Contenido

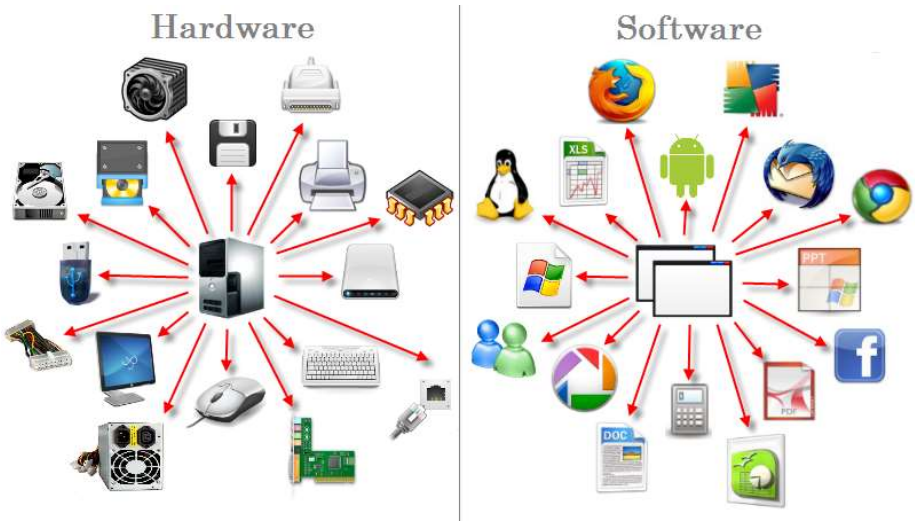
1. Objetivo

- ✓ Es preciso comprender los principios básicos del desarrollo de software. En esta unidad se tratarán aspectos como lenguajes de programación y sus características, así como las fases de desarrollo de una aplicación.
- ✓ También es importante que entiendas la diferencia entre compiladores, intérpretes y máquinas virtuales, pues los lenguajes de programación funcionarán y tendrán ventajas y desventajas dependiendo de la tecnología subyacente.

2. INTRODUCCIÓN

Un ordenador se compone de dos partes: hardware y software:

- **Hardware**: es la parte física, los componentes físicos que se pueden ver y tocar: monitor, teclado, ratón, disco duro, microprocesador, etc.
- **Software**: es la parte lógica, no se puede "tocar", es el conjunto y aplicaciones que actúan sobre el hardware del ordenador facilitando al usuario la realización de diferentes tareas.



3. SOFTWARE

Hay varias definiciones para el término *software*:

- **Según la RAE *(Real Academia Española)***: "*Software es el conjunto de programas, instrucciones y reglas informáticas para ejecutar ciertas tareas en un ordenador".*

- **Según el IEEE (*Intitute of Electrical and Electronic Engineer*):** *"Software es el conjunto de programas de cómputo, procedimientos, reglas, documentación y datos asociados que forman parte de las operaciones de un sistema de computación"*

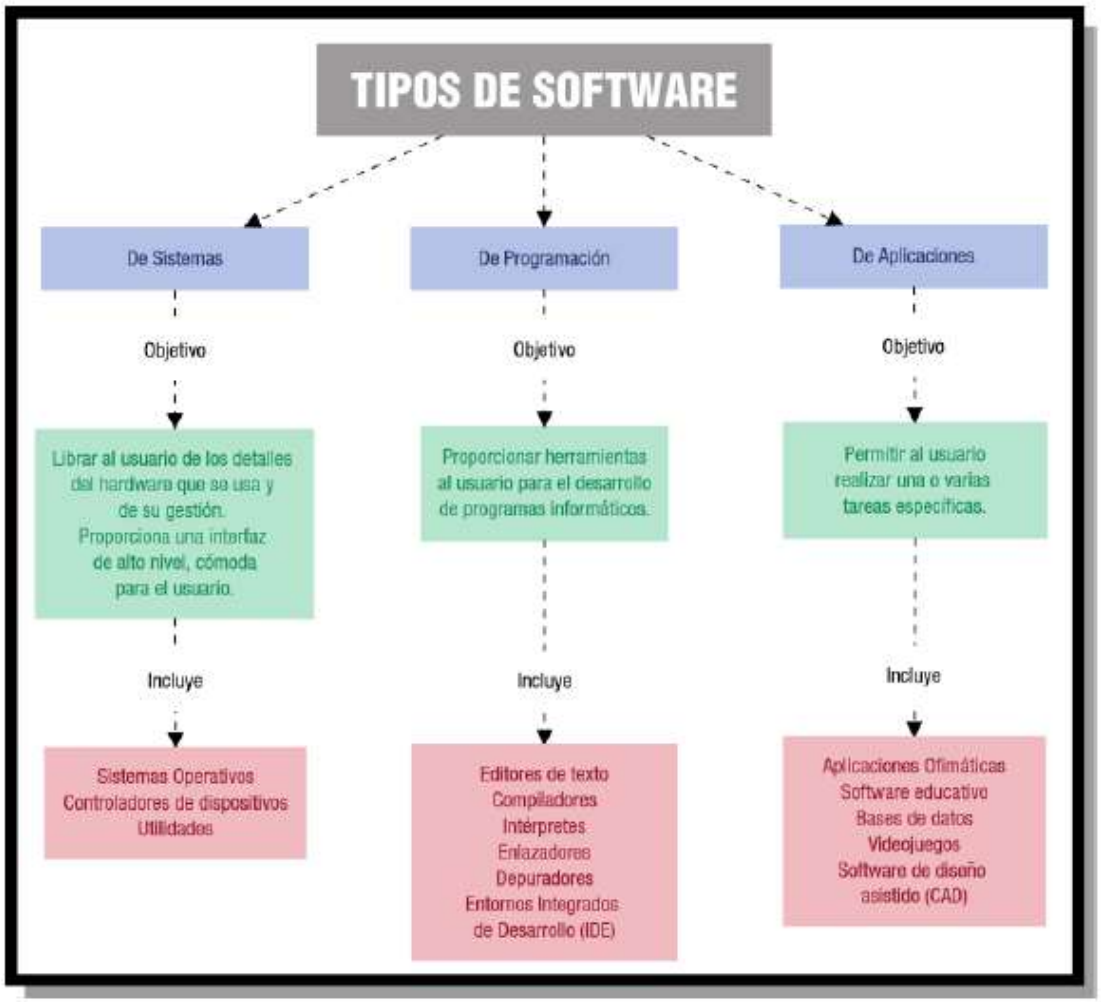
Resumiendo: *Software es todo lo referente a programas y datos almacenados en un ordenador para ejecutar tareas*.

2.1 Clasificación del Software

En función del tipo de tarea que realiza podemos distinguir:

- ✓ ***Software de Sistema:*** es el software base que permite que nuestro hardware funcione, básicamente son el Sistema Operativo y los controladores de dispositivo, por tanto, es lo primero que teneos que instalar y configurar en un ordenador.
- ✓ ***Software de Aplicación***: es el conjunto de programas que nos ayudan a realizar tareas específicas, como son aplicaciones ofimáticas, reproductores de vídeo, programas de contabilidad, etc.
- ✓ ***Software de Programación (ó Desarrollo)***: conjunto de herramientas que permiten desarrollar programas informáticos.

  Un software a medida es una o varias aplicaciones realizadas según los requerimientos e instrucciones de una empresa u organismo. Dichos programas se amoldan o adecuan a la actividad desarrollada y son diseñados a la medida del organismo, su forma de trabajar, sus necesidades.



✓ Ejemplos de tipo de software:

Software de sistema.	Software de Aplicación	Software de Programación
Sistemas operativos (Linux, Windows, Solaris, etc.)	Aplicaciones de Sistema de control y automatización industrial (Cibermatrix)	Editores de texto (Pascal, Edit de MS-DOS)
Controladores de dispositivo ó drivers (driver de tarjeta de video o audio)	Software educativo (Clic, GCompris, PLATO)	Enlazadores (Oracle, Circle)
Herramientas de diagnóstico (Everest, Sonia, etc.)	Software de Control Numérico ó CAM (CAM350, Gerbtool)	Entornos de Desarrollo Integrados ó IDE (NetBean, Visual Basic)
Utilidades (Accesorios de Windows, símbolo de sistema)	Software de Cálculo Numérico (Excel, COI, NOI, SAE)	Entornos de Desarrollo Integrados ó IDE (NetBean, Visual Basic)

2.2 Licencias de Software

La licencia de software es una especie de contrato, (entre el desarrollador de un software y el usuario) en donde se especifican todas las normas y cláusulas que rigen el uso de un determinado programa, principalmente se estipulan los alcances de uso, instalación, reproducción, copia y distribución de estos productos.

Las licencias de uso de software generalmente son de alguno de estos tipos:

- **Licencia propietaria**. Uso en un ordenador por el pago de un precio.
- **Shareware**. Uso limitado en tiempo o capacidades, después pagar un precio.
- **Freeware**. Usar y copiar ilimitado, precio es cero.
- **Software libre**. Usar, copiar, estudiar, modificar, redistribuir. Código fuente incluido.

Es posible dividir las licencias de **software libre** en dos grandes familias. Una de ellas está compuesta por las licencias que no imponen condiciones especiales, sólo especifican que el software se puede redistribuir o modificar. Estas son las llamadas **licencias permisivas**. La otra familia, denominadas licencias robustas o licencias ***copyleft***, imponen condiciones en caso de que se quiera redistribuir el software, condiciones que van en la línea de forzar a que se sigan cumpliendo las condiciones de la licencia después de la primera redistribución.

Mientras que el primer grupo hace énfasis en la libertad de quien recibe un programa, ya que le permite hacer casi lo que quiera con él (en términos de las sucesivas redistribuciones), el segundo obliga a que las modificaciones y redistribuciones respeten los términos de la licencia original.

Para saber más sobre tipos de licencias libres:
http://www.monografias.com/trabajos55/licencias-de-software/licencias-de-software2.shtml

http://www.gnu.org/licenses/license-list.html

4. PROGRAMAS DE ORDENADOR

Un **ordenador** es una máquina digital y sincrónica con cierta capacidad de cálculo numérico y controlado por un programa almacenado en memoria con capacidad para comunicarse con el exterior.

Una **máquina digital** es un ordenador que maneja señales eléctricas que representan los dos únicos estados posibles de información (0 y 1).

Es una **máquina sincrónica** porque todas las operaciones se realizan coordinadas por un único reloj central que envía pulsos a los elementos del ordenador para que operen a su debido tiempo.

Un **programa de ordenador** (o programa informático) es un conjunto de instrucciones ordenadas escritas en un lenguaje de programación que aplicadas sobre un conjunto de datos resuelven un problema, o parte del mismo.

Véase el típico programa Hola mundo desarrolla en lenguaje C

```
//**Código:Programa Holamundo****
/* Programa holamundo.c */
#include <stdio.h>
main(){
   printf("Hola Mundo");
}
```

El texto anterior representa un ejemplo de los sería un programa. Se compone de 6 líneas, que van a comentarse a continuación:

```
/* Programa holamundo.c */
Esta línea no realiza ninguna función solo dice cuál es el nombre del pro-
   grama.
#include <stdio.h>
Esta línea es necesaria si va a sacarse algo por pantalla.
main ()
Esta línea indica que esto es lo primero que va ejecutar el programa (lo
   contenido entre { y }).{
   printf ("Hola Mundo");
   Esta línea muestra las palabras Hola mundo por pantalla.
}
```

En resumen, este programa mostrará las palabras ***Hola mundo*** por pantalla.

Una **tarea** es el objetivo o el problema que queremos resolver con nuestro programa.

Un **dato** es una representación de algún objeto del mundo real, relacionado con la tarea que debe resolver el programa. Por ejemplo, un nombre, una fecha, una edad, un precio.

5. RESOLUCIÓN DE PROBLEMAS

5.1 Ingeniería del software

La Ingeniería del Software es una disciplina o área de la Informática que ofrece métodos y técnicas para desarrollar y mantener software de calidad que resuelven problemas de todo tipo. Podemos decir que es la ciencia y el arte de especificar, diseñar y desarrollar programas, documentación y procedimientos operativos.

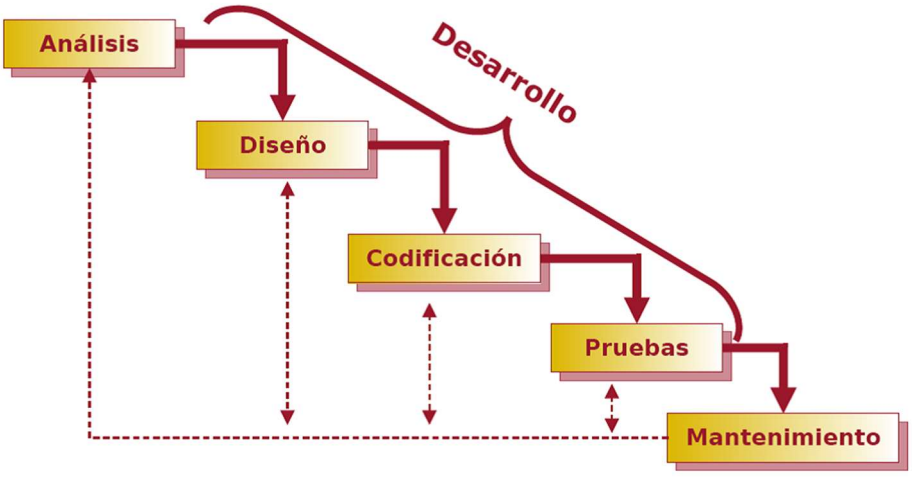
5.2 Ciclo de vida del software

El ciclo de vida de un producto software es el **conjunto de actividades, procesos y tareas** involucradas en el desarrollo, la explotación y el mantenimiento de un producto software.

El ciclo de vida del software comprende el periodo que transcurre desde que el producto es concebido hasta que deja de estar disponible o es retirado. Normalmente, se divide en etapas y en cada etapa se realizarán una serie de tareas.

Usualmente se consideran las siguientes etapas:

1. **Análisis**: Se construye un modelo de los requisitos. En esta etapa se debe entender y comprender de forma detallada el problema que se va a resolver. Es muy importante producir en esta etapa una documentación entendible, completa y fácil de verificar y modificar.
2. **Diseño**: En esta etapa ya se sabe qué es lo que hay que hacer, ahora hay que definir cómo se va a resolver el problema. Se establecen las estructuras de datos, la arquitectura del software, la interfaz de usuario y los procedimientos. Por ejemplo, hay que elegir el lenguaje de programación, seleccionar el Sistema Gestor de Bases de Datos, etc.
3. **Codificación**: se traduce lo descrito en el diseño a una forma legible por el ordenador: código ejecutable.
4. **Pruebas**: se comprueba que se cumplen criterios de corrección y calidad. Las pruebas deben garantizar el correcto funcionamiento del sistema.
5. **Mantenimiento**: esta etapa tiene lugar después de la entrega del software al cliente. En ella se asegura que el sistema se puede adaptar a los cambios, bien porque se han encontrado errores, porque es necesario adaptarse al entorno (por ejemplo, cambio de SO) o porque el cliente requiere mejoras.

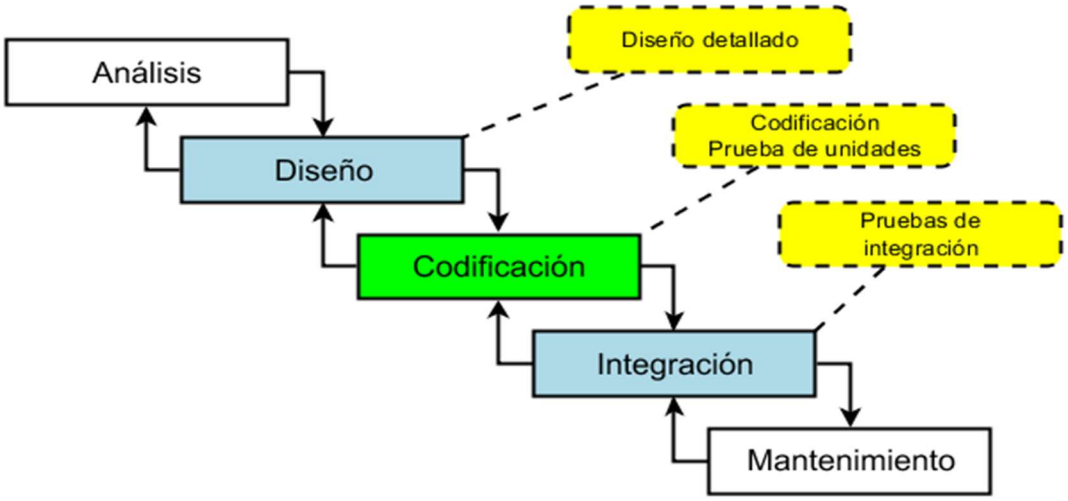


Cada etapa tiene como entrada uno o varios documentos procedentes de las etapas anteriores y produce otros documentos de salida, por ello una tarea importante a realizar en cada etapa es **la documentación**.

Existen varios modelos de ciclo de vida, es importante tener en cuenta las características del problema a resolver (proyecto software) para elegir un modelo u otro. Los modelos más importantes son el ***Modelo en Cascada con Realimentación*** y ***el Modelo Evolutivo.***

A) Modelo en Cascada con Realimentación

El ciclo de vida lo componen las siguientes fases:



- **Análisis del problema**: Consiste en ver y comprender que tarea se quiere resolver. Es imprescindible partir de una especificación de requisitos lo más exacta y detallada posible. El proceso de comprensión y simplificación del mundo real se llama análisis del problema y lo que se obtiene tras el análisis es el modelo.
- **Diseño de una solución**: Consiste en cómo se va a resolver el problema, suele consistir en dividir el problema principal en problemas más sencillos cuya combinación resuelve la tarea final.
- **Especificación de módulos**: Consiste en, para cada subproblema ideado, diseñar una solución que lo resuelva lo más eficiente posible, esto se realiza con un algoritmo.
- **Codificación**: Una vez definidos los algoritmos, los traducimos al lenguaje de programación que tengamos que usar. Un lenguaje de programación es un conjunto de símbolos y reglas sintácticas especialmente diseñado para transmitir ordenes al ordenador (C, Java, Pascal…).
- **Pruebas**: Sirven para corregir posibles errores, tendremos dos tipos de errores:

- **Sintácticos**: Producidos por un mal uso del lenguaje.
- **Semánticos**: Lo que está equivocado es la solución que yo he ideado.

➢ **Mantenimiento:** Hay tres tipos:

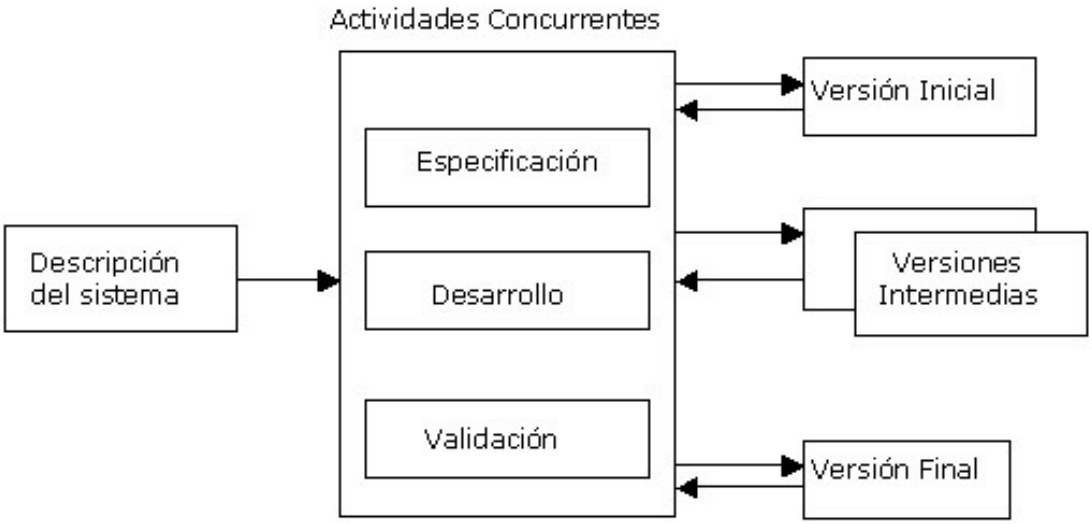
- **Correctivo:** Sirve para corregir posibles errores o fallos del programa.
- **Perfectivo:** Se usa para perfeccionarlo**.**
- **Adaptativo:** Sirve para adaptarlo a nuevas situaciones**.**

B) Modelo Evolutivo

El SW evoluciona con el tiempo, es normal que los requisitos del usuario y del producto cambien conforma se desarrolla el mismo. La competencia en el mercado del sw es tan grande que las empresas no pueden esperar a tener un producto totalmente completo para lanzarlo al mercado, en su lugar se van introduciendo versiones cada vez más completas que de alguna manera alivian las presiones competitivas.

Los modelos evolutivos más conocidos son: El Iterativo incremental y el Espiral.

Este enfoque entrelaza las actividades de *especificación, desarrollo y validación.* Un sistema inicial se desarrolla rápidamente a partir de especificaciones abstractas. Éste se refina basándose en las peticiones del cliente para producir un sistema que satisfaga sus necesidades.



La idea de este modelo es el desarrollo de una implantación del sistema inicial, exponerla a los comentarios del usuario, **refinarla en N versiones** hasta que se desarrolle el sistema adecuado. Una ventaja de este modelo es que se obtiene una rápida realimentación del usuario, ya que las actividades de especificación, desarrollo y pruebas se ejecutan en cada iteración.

VENTAJAS DEL EVOLUTIVO RESPECTO AL SECUENCIAL.

- La especificación puede desarrollarse de **forma creciente**.
- Los usuarios y desarrolladores logran un **mejor entendimiento** del sistema. Esto se refleja en una mejora de la calidad del software.
- Es **más efectivo** que el modelo de cascada, ya que cumple con las necesidades inmediatas del cliente.

VENTAJAS DEL SECUENCIAL RESPECTO AL EVOLUTIVO.

- Los administradores **necesitan entregas para medir el progreso**. Si el sistema se necesita desarrollar rápido, no es efectivo producir documentos que reflejen cada versión del sistema por lo que un **diseño más clásico es más eficiente**.
- Sistemas mejor estructurados: Los **cambios continuos pueden ser perjudiciales para la estructura del software** haciendo costoso el mantenimiento por lo que el secuencial en cascada, más estructurado, es más eficiente para este caso.
- **Para el evolutivo se requieren técnicas y herramientas**. Para el rápido desarrollo se necesitan herramientas que pueden ser incompatibles con otras o que poca gente sabe utilizar. Por lo tanto, cuando disponemos de un equipo suficientemente experto y numeroso para encargarse de las tareas secuenciales es preferible el secuencial.

5.3- La documentación

Todas las etapas del desarrollo deben quedar perfectamente documentadas. En esta etapa, será necesario reunir todos los documentos generados y clasificarlos según el nivel técnico de sus descripciones.

- La documentación presentada se divide en dos clases:
- La documentación de proceso.
    - Estos documentos registran el proceso de desarrollo y mantenimiento.
    - Se indican planes, estimaciones, para predecir y controlar el proceso de SW, informan de cómo usara los recursos durante el proceso de desarrollo.
- La documentación de producto
    - Esta documentación describe el producto que esta siendo desarrollado.
    - Se definen dos tipos de documentación:

- Doc. Del sistema – describe el producto desde un punto de vista técnico, orientado al desarrollo, instalación y mantenimiento del mismo. (Guía técnica, Guía de instalación)
- Doc. Del usuario- que ofrece una descripción del producto orientada a los usuarios que utilizarán el sistema. (Guía de uso)

En cualquier aplicación, como mínimo, deberán generarse los siguientes documentos:

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 codifica-ción de los programas. • Las pruebas realizadas.	• Descripción de la funciona-lidad de la aplicación. • Forma de comenzar a ejecu-tar la aplicación. • Ejemplos de uso del progra-ma. • 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 técni-co en informática (analistas y pro-gramadores).	A los usuarios que van a usar la aplicación (clientes).	Al personal informático res-ponsable de la instalación, en colaboración con los usuarios que van a usar la aplicación (clientes).
¿Cuál es su objetivo?	Facilitar un co-rrecto desarrollo, realizar correc-ciones en los pro-gramas y permitir un mantenimiento futuro.	Dar a los usuarios finales toda la información necesaria para utilizar la aplicación.	Dar toda la información nece-saria para garantizar que la implantación de la aplicación se realice de forma segura, confiable y precisa.

5.4- Roles o figuras del proceso de desarrollo de SW

El equipo de desarrollo de Sw se compone de una serie de personas, o más bien roles, los cuales tiene unas atribuciones y responsabilidades diferentes. A continuación, se describirán los principales roles.

Arquitecto de Sw: Persona encargada de decidir cómo va a realizarse el proyecto y cómo va a cohesionarse. Tiene un conocimiento profundo de las tecnologías, los framework, las librerías, etc. Decide la forma y los recursos con los que va a llevarse a cabo un proyecto.

Jefe de proyecto: Dirige el proyecto. Puede ser un analista con experiencia, un arquitecto o simplemente una persona dedicada solamente a ese puesto. Tiene que saber gestionar un equipo y los tiempos, tener una relación fluida con el cliente.

Analista de sistemas: Es un rol tradicional en el desarrollo de SW. Es una persona con experiencia que realizan un estudio exhaustivo del problema que ha de analizarse y ejecuta tanto el análisis como el diseño de todo el sistema.

Analista programador: Puesto entre analista y programador. Realiza funciones de análisis porque sus conocimientos lo permiten y también codifica. En proyectos pequeños, puede realizar ambas funciones(analista y programador)

Programador: Su función es conocer en profundad el lenguaje de programación y codificar las tareas que le han sido encomendadas por el analista o analista-programador.

6. ESTILOS DE PROGRAMACIÓN

6.1 Programación convencional

La **programación convencional** usa repeticiones y saltos entremezclados con el objetivo de conseguir el funcionamiento del programa. Esta programación es confusa e implica una alta probabilidad de fallos.

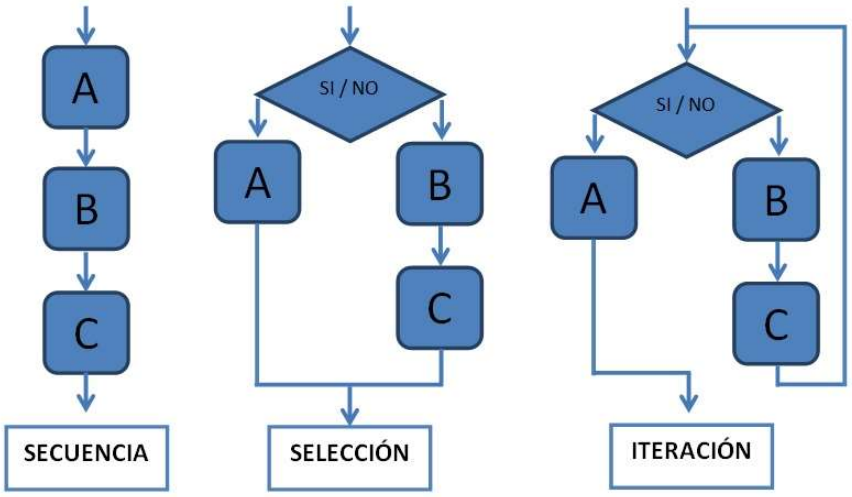
Cuanto más grande se hace el programa, llegara a un punto en el que el código se hace inmanejable.

5.2 Programación estructurada

Es una técnica de programación que usa una serie de estructuras específicas que optimizan los recursos físicos y lógicos del ordenador, establecida por *Dijhstra* y su equipo en los 70.

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

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



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

Ventajas de la Programación Estructurada:

- Los programas son fáciles de leer, sencillos y rápidos.
- El mantenimiento de los programas es sencillo.
- La estructura del programa es sencilla y clara.
- Todos los programadores utilizan las mismas técnicas por lo que es más fácil entender el código escrito por otra persona.

Inconvenientes de la Programación Estructurada:

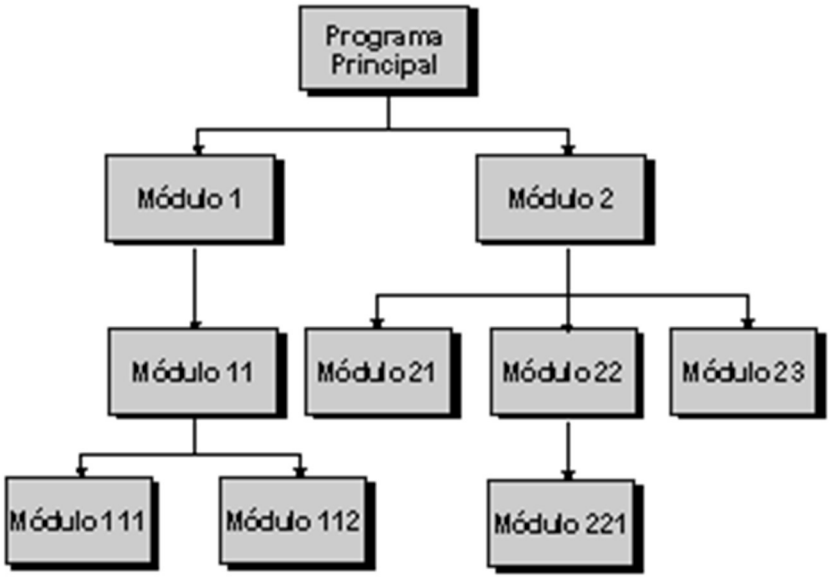
- Todo el programa se concentra en un único bloque (si se hace demasiado grande es difícil manejarlo).
- No permite reutilización eficaz de código, ya que todo va "en uno". Es por esto que a la programación estructurada le sustituyó la programación modular, donde los programas se codifican por módulos y bloques, permitiendo mayor funcionalidad.

5.3 Programación modular

Consiste en dividir un programa complejo en **programas más sencillos** que combinados resuelven el programa principal.

Uno de los métodos más conocidos para resolver un problema es dividirlo en problemas más pequeños, llamados **subproblemas**. De esta manera, en lugar de resolver una tarea compleja y tediosa, resolvemos otras más sencillas y a partir de ellas llegamos a la solución. Esta técnica se usa mucho en programación ya que programar no es más que resolver problemas, y se le suele llamar **diseño descendente**, metodología del **divide y vencerás** o programación ***top-down***.

Es evidente que si esta metodología nos lleva a tratar con subproblemas, entonces también tengamos la necesidad de poder crear y trabajar con **subprogramas** para resolverlos. A estos subprogramas se les suele llamar **módulos**, de ahí viene el nombre de programación modular. En programación disponemos de dos tipos de módulos: los **procedimientos** y las **funciones**.



5.4 Programación Orientada a Objetos

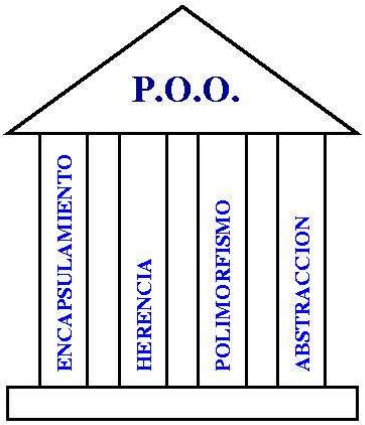
Después de comprender que la programación estructurada no es útil cuando los programas se hacen muy largos, es necesaria otra técnica de programación que solucione este inconveniente. Nace así la **Programación Orientada a Objetos** (en adelante, **P.O.O.**).

La programación orientada a objetos consiste en ordenar datos en conjuntos modulares de elementos de información del mundo real. Estos elementos de datos se llaman **objetos.**

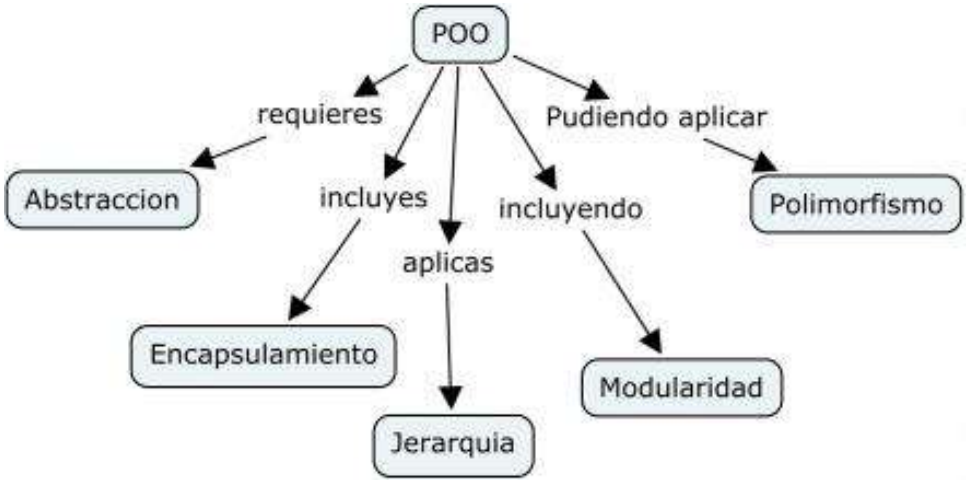
Un objeto consta de una estructura de datos y de una colección de métodos u operaciones que manipulan esos datos. Los datos definidos dentro de un objeto son sus **atributos**. Las operaciones, el comportamiento del objeto y cambiar el valor de los

atributos. Los objetos se comunican unos con otros a través del paso de **mensajes**. Una **clase** es una plantilla para la creación de objetos.

Es decir, consiste en usar **objetos y sus interacciones,** para diseñar aplicaciones y programas informáticos. Está basado en varias técnicas, incluyendo ***herencia***, ***abstracción***, ***polimorfismo*** y ***encapsulamiento***.



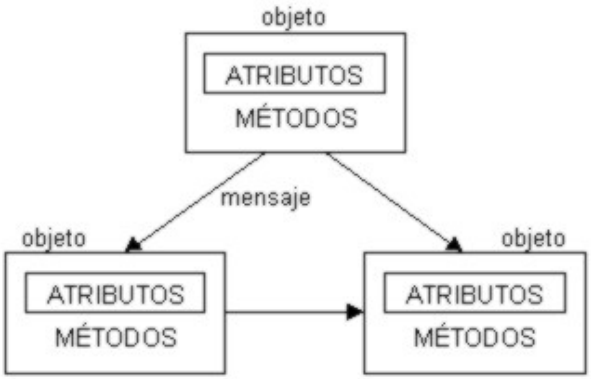
Su uso se popularizó a principios de la década de los años 1990. En la actualidad, existe una gran variedad de lenguajes de programación que soportan la orientación a objetos.



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

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

Los objetos son reutilizables para proyectos futuros.



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

A pesar de eso, alrededor del 55% del software en las empresas se hace usando esta técnica.

Razones:

- El código es reutilizable.
- Si hay algún error, es más fácil de localizar y depurar en un objeto que en un programa entero.

Características:

- Los objetos del programa tendrán una serie de atributos que los diferencian unos de otros.

- Se define ***clase*** como una colección de objetos con características similares.
- Mediante los llamados ***métodos***, los objetos se comunican 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 lenguajes orientados a objetos: *Ada, C++, VB.NET, Delphi, Java, PowerBuilder*.

7. LENGUAJES DE PROGRAMACIÓN

Un lenguaje de programación es un lenguaje formal diseñado para realizar procesos que pueden ser llevados a cabo en un ordenador. Permiten a las personas "comunicarse" con los ordenadores.

Formalmente podemos definir un **lenguaje de programación** como un conjunto de caracteres, las reglas para la combinación de esos caracteres y las reglas que definen su funcionamiento. Un lenguaje consta de los siguientes elementos:

- Un **alfabeto (léxico):** formado por el conjunto de símbolos permitidos.
- Una **sintaxis**: son las reglas que indican cómo realizar las construcciones con los símbolos del sistema.
- Una **semántica**: son las reglas que determinan el significado de las construcciones del lenguaje.

Clasificación de los lenguajes de programación

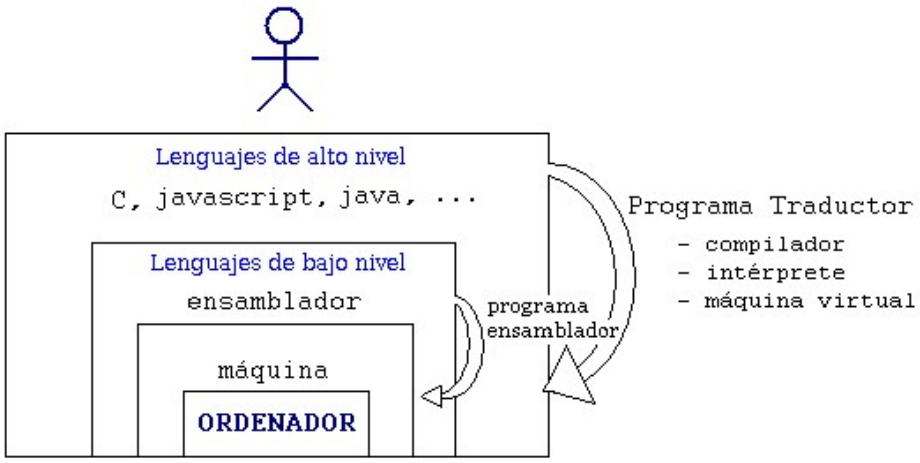
Los lenguajes de programación se pueden clasificar atendiendo a varios criterios.

Según su nivel de abstracción	Lenguajes de bajo nivel Lenguajes de nivel medio Lenguajes de alto nivel
Según la forma de ejecución	Lenguajes compilados Lenguajes interpretados
Según el paradigma de programación	Lenguajes imperativos Lenguajes funcionales Lenguajes lógicos Lenguajes de programación estructurada Lenguajes de programación orientada a objetos

6.1 Clasificación según su nivel de abstracción

El ordenador sólo entiende un lenguaje conocido como **código binario** o código máquina, consistente en ceros y unos. Es decir, sólo utiliza 0 y 1 para codificar cualquier acción.

Los lenguajes más próximos al hardware se denominan **lenguajes de bajo nivel** y los que se encuentran más cercanos a los programadores y usuarios se denominan **lenguajes de alto nivel**.



Lenguajes de bajo nivel

Son lenguajes totalmente dependientes de la máquina, es decir que el programa que se realiza con este tipo de lenguajes no se puede migrar o utilizar en otras máquinas.

Al estar prácticamente diseñados a medida del hardware, aprovechan al máximo las características del mismo.

Dentro de este grupo se encuentran:

- El **lenguaje máquina**: este lenguaje ordena a la máquina las operaciones fundamentales para su funcionamiento. Consiste en la combinación de 0's y 1's para formar las ordenes entendibles por el hardware de la máquina. Este lenguaje es mucho más rápido que los lenguajes de alto nivel. La desventaja es que son bastantes difíciles de manejar y usar, además de tener códigos fuente enormes donde encontrar un fallo es casi imposible.

- El **lenguaje ensamblador**: es un derivado del lenguaje máquina y está formado por abreviaturas de letras y números llamadas mnemotécnicos. Con la aparición de este lenguaje se crearon los programas traductores para poder pasar los programas escritos en lenguaje ensamblador a lenguaje máquina. Como ventaja con respecto al código máquina es que los códigos fuentes eran más cortos y los programas creados ocupaban menos memoria. Las desventajas de este lenguaje siguen siendo prácticamente las mismas que las del lenguaje ensamblador, añadiendo la dificultad de tener que aprender un nuevo lenguaje difícil de probar y mantener.

Lenguajes de alto nivel

Son aquellos que se encuentran más cercanos al lenguaje natural que al lenguaje máquina. Se tratan de **lenguajes independientes de la arquitectura del ordenador**.

Por lo que, en principio, un programa escrito en un lenguaje de alto nivel, lo puedes migrar de una máquina a otra sin ningún tipo de problema.

Estos lenguajes permiten al programador olvidarse por completo del funcionamiento interno de la maquina/s para la que están diseñando el programa. Tan solo necesitan un traductor que entiendan el código fuente como las características de la máquina.

Suelen usar tipos de datos para la programación y hay lenguajes de propósito general (C#, Java, Visual Basic, etc.) y de propósito específico (como FORTRAN para trabajos científicos, COBOL para programación financiera y LISP/PROLOG para inteligencia artificial).

Lenguajes de Medio nivel

Se trata de un término no aceptado por todos pero sí pueden catalogarse así. Estos lenguajes se encuentran en un punto medio entre los dos anteriores. Dentro de estos lenguajes podría situarse C ya que puede acceder a los registros del sistema, trabajar con direcciones de memoria, todas ellas características de lenguajes de bajo nivel y a la vez realizar operaciones de alto nivel.

Generaciones

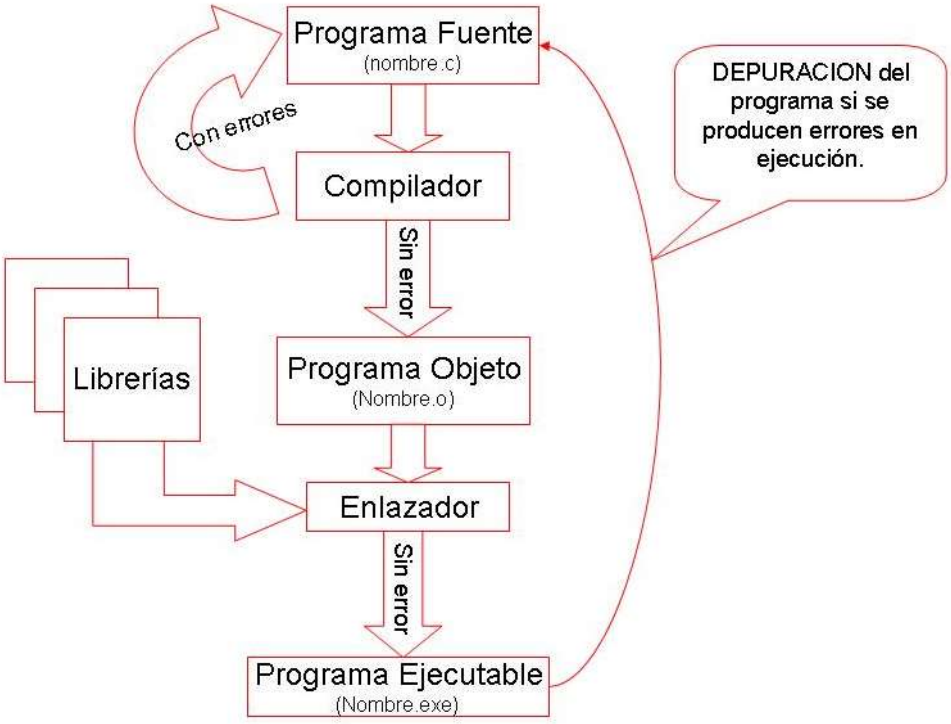
La evolución de los lenguajes de programación se puede dividir en 5 etapas o generaciones.

- **Primera generación**: lenguaje máquina.
- **Segunda generación**: se crearon los primeros lenguajes ensambladores.
- **Tercera generación**: se crean los primeros lenguajes de alto nivel. Ej. C, Pascal, Cobol…
- **Cuarta generación**. Son los lenguajes capaces de generar código por si solos, son los llamados RAD, con lo cuales se pueden realizar aplicaciones sin ser un experto en el lenguaje. Aquí también se encuentran los lenguajes orientados a objetos, haciendo posible la reutilización de partes del código para otros programas. Ej. Visual, Java…
  Estos lenguajes tienen una estructura lo más parecido al idioma inglés, algunas características son:
  - Acceso a base de datos.
  - Capacidades Gráficas.
  - Generación de código automáticamente.
- **Quinta generación**: son los lenguajes orientados a la inteligencia artificial. Ej. LISP

6.2 Clasificación según la forma de ejecución

Compiladores

Para obtener un programa ejecutable (en lenguaje máquina) a partir de un lenguaje de alto nivel (compilado), es preciso un proceso de traducción. Se divide en varias fases:



A. Compilación

En esta fase se convierte el código fuente a lenguaje máquina, pero no se asignan direcciones absolutas de memoria, pues no se sabe con exactitud a qué lugar va a ir el programa, en su lugar se colocan informaciones para el programa enlazador. Se divide en varias etapas:

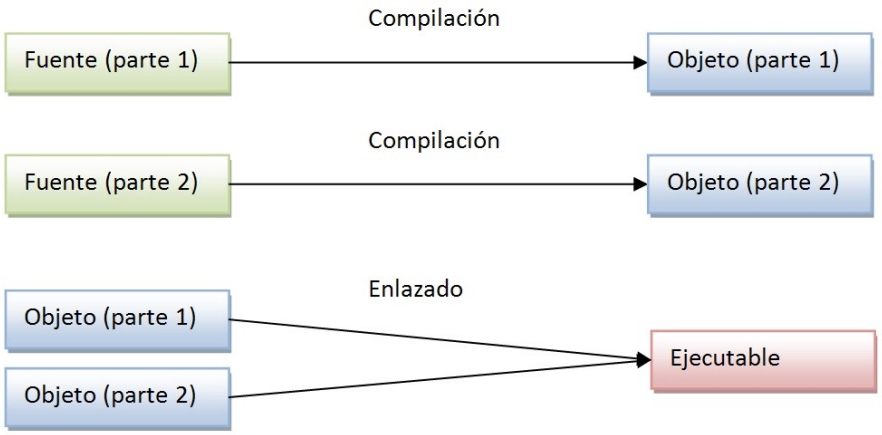
1. **preprocesamiento**: no se genera código objeto, básicamente lo que se hace son modificaciones al código fuente (se expanden las macros, pueden eliminarse ciertos módulos de programa -compilación condicional-, pueden añadirse librerías externas, ...). En C las instrucciones para el preprocesador vienen precedidas por # (#include → incluye un fichero, #define → (definición de macro) realiza una macrosustitución).
2. **generación de código intermedio**: se genera un *pseudo-código* ensamblador, es decir, independiente de la máquina. Es un lenguaje ensamblador muy genérico, que elimina todas las complejidades de este lenguaje.

3. **generación de código objeto**: el código objeto ya es código máquina, pero ciertas direcciones de memoria todavía no están resueltas y han sido sustituidas por etiquetas (nombres), pues todavía no sabemos en qué lugar de la memoria se va a cargar cada módulo del programa.

Entre medias se pueden hacer diversas **optimizaciones**, bien para que se ejecute más rápido, ocupe menos espacio, obtener el código óptimo para mi microprocesador o hacerlo más compatible con viejos procesadores.

B. Enlace

Si nuestro programa es muy extenso podemos descomponerlo en módulos que se compilan y prueban por separado, pero para construir el programa completo es necesario enlazarlos. Estos módulos también pueden ser **librerías estáticas**, que debemos enlazar antes de poder ejecutarlo (las **librerías dinámicas** se enlazan mientras se ejecuta el programa, es decir, se cargan en la memoria asignada al programa cuando se está ejecutando).



C. Ejecución

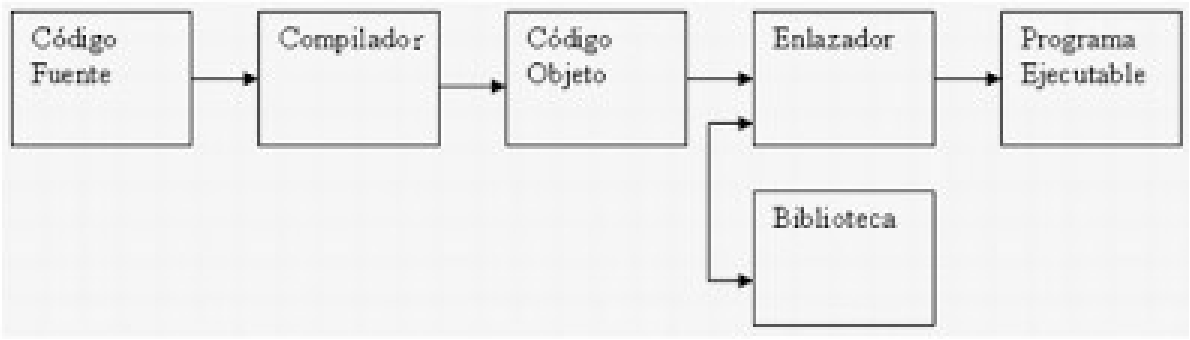
Para ejecutar un programa es necesario cargarlo en memoria. El caso más sencillo es cuando un programa cabe completamente en memoria, si el programa es demasiado grande se tendrá que dividir en **segmentos**, que se van cargando y descargando según necesidad.

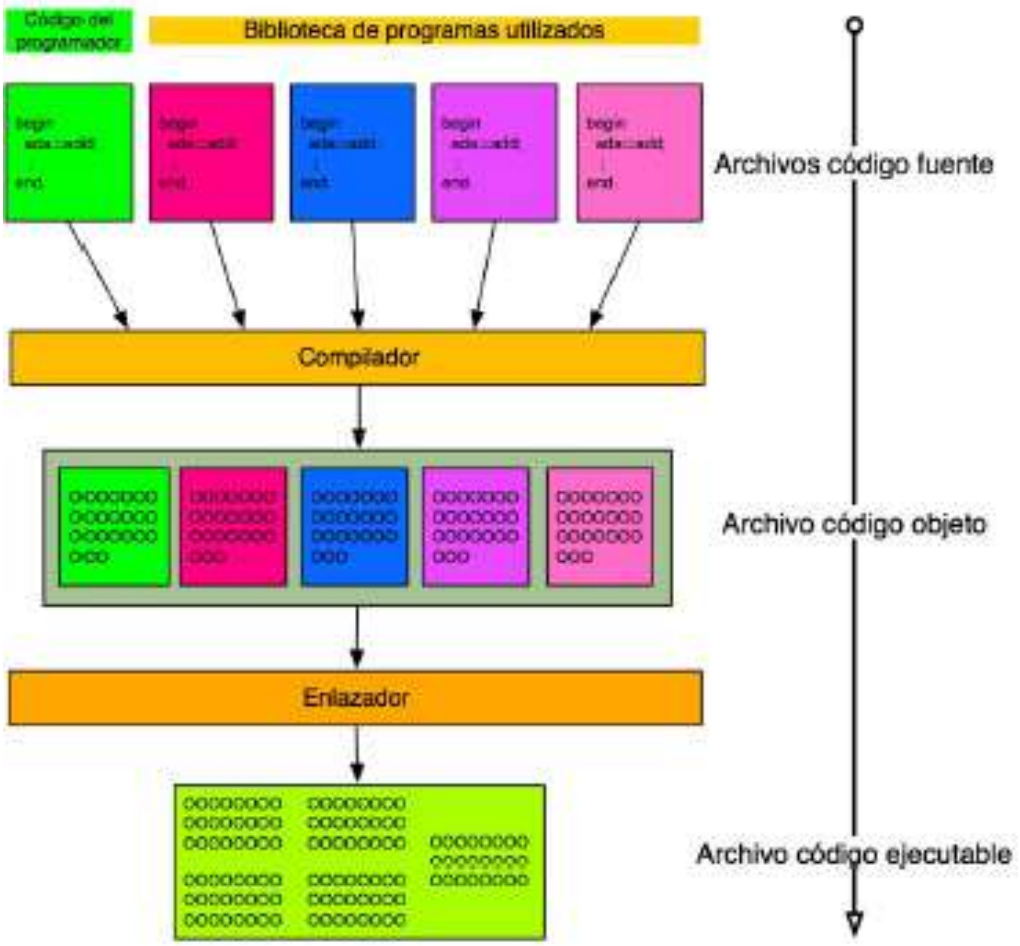
Tipos de Código

Durante todo este proceso, el código pasa por diferentes estados:

- **Código Fuente**: es el escrito por los programadores en algún editor de texto. Se escribe usando algún lenguaje de programación de alto nivel y contiene el conjunto de instrucciones necesarias.
- **Código Objeto**: es el código binario resultado de compilar el código fuente. El código objeto no es directamente inteligible por el ser humano, pero tampoco por la computadora. Es un código intermedio entre el código fuente y el ejecutable y sólo existe si el programa se compila, ya que si se interpreta (traducción línea a línea del código) se traduce y se ejecuta en un solo paso.

- **Código Ejecutable**: Es el código binario resultante de enlazar los archivos de código objeto con ciertas rutinas y bibliotecas necesarias. El sistema operativo será el encargado de cargar el código ejecutable en memoria RAM y proceder a ejecutarlo. También es conocido como código máquina y ya sí es directamente inteligible por el ordenador.

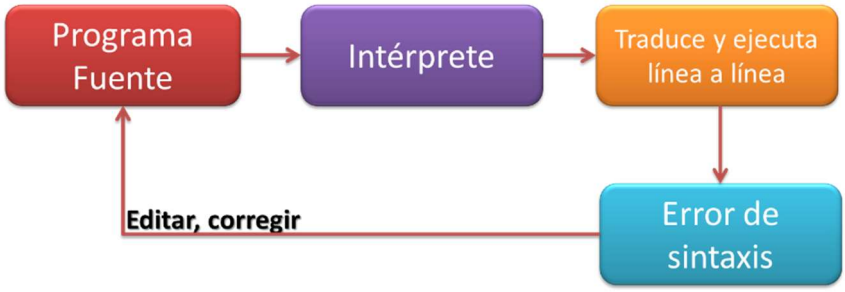




Hoy en día, con los **IDE** (***Entornos de Desarrollo Integrado***) todas estas fases se realizan de manera automática al pulsar un botón. Nosotros lo único que tenemos que hacer es trabajar en un proyecto donde se encuentran codificados todos los programas fuente (en uno o varios módulos).

Intérpretes

Es un programa informático capaz de analizar y ejecutar otros programas, escritos en un lenguaje de alto nivel. Los intérpretes se diferencian de los compiladores en que mientras estos traducen un programa desde su descripción en un lenguaje de programación al código de máquina del sistema, los intérpretes sólo realizan la traducción a medida que sea necesaria, típicamente, **instrucción por instrucción**, y normalmente **no guardan el resultado de dicha traducción**.



Los lenguajes interpretados dan a los programas cierta flexibilidad adicional sobre los lenguajes compilados. Algunas características que son más fáciles de implementar en intérpretes que en compiladores incluyen, pero no se limitan, a:

- **Independencia de plataforma** (por ejemplo PHP)
- **Pequeño tamaño del programa** (puesto que los lenguajes interpretados tienen flexibilidad para elegir el código de instrucción)

La ejecución del programa por medio de un intérprete es usualmente mucho menos eficiente que la ejecución de un programa compilado. No es eficiente en tiempo porque, o cada instrucción debe pasar por una interpretación en tiempo de ejecución, o como en más recientes implementaciones, el código tiene que ser compilado a una representación intermedia antes de cada ejecución. La máquina virtual es una solución parcial al problema de la eficiencia del tiempo pues la definición del lenguaje intermedio es mucha más cercana al lenguaje de máquina y por lo tanto más fácil de ser traducida en tiempo de ejecución.

Otra desventaja es la **necesidad de un intérprete en la máquina local** para poder hacer la ejecución posible

Según este criterio, los lenguajes de programación pueden, en líneas generales, dividirse en dos categorías:

- lenguajes interpretados
- lenguajes compilados

Lenguaje interpretado

Un lenguaje de programación es, por definición, diferente al lenguaje máquina. Por lo tanto, debe traducirse para que el procesador pueda comprenderlo. Un programa escrito en un lenguaje interpretado requiere de un programa auxiliar (el intérprete), que traduce los comandos de los programas según sea necesario.

Lenguaje compilado

Un programa escrito en un lenguaje "**compilado**" se traduce a través de un programa *anexo* llamado compilador que, a su vez, crea un nuevo archivo independiente que no necesita ningún otro programa para ejecutarse a sí mismo. Este archivo se llama **ejecutable**.

Un programa escrito en un lenguaje compilado posee la ventaja de no necesitar un programa anexo para ser ejecutado una vez que ha sido compilado. Además, como sólo es necesaria una traducción, la ejecución se vuelve más rápida. Sin embargo, no es tan flexible como un programa escrito en lenguaje interpretado, ya que cada modificación del archivo fuente (el archivo comprensible para los seres humanos: el archivo a compilar) requiere de la compilación del programa para aplicar los cambios.

Por otra parte, un programa compilado tiene la ventaja de garantizar la **seguridad del código fuente**. En efecto, el lenguaje interpretado, al ser directamente un lenguaje legible, hace que cualquier persona pueda conocer los secretos de fabricación de un programa y, de ese modo, copiar su código o incluso modificarlo. Por lo tanto, existe el riesgo de que los derechos de autor no sean respetados. Por otro lado, ciertas aplicaciones aseguradas necesitan confidencialidad de código para evitar las copias ilegales (transacciones bancarias, pagos en línea, comunicaciones seguras...).

Lenguajes intermediarios

Algunos lenguajes pertenecen a ambas categorías (LISP, Java, Python...) dado que el programa escrito en estos lenguajes puede, en ciertos casos, sufrir una fase de compilación intermediaria, en un archivo escrito en un lenguaje ininteligible (por lo tanto diferente al archivo fuente) y no ejecutable (requeriría un intérprete). Los ***applets*** Java, pequeños programas que a menudo se cargan en páginas web, son archivos compilados que sólo pueden ejecutarse dentro de un navegador web (son archivos con la extensión .***class***).

6.3 Clasificación según el paradigma de programación

Un paradigma de programación es un enfoque particular para el desarrollo del software. Define un **conjunto de reglas, patrones y estilos de programación**. Un lenguaje de programación puede usar más de un paradigma. Dependiendo del problema a resolver resultará más apropiado uno que otro.

Constituyen varias categorías de los lenguajes de programación:

Lenguajes imperativos

Los primeros lenguajes imperativos fueron los lenguajes máquina, en los que las instrucciones eran muy simples, después se utilizaron los lenguajes ensambladores. En estos lenguajes un cálculo consiste en una serie de sentencias que establecen explícitamente cómo se debe manipular la información.

La sentencia principal es la asignación. Las estructuras de control permiten establecer el orden de ejecución y cambiar el flujo del programa dependiendo de los resultados obtenidos.

La mayoría de los lenguajes usados para el desarrollo de software comercial son imperativos.

Ejemplos de lenguajes imperativos son: *Basic, Fortran, Algol, Pascal, C, Ada, C++, Java, C#.*

Lenguajes funcionales

El paradigma funcional está basado en el concepto matemático de función. Los programas escritos en estos lenguajes estarán constituidos por un conjunto de definiciones de funciones junto con los argumentos sobre los que se aplican.

En los lenguajes funcionales:

- No existe la operación de asignación.
- Las variables almacenan definiciones o referencias a expresiones.
- La operación fundamental es la aplicación de una función a una serie de argumentos.
- La computación se realiza mediante la evaluación de expresiones.

Ejemplos de lenguajes funcionales son: *Lisp, Scheme, ML, Mirana o Haskell*.

Lenguajes lógicos

Los programas escritos en estos lenguajes se pueden ver como una base de datos formada por listas de declaraciones lógicas (reglas) que se pueden consultar. La ejecución consistirá en realizar preguntas de forma interactiva.

El lenguaje lógico por excelencia es *Prolog*, está especialmente indicado para aplicaciones muy específicas como: sistemas expertos, demostración de teoremas, consulta de bases de datos relacionales, procesamiento del lenguaje natural, etc.

Lenguajes de programación estructurados

Un programa estructurado utiliza las tres construcciones básicas vistas anteriormente: secuencia, selección e iteración. Actualmente cuando se habla de programación estructurada no solemos referir también a la programación modular. Un programa estructurado puede estar compuesto por un conjunto de módulos, cada uno tendrá una entrada y una salida. La comunicación entre ellos debe estar perfectamente controlada y se debe poder trabajar de forma independiente con cada uno de ellos,

Ejemplos de lenguajes estructurados son: *Pascal, C, Fortran, Modula-2*.

Lenguajes de programación orientados a objetos

En la programación orientada a objetos está compuesto por un conjunto de objetos no por un conjunto de instrucciones o un conjunto de módulos, como en la programación estructurada.

Ejemplos de lenguajes orientados a objetos son: *C++, Java, Ada, Smaltalk*.

Ada Lovelace que está considerada como la primera programadora de ordenadores conocida en todo el mundo. De ahí, curiosamente que se hablara en su honor del lenguaje de programación ***Ada***.

6.4 Información adicional sobre lenguajes de programación.

A continuación, encontrarás una breve lista de los lenguajes de programación actuales:

Lenguaje	Principal área de aplicación	Compilado/interpretado
ADA	Tiempo real	Lenguaje compilado
BASIC	Programación para fines educativos	Lenguaje interpretado
C	Programación de sistema	Lenguaje compilado
C++	Programación de sistema orientado a objeto	Lenguaje compilado
Cobol	Administración	Lenguaje compilado
Fortran	Cálculo	Lenguaje compilado
Java	Programación orientada a Internet	Lenguaje intermediario
MATLAB	Cálculos matemáticos	Lenguaje interpretado
LISP	Inteligencia artificial	Lenguaje intermediario
Pascal	Educación	Lenguaje compilado
PHP	Desarrollo de sitios web dinámicos	Lenguaje interpretado
Perl	Procesamiento de cadenas de caracteres	Lenguaje interpretado

8.

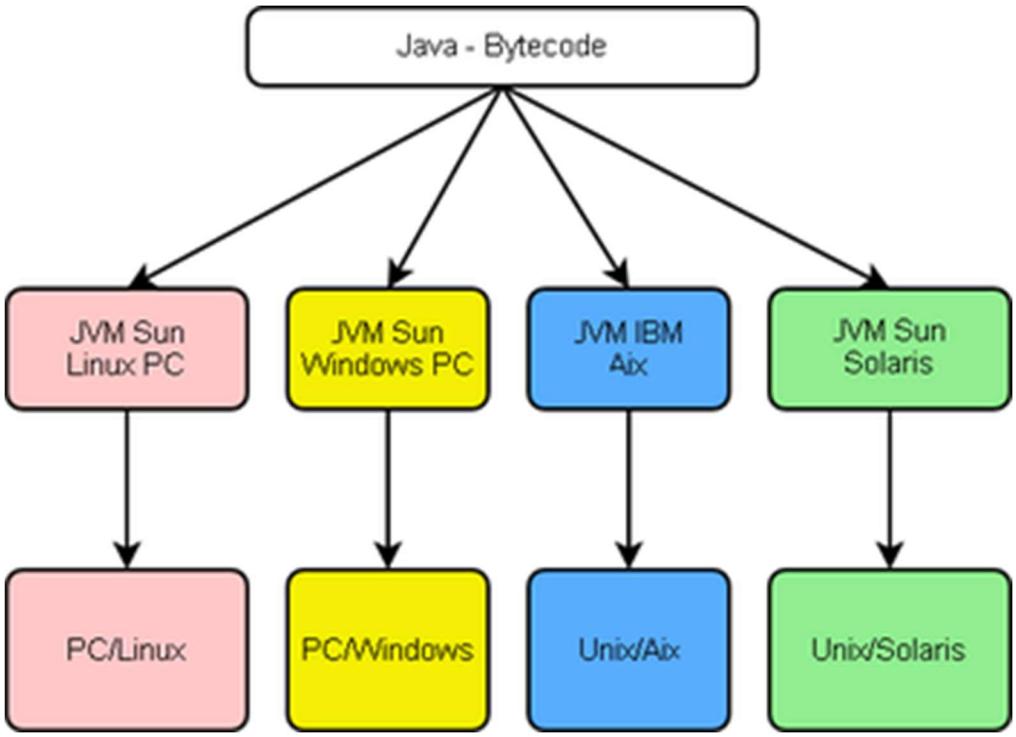
9. MÁQUINA VIRTUAL JAVA

Una **Máquina virtual Java** (en inglés *Java Virtual Machine, JVM*) es un máquina virtual de proceso nativo, es decir, ejecutable en una plataforma específica, capaz de interpretar y ejecutar instrucciones expresadas en un código binario especial (el ***bytecode*** Java), el cual es generado por el compilador del lenguaje Java

El **código binario de Java no es un lenguaje de alto nivel**, sino un verdadero código máquina de bajo nivel, viable incluso como lenguaje de entrada para un microprocesador físico. Como todas las piezas del rompecabezas Java, fue desarrollado originalmente por ***Sun Microsystems***.

La **JVM** es una de las piezas fundamentales de la plataforma Java. Básicamente se sitúa en un nivel superior al Hardware del sistema sobre el que se pretende ejecutar la aplicación, y este actúa como un puente que entiende tanto el bytecode, como el sistema sobre el que se pretende ejecutar. Así, cuando se escribe una aplicación Java, se hace pensando que será ejecutada en una máquina virtual Java en concreto, siendo ésta la que en última instancia convierte de código *bytecode* a código nativo del dispositivo final.

La gran ventaja de la máquina virtual java es aportar **portabilidad** al lenguaje de manera que desde *Sun Microsystems* se han creado diferentes máquinas virtuales java para diferentes arquitecturas y así un programa *.class* escrito en Windows puede ser interpretado en un entorno Linux. Tan solo es necesario disponer de dicha máquina virtual para dichos entornos. De ahí el famoso axioma que sigue a Java, "*escríbelo una vez, ejecútalo en cualquier parte*", o "*Write once, run anywhere*".



Pero, los intentos de la compañía propietaria de Java y productos derivados de construir microprocesadores que aceptaran el Java *bytecode* como su lenguaje de máquina fueron más bien infructuosos.

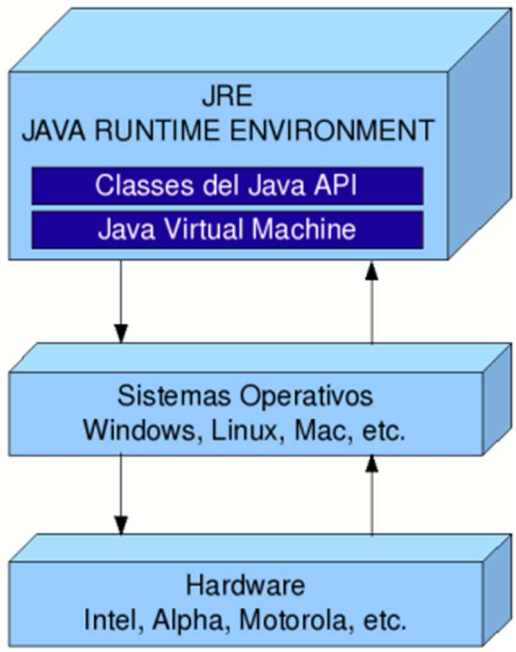
La **máquina virtual de Java** puede estar implementada en software, hardware, una herramienta de desarrollo o un *Web browser*; lee y ejecuta código precompilado *bytecode* que es independiente de la plataforma multiplataforma. La JVM provee definiciones para un **conjunto de instrucciones**, un **conjunto de registros**, un **formato para archivos de clases**, la **pila**, un **heap** con recolector de basura y un **área de memoria**. Cualquier implementación de la JVM que sea aprobada por *SUN* debe ser capaz de ejecutar cualquier clase que cumpla con la especificación.

Existen varias versiones, en orden cronológico, de la máquina virtual de Java. En general la definición del Java *bytecode* no cambia significativamente entre versiones, y si lo hacen, los desarrolladores del lenguaje procuran que exista compatibilidad hacia atrás con los productos anteriores.

7.1 Java Runtime Enviroment (JRE).

En su forma más simple, el entorno en tiempo de ejecución de Java está conformado por una **Máquina Virtual de Java o JVM**, un conjunto de bibliotecas Java y otros componentes necesarios para que una aplicación escrita en lenguaje Java pueda ser ejecutada. El JRE actúa como un "intermediario" entre el sistema operativo y Java.

La **JVM** es el programa que ejecuta el código Java previamente compilado (*bytecode*) mientras que las librerías de clases estándar son las que implementan el API de Java. Ambas JVM y API deben ser consistentes entre sí, de ahí que sean distribuidas de modo conjunto.



Un usuario sólo necesita el JRE para ejecutar las aplicaciones desarrolladas en lenguaje Java, mientras que para desarrollar nuevas aplicaciones en dicho lenguaje es necesario un entorno de desarrollo, denominado JDK, que además del JRE (mínimo imprescindible) incluye, entre otros, un compilador para Java.

7.2 Java Development Kit (JDK).

Java Development Kit o (**JDK**), es un software que provee herramientas de desarrollo para la **creación de programas en Java**. La escritura de aplicaciones y *applets* de Java necesita herramientas de desarrollo como JDK. JDK incluye *Java Runtime Environment*,

el compilador *Java* y las *API* de Java. Familiarizarse resulta fácil para los programadores nuevos y con experiencia.

El **API Java es una Interfaz de Programación de Aplicaciones** (API: por sus siglas en inglés) provista por los creadores del lenguaje Java, y que da a los programadores los medios para desarrollar aplicaciones Java. Como Java es un Lenguaje Orientado a Objetos, la API de Java provee de un conjunto de clases utilitarias para efectuar toda clase de tareas necesarias dentro de un programa.

10. HERRAMIENTAS DE PROGRAMACIÓN

Para llevar a cabo la codificación y prueba de los programas se suelen utilizar entornos de programación. Estos entornos nos permiten realizar diferentes tareas:

- Crear, editar y modificar el código fuente del programa.
- Compilar, montar y ejecutar el programa.
- Examinar el código fuete.
- Ejecutar el programa en modo depuración.
- Realizar pruebas del programa de forma automática.
- Generar documentación.
- Gestionar los cambios que se van haciendo en el programa (control de versiones).

A estos entornos de programación se les suele llamar **entornos de desarrollo integrado** o **IDE** (*Integrated Development Enviroment*). Los IDEs están diseñados para maximizar la productividad del programador. Un IDE es un programa informático formado por un conjunto de herramientas de programación que facilitan las tareas de creación, modificación, compilación, implementación y depuración de software.

La mayoría de los IDEs actuales proporcionan un entorno de trabajo visual formado por ventanas, barras de menús, barras de herramientas, pestañas, paneles laterales, que nos permiten aplicar determinadas herramientas o bien nos permiten tener abiertos varios ficheros del proyecto, asistentes que ayudan al programador en el desarrollo, etc.

Los editores suelen ofrecer facilidades como el resaltado de la sintaxis utilizando diferentes colores y tipos de letra, el emparejamiento de llaves o paréntesis el plegado y desplegado de código, etc.

Un mismo IDE funcionar con varios lenguajes de programación, este es el caso de ***Eclipse*** o ***Netbeans o IntelliJ IDEA o Visual Studio*** que mediante la instalación de *pluging* se le puede añadir soporte de lenguajes adicionales.