecode, código intermedio). Para ejecutarlo en cualquier tipo de 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 (máquina real); 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 real.

La Máquina Virtual Java verifica todo el bytecode antes de ejecutarlo, además protege direcciones de memoria para no interferir con la máquina real.

Resumen comparativa ente compiladores, interpretes y máquinas virtuales:

Compilador	**Interprete**	**Máquina virtual Java (JVM)**
Genera un ejecutable.	No genera un ejecutable.	Genera un ejecutable dirigido a una CPU imaginaria.
El proceso de traducción se realiza una sola vez.	El proceso de traducción se realiza en cada ejecución.	La traducción a bytecode se realiza una sola vez. De bytecode a código máquina se interpreta en cada ejecución.
La ejecución es muy rápida debido a que el programa ya ha sido traducido a código máquina.	La ejecución es más lenta, ya que para cada línea del programa es necesario realizar la traducción y posterior ejecución.	La ejecución no es tan rápida como en la compilación pero más rápida que en la intepretación.
El ejecutable va dirigido a una CPU y un SO concreto. Es complicado portarlo a otra plataforma. Se puede recompilar a otra plataforma, pero suele plantear problemas. Los programas están muy ligados a la plataforma de destino.	No hay ejecutable, así que si existe un intérprete para una plataforma concreta, el programa se podrá ejecutar en ambas. Suelen ser más portables que los compilados.	El ejecutable va dirigido a una CPU imaginaria. Se puede transportar a otra plataforma para la cual exista una JVM.
Ofrecen al programador mecanismos más potentes y flexibles, a costa de una mayor ligazón a la plataforma.	No son ser muy dependientes de la plataforma de destino, pero en contrapartida suelen ser menos flexibles y potentes que los compilados.	La plataforma de destino es virtual, así pues, los programas son dependientes de esa plataforma virtual. La maquina virtual si tiene dependencia de la real.

Una vez compilado el programa, el código fuente ya no es necesario para ejecutarlo, así que puede permanecer en secreto si se desea.	El código fuente es necesario en cada ejecución, así que no puede permanecer en secreto.	El código fuente una vez compilado ya no es necesario para la ejecución; el código intermedio si.
Los errores sintácticos se detectan durante la compilación. Si el fuente contiene errores sintácticos, el compilador no producirá un ejecutable.	Los errores sintácticos se detectan durante la ejecución, ya que traducción y ejecución se hacen simultáneamente. Algún error sintáctico podría quedar enmascarado, si para una ejecución concreta no es necesario traducir la línea que lo contiene. (Algunos intérpretes son capaces de evitar esto)	Los errores sintácticos se detectan durante la compilación.
Un error grave en un programa compilado puede afectar seriamente a la estabilidad de la plataforma, llegando incluso a colgar el equipo.	Un programa interpretado con un comportamiento torpe normalmente puede ser interrumpido sin dificultad, ya que su ejecución está bajo el control del intérprete, y no solo del sistema operativo.	Un programa con un comportamiento torpe es ejecutado sobre la máquina virtual, que tiene un control absoluto sobre él, con lo que no se suele comprometer la estabilidad de la plataforma real.

5.3.7.2 Entornos de Ejecución.

Sabemos que una máquina virtual se ejecuta sobre una máquina real con su sistema operativo. Los programas realizados en Java se compilan para la máquina virtual (JVM) y se ejecutan en ella, aunque es verdad que a su vez la máquina virtual se ejecuta sobre la real. Un entorno de ejecución es un servicio de máquina virtual que sirve como base software para la ejecución de programas sin interferir sobre el resto de programas que se están ejecutando en la máquina real. En ocasiones el entorno de ejecución pertenece al propio sistema operativo, pero también se puede instalar como software independiente que funcionará entre la aplicación y el sistema operativo real. Resumiendo, un entorno de ejecución es un conjunto de utilidades que permiten la ejecución de programas.

Se denomina RUNTIME, al intervalo de tiempo en el que un programa de computadora se está ejecutando en un sistema operativo. Este tiempo se inicia con la puesta en memoria principal del programa. El intervalo finaliza en el momento en que el programa envía al sistema operativo la señal de terminación, sea ésta una terminación normal, en que el programa tuvo la posibilidad de concluir sus

instrucciones satisfactoriamente, o una terminación anormal, en el que el programa produjo algún error y el sistema debió forzar su finalización.

RECUERDA: Un programa en ejecución está cargado en la memoria principal (RAM).

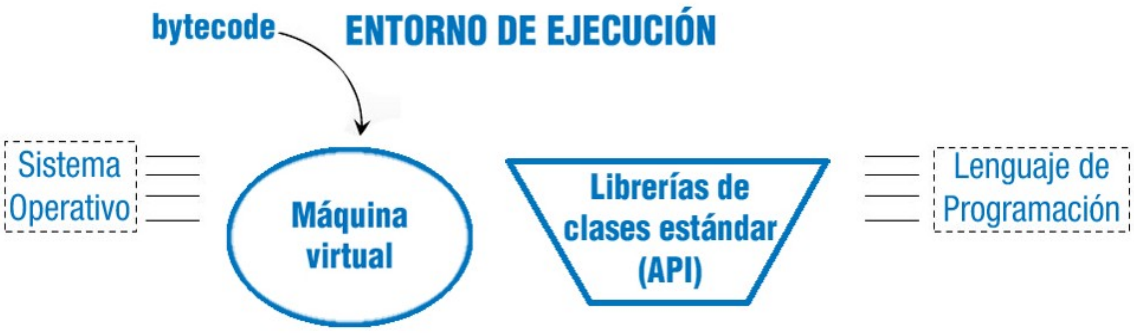
Para el funcionamiento de un programa, los entornos de ejecución se encargarán de:

- Reservar parte de la memoria principal RAM disponible en el sistema.
- Enlazar los archivos del programa (normalmente formado por varios subprogramas) con las bibliotecas existentes (vínculos dinámicos). Las bibliotecas son subprogramas ya compilados realizados en muchos casos por otros desarrolladores.
- 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.



Para desarrollar aplicaciones necesitamos algo más. Ese "algo más" se llama **entorno de desarrollo**.

En el caso de Java:

- **JRE** (Java Runtime Environment) es necesario para ejecutar aplicaciones Java como por ejemplo Eclipse. Un usuario normal (no programador) solo necesita el JRE para ejecutar aplicaciones Java. Está compuesto de un conjunto de utilidades que permitirá la ejecución de programas Java sobre distintos tipos de plataformas. El JRE está formado por:
  - Una Máquina virtual Java (JVM), que es el programa que interpreta el código de la aplicación escrito en Java.
  - Bibliotecas de clase estándar que implementan la interfaz de programación de aplicaciones (API) de Java.

  Tanto la JVM como la API de Java son consistentes entre sí, por ello son distribuidas conjuntamente.
- **JDK** (Java Development Kit) es un software que provee herramientas de desarrollo para la creación de programas en Java. Un programador Java necesita el JDK (incluye el JRE) para crear nuevos programas.

5.4 Pruebas.

Mientras se va desarrollando un programa se van realizando pruebas de los distintos módulos, pero la verdadera prueba de fuego es cuando se supone que ya está todo listo y se integran todos las partes.

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 uno a uno los diferentes módulos de que consta el programa y comprobar su funcionamiento por separado, de manera independiente. JUnit es el entorno de pruebas para Java.

- **Pruebas de integración**

Se realizan una vez que se han realizado con éxito las pruebas unitarias y consistirán en comprobar el funcionamiento del sistema completo con todas sus partes interrelacionadas.

La prueba final antes del lanzamiento de un programa se denomina comúnmente **Beta Test**, ésta prueba se realiza sobre el entorno de producción donde el software va a ser utilizado por el cliente (a ser posible, en los equipos del cliente y bajo un funcionamiento normal de su empresa).

Para saber más

Puedes visitar la siguiente página web, donde se detallan los tipos de pruebas que suelen hacer al software y la función de cada una.

Prueba de software

5.5 Documentación.

Todas las etapas en el desarrollo de software deben quedar perfectamente documentadas. La documentación generada debe ser guardada para su posible uso más adelante en el caso de tener que realizar modificaciones o solucionar algún error que quedó oculto en la fase de pruebas.

¿Por qué hay que documentar todas las fases del proyecto?

Para dar toda la información a los usuarios de nuestro software y poder acometer futuras revisiones del proyecto. Por tanto podemos decir que básicamente hay 2 tipos de documentación, la dirigida al usuario y la dirigida a los desarrolladores.

Tenemos que ir documentando el proyecto en todas las fases del mismo, para pasar de una a otra de forma clara y definida. Una correcta documentación permitirá la reutilización de parte de los programas en otras aplicaciones, siempre y cuando se desarrollen con diseño modular.

Distinguimos tres grandes documentos en el desarrollo de software: La guía técnica, la guía de uso y la guía de instalación.

Documentos a elaborar en el proceso de desarrollo de software

	GUÍA TÉCNICA	GUÍA DE USO	GUÍA DE INSTALACIÓN
Quedan reflejados:	• El diseño de la aplicación. • La codificación de los programas. • Las pruebas realizadas.	• 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.	Toda la información necesaria para: • Puesta en marcha. • Explotación. • Seguridad del sistema.
¿A quién va dirigido?	Al personal técnico en informática (analistas y programadores).	A los usuarios que van a usar la aplicación (clientes).	Al personal informático responsable de la instalación, en colaboración con los usuarios que van a usar la aplicación (clientes).
¿Cuál es su objetivo?	Facilitar un correcto desarrollo, realizar correcciones en los programas y permitir un mantenimiento futuro.	Dar a los usuarios finales toda la información necesaria para utilizar la aplicación.	Dar toda la información necesaria para garantizar que la implantación de la aplicación se realice de forma segura, confiable y precisa.

Un detalle muy importante a cerca de la calidad de la documentación, es que no siempre son los mismos desarrolladores los que van a tener que realizar algún cambio o modificación.

5.6 Explotación.

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 en su versión final y comienzan a utilizarla en su lugar de trabajo con datos reales.

La explotación conlleva 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 clientes estén presentes en este momento e irles comentando cómo se va planteando la instalación.

Una vez instalada la aplicación, 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 lo hagan aquellos que la han fabricado).

Una vez se ha configurado el programa, 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 presentar el producto al cliente: será el momento crítico del proyecto. No conviene dar una mala imagen por falta de preparación del producto.

Hay que tener especial cuidado cuando el cliente pasa de utilizar un sistema antiguo al actual, ya que se pueden llegar a perder datos en ese cambio. En muchos casos se opta por tener los 2 sistemas funcionando hasta que se comprueba que el nuevo es plenamente funcional. Evidentemente esta es una situación que no gusta a los usuarios pues implica mucho más trabajo, pero es más segura.

No es deseable ver noticias como la referida a la implantación del software denominado “Millenium” en el nuevo Hospital Universitario Central de Asturias (HUCA). (extraída de http://www.asturiashoy.es)

‘Millenium’, el software que puede llevar el caos al HUCA

El nuevo sistema costó 17 millones de euros y los médicos apenas tienen experiencia en su uso, advierte Albano Longo

A.F./E.P.

OVIEDO, 31 May. 2014.–

El programa informático ‘Millenium’ amenaza el orden del nuevo Hospital Central Universitario de Asturias (HUCA). Esta advertencia la lanzó hoy Albano Longo, diputado de Foro en la Junta General de Asturias, quien sostiene que la Consejería de Sanidad va a generar el “caos” en el recién estrenado centro sanitario de La Cadellada, por el empeño que tienen los gestores de este Hospital en imponer este sistema para el control de los pacientes.

5.7 Mantenimiento.

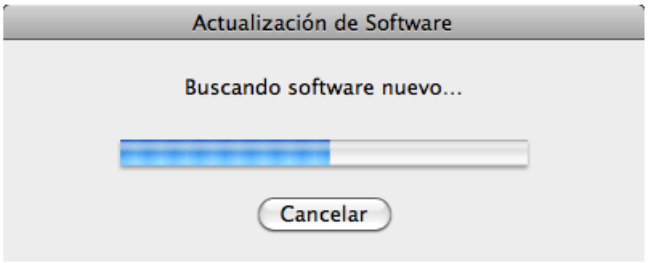
Sería lógico pensar que con la entrega de nuestra aplicación (la instalación y configuración de nuestro proyecto en los equipos del cliente) hemos terminado nuestro trabajo. Pero la realidad es otra, con el uso es normal del programa es habitual que aparezcan fallos, cosas mejorables o modificaciones futuras por cambios de legislación u otros factores que no estaban previstos durante el desarrollo. En cualquier otro sector laboral esto no es muy frecuente, pero en el caso de la construcción de software si.

La etapa de mantenimiento es la más larga de todo el ciclo de vida del software, ya que cubre el tiempo que transcurre desde la instalación del programa hasta que se deja de utilizar.

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 y afrontar situaciones nuevas que no existían cuando el software se construyó.

Pero aparte de las mejores, siempre surgen errores que habrá que ir corrigiendo en las nuevas versiones del producto.

Por todo ello, normalmente 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 tendrá en el futuro nuevas necesidades. Por tanto, serán necesarias modificaciones, expansiones o eliminaciones de código.
- **Adaptativos**: Modificaciones, actualizaciones... para adaptarse a las nuevas tendencias del mercado, a nuevos componentes hardware, etc.
- **Correctivos**: La aplicación tendrá errores en el futuro (sería utópico pensar lo contrario).