

TEMA 1 – INTRODUCCIÓN AL DESARROLLO DE SOFTWARE

AN INTRODUCTION TO SOFTWARE DEVELOPMENT

- 1.1- Introducción. Conceptos básicos
- 1.2- Traductores: intérpretes, compiladores y el caso Java
- 1.3- Generaciones, lenguajes y ordenadores
- 1.4- Técnicas de programación
- 1.5- Ingeniería del software
- 1.6- Ciclo de vida del software
- 1.7- Principales metodologías de desarrollo
- 1.8- Licencias de software. software libre y propietario
- 1.9- Conceptos adicionales

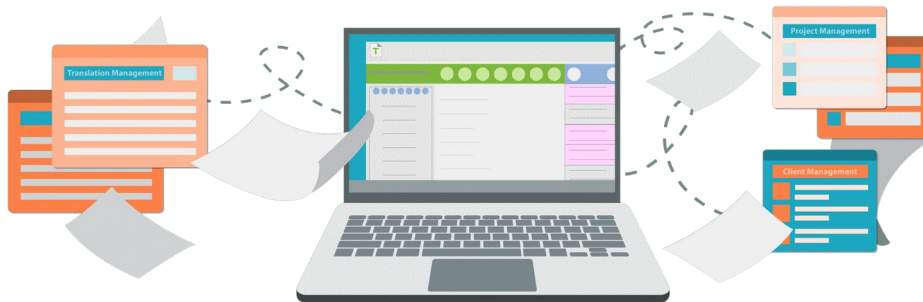


Photo by [LYCS Architecture](#) on [Unsplash](#)

1.1 – Introducción. Conceptos básicos

Es de sobra conocido que cuando hablamos de informática se distinguen dos componentes bien diferenciados: hardware y software.

- El **hardware** lo constituyen todas los elementos meramente físicos, la máquina en su conjunto, con cada una de las piezas que lo conforman: placa base, microprocesador, memoria, pantalla o monitor, periféricos, etc...
- El **software**, que es el conjunto de código informático que utiliza el hardware para representar, almacenar y ejecutar lo que el usuario desee

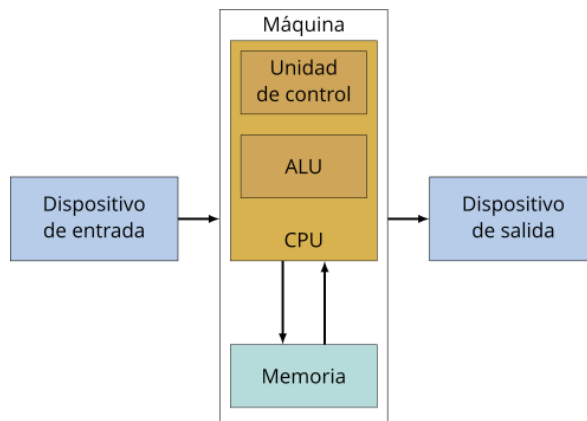


Vectorial Image by Transwise

Pero conviene aclarar un par de cosas:

1. **La primera, relativa al hardware:** que la informática ya no trata solamente de ordenadores, sino que también incluye, implícitamente, el uso de móviles, tablets (y así lo entenderemos nosotros) y también todo tipo de máquinas que puedan conectarse y programarse, como asistentes de voz, electrodomésticos, coches, etc... Es el momento del **IoT, Internet of Things** o Internet de las cosas.

Centrándonos en el hardware de ordenadores, tablets y smartphones, encontramos siempre de una u otra forma, los componentes básicos de la llamada **Arquitectura Von Neumann**:



Graphic from [Wikipedia](#)

- **Los dispositivos de entrada** pueden ser: teclado, ratón, pantalla táctil, sensores de señales analógicas (voz, temperatura, luz, humedad)
- **Los dispositivos de salida** pueden ser: pantalla, impresora, señales analógicas (voz, temperatura, luz,...)
- **La CPU** contiene el procesador (formado a su vez por Unidad de Control y Unidad Aritmético Lógica)
- **La memoria central** (a diferentes niveles: registros, varios niveles de caché, RAM y ROM)
- Aunque no aparezca en el gráfico, debido a que proviene de una época en que no existía Internet, también se suele contar con algún dispositivo de **conexión en red** en la gran mayoría de los casos
- Finalmente, aunque no menos importante, debido a que la memoria central es volátil, siempre resulta necesario disponer de **memorias secundarias** que almacenen la información de forma estable: pen drives, discos duros, discos sólidos, o discos ópticos.

1. La segunda, que históricamente, y según su funcionalidad, se suelen distinguir **tres tipos de software**:

1. Sistema operativo ó software básico
2. Software para el desarrollo de programación
3. Aplicaciones de usuario final

Veamos de qué hablamos en cada caso particular:

- **Sistema Operativo (*Operating system*) :**

Es el software básico y mínimo que ha de estar instalado y configurado en nuestro ordenador para que todos los demás programas puedan ejecutarse y funcionar. Es el que se encarga de manejar directamente el hardware al más bajo nivel.

Son ejemplos de sistemas operativos para ordenadores: Windows, Linux, Mac OS, Unix... y para dispositivos móviles y tablets: Android, iOS, etc...



Photo from [Flickr](#)

- **Software para el Desarrollo de programación** (*CASE Tools, Computer Aided Software Engineering Tools*)

Es el conjunto de herramientas que nos permiten crear programas informáticos. Las hay de muy diversos tipos, según la fase de desarrollo y la plataforma sobre la que se ejecutan, aunque cada vez más, suelen ser multiplataforma o “en la nube”. Podemos poner como ejemplo, *Notion, Taiga, Jira* o *Trello* para la planificación de tareas, o *Eclipse, IntelliJ* y *Visual Studio Code* para la escritura, depuración y testeo de código, toda la gama de productos *Rational* de IBM, que cubre distintas fases del desarrollo de software, etc...

- **Aplicaciones informáticas** (*Computer or Mobile applications*)

Son programas que tienen una **utilidad funcional** para el usuario final más o menos específica. Son ejemplos de aplicaciones: un procesador de textos, una hoja de cálculo, el software para reproducir música, un navegador, un videojuego, una app para encargar comida, o para monitorizar la actividad deportiva, etc



Photo by [Rami Al-zayat](#) on [Unsplash](#)

En este módulo en particular, nuestro interés se va a centrar expresamente en las aplicaciones informáticas, concretamente en **cómo se desarrollan** y cuáles son las fases por las que necesariamente han de pasar durante este proceso. Y para ello, tendremos que utilizar diferentes **herramientas de desarrollo** o **CASE tools**. Dedicaremos, de hecho, bastante tiempo a instalar, configurar y utilizar algunas de las más conocidas en el entorno productivo.

Las aplicaciones informáticas, pueden ser de 3 tipos genéricos:

- **Aplicaciones de escritorio:** los programas tradicionales que se instalan y configuran en un equipo y (en principio, aunque no siempre) no necesitan conexión a internet para funcionar, p.ej. cualquiera de los componentes de una suite ofimática como Open Office, Libre Office o Microsoft Office, cualquiera de las aplicaciones Adobe: Acrobat, Photoshop, Illustrator, Dreamweaver, ...
- **Aplicaciones web:** Una aplicación web no se instala ni se ejecuta en el ordenador particular del usuario, sino en un servidor de internet, y el usuario accede a ella a través de un navegador, que le ofrece un interfaz de comunicación construido con HTML+CSS+JS. Ejemplos de aplicaciones web son: Gmail, Hotmail, Google Docs, una plataforma Moodle...

- **Aplicaciones móviles (o multiplataforma):** con la aparición de smartphones y tablets, aparecen este tipo de aplicaciones, denominadas apps (pequeñas aplicaciones) que permiten optimizar las características de capacidades limitadas y movilidad de estos dispositivos, ofreciendo múltiples funcionalidades: geolocalización, salud, compra on-line, etc... muchas veces se trata de la versión móvil de una aplicación web, pero no siempre

Y dentro de las aplicaciones móviles, nos encontramos otros 3 subtipos:

3.1. **Nativas:** son aplicaciones móviles desarrolladas utilizando el lenguaje propio de cada plataforma (Java para Android y Swift u Objective C para iOS). Se suelen publicar en las tiendas de aplicaciones propias, como la App Store de Apple y Google Play de Android

3.2. **WebApp:** en realidad son páginas Web adaptadas para que se vean correctamente en dispositivos móviles y tratan de emular el funcionamiento de una App nativa. Ofrecen independencia con respecto al dispositivo, pero a cambio, no sacan partido al 100% de las características de cada plataforma

3.3. **Híbridas:** son una mezcla entre WebApp y App nativa y en función de la tecnología usada, pueden tener características más o menos avanzadas. Suelen encapsular una Web dentro de un WebView nativo, con lo que en realidad es una WebApp dentro de un navegador.

A lo largo de esta primera unidad vamos a aprender los conceptos fundamentales entorno al software de usuario final, y las fases del llamado ciclo de vida del software (*software lifecycle*).

También aprenderemos a distinguir los diferentes lenguajes de programación y los procesos de ingeniería que se siguen desde que una aplicación se concibe hasta que está **en producción**, osea, en uso.

Finalmente, el software básico, esto es, los sistemas operativos, serán objeto de estudio en otros módulos de este ciclo formativo.

Veamos antes de nada una serie de definiciones:

Programa informático (*Computer program*)

La definición que da la RAE de programa informático es:

“Conjunto unitario de instrucciones que permite a un ordenador realizar funciones diversas, como el tratamiento de textos, el diseño de gráficos, la resolución de problemas matemáticos, el manejo de bancos de datos, etc.”

Pero normalmente, y de forma más práctica, se entiende por programa un:

- conjunto de instrucciones ordenadas
- entendibles y ejecutables por un ordenador
- siguiendo una sintaxis no ambigua
- que implementan un cierto algoritmo

Photo from [Pixabay](#)

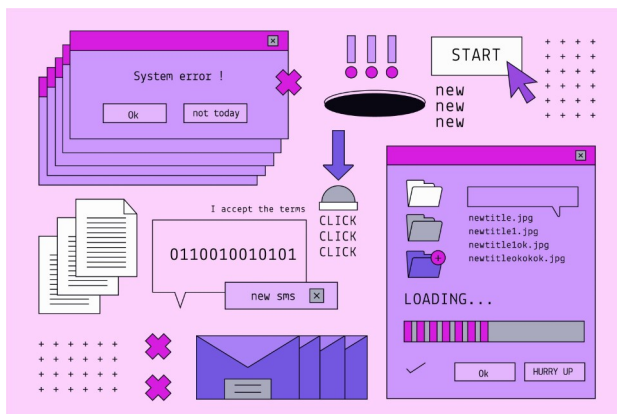
También se suele decir que un programa es un **algoritmo codificado en un lenguaje que entienden los ordenadores**. Lo que nos lleva a la siguiente definición:

Algoritmo (*Algorithm*)

Un algoritmo es un

- conjunto de órdenes o pasos
- ordenado y finito
- que describen como resolver un problema

En sentido amplio, una receta de cocina se ajusta a esta definición. Lo que hace que un algoritmo se constituya en programa es el que se utilice en su enunciado un lenguaje (símbolos, reglas sintácticas...) que entienda una máquina.



Vectorial image from [Freepik](#)

Por otra parte, a este enfoque mecanicista y clásico, habría que añadirle un peso creciente al factor de facilidad de uso: interfaz humano/máquina o usabilidad (**UX, User eXperience**). Un programa cuenta ya, además de con una lógica interna o **back-end**, con un diseño cada vez más complejo de este interfaz, el **front-end**

Aplicación (*Application*)

Se denomina aplicación a una pieza de software formado por uno o más programas, la documentación de los mismos y los archivos necesarios para su funcionamiento, de modo que el conjunto completo de archivos solucionan un problema de manejo de información y conforman una herramienta de trabajo en un ordenador.

Esto era así hasta la irrupción en escena de los dispositivos móviles. En este caso, las aplicaciones que se ejecutan en ellos se suelen denominar **Apps**, etimológicamente, pequeñas aplicaciones.

NOTA: Normalmente, en el lenguaje cotidiano no se distingue entre aplicación y programa. Pero en nuestro caso, entenderemos que la aplicación es un software complejo, compuesto por un conjunto de módulos y componentes, más acabado, mientras que un programa sería un cierto código entendible por el ordenador, y que podría ser perfectamente un único componente sencillo aislado, o ser integrante de una aplicación marco mayor.

Lenguajes de programación (*Programming languages*)

Para el desarrollo o construcción de una aplicación se utilizan uno o varios lenguajes de programación.

Podemos decir que un lenguaje de programación es una **notación o conjunto de símbolos y caracteres combinados entre sí de acuerdo con una sintaxis definida**.

Esa sintaxis está diseñada para posibilitar la transmisión clara de instrucciones al hardware, por lo que suele ser bastante **exigente y preciso**, no contiene ambigüedades.

Los símbolos de dichos lenguajes, son traducidos a un conjunto de señales eléctricas representadas en **código binario** (0 y 1), siendo éste, en verdad, **el único código directamente inteligible por parte del microprocesador**.

Existen cientos de lenguajes diferentes. Históricamente se han clasificado en dos grandes grupos:




















- **Lenguajes de bajo nivel (*Low-level language*)**: dependen absolutamente de la arquitectura de cada máquina

- **Lenguajes máquina (*Machine language*)**: prácticamente ya no se utilizan, ni siquiera en electrónica. Contienen un repertorio de instrucciones para programar directamente en 0's y 1's, y son diferentes para cada procesador
- **Lenguajes Ensamblador (*Assembly language*)**: se sustituyen cadenas binarias de operandos y códigos de operaciones por nombres simbólicos, que necesitan ser posteriormente traducidas a 0's y 1's por un software traductor específico, denominado también ensamblador.
- **Lenguajes de alto nivel (*High-level language*)**: más cercanos al lenguaje humano, aunque con restricciones sintácticas. Si se utiliza un lenguaje de este tipo, es imprescindible contar con un **software traductor** que convierta las instrucciones de un programa escrito en ellos a código máquina.

Pueden establecerse actualmente varias subdivisiones en los lenguajes de alto nivel pero dependen de los autores y no están estandarizadas.

Si consultamos la página de [TIOBE](#) podremos saber en cada momento la popularidad y evolución en el uso de los lenguajes de programación de alto nivel.

En la fecha actual (agosto de 2021), encabezan el ranking, por este orden: C, Python, Java, C++, C#, Visual Basic .NET, JavaScript, PHP, Assembly y SQL. Los 3 primeros vienen liderando el ranking en los últimos 15 años, y Python no para de crecer. Javascript, por su parte, también está teniendo una evolución al alza.

Aug 2021	Aug 2020	Change	Programming Language	Ratings	Change
1	1		 C	12.57%	-4.41%
2	3	▲	 Python	11.88%	+2.17%
3	2	▼	 Java	10.43%	-4.00%
4	4		 C++	7.38%	+0.52%
5	5		 C#	5.14%	+0.48%
6	6		 Visual Basic	4.87%	+0.01%
7	7		 JavaScript	2.95%	+0.07%
8	9	▲	 PHP	2.19%	-0.05%
9	14	▲	 Assembly language	2.03%	+0.99%
10	10		 SQL	1.47%	+0.02%
11	18	▲	 Groovy	1.38%	+0.59%
12	17	▲	 Classic Visual Basic	1.23%	+0.41%
13	42	▲	 Fortran	1.14%	+0.83%
14	8	▼	 R	1.05%	-1.75%
15	15		 Ruby	1.01%	-0.03%
16	12	▼	 Swift	0.98%	-0.44%
17	16	▼	 MATLAB	0.98%	+0.11%
18	11	▼	 Go	0.90%	-0.52%
19	38	▲	 Prolog	0.80%	+0.41%
20	13	▼	 Perl	0.78%	-0.33%

Snapshot from [Tiohe](#)

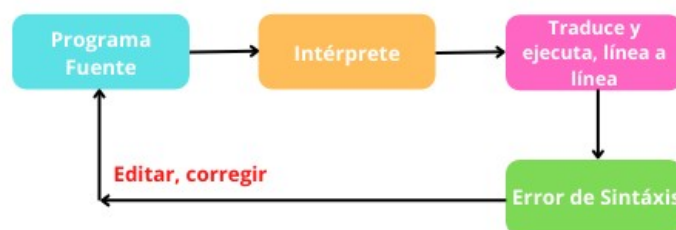
1.2 – Traductores: Intérpretes, compiladores y el caso Java

Los traductores son programas que traducen los **programas fuente** (código fuente, o **source code**) escritos en lenguajes de alto nivel (más cómodos de usar para las personas y potentes a la hora de expresar algoritmos) a programas ejecutables (en código máquina, el único que entienden los procesadores o **machine code**), y que tradicionalmente se han clasificado en dos grandes tipos: intérpretes y compiladores... hasta la llegada de Java.

Intérpretes (*Interpreters*)

Un intérprete traduce un código fuente escrito en un lenguaje de alto nivel a lenguaje máquina, instrucción por instrucción, y ejecuta cada orden una vez que se traduce, sin almacenar el resultado de esa traducción, o sea, sin almacenar las instrucciones binarias para posteriores ejecuciones.

En cierto sentido, su trabajo es similar al de un traductor humano que intermediara en la conversación entre dos personas que no hablan el mismo idioma. Iría traduciendo la conversación de viva voz conforme se produce.



Las ventajas que ofrece el proceso de traducción interpretada suelen ser relativas a la **flexibilidad en la fase de depuración y en el manejo de estructuras de datos y sintaxis**. Los lenguajes interpretados tienen unas reglas sintácticas menos exigentes, y si bien esto da más libertad al programador, puede convertirse en un inconveniente también si no se tiene una buena disciplina como programador.

Otra ventaja que se les suele atribuir a estos lenguajes, es la **portabilidad del código**: al estar el fuente siempre disponible, sus instrucciones no están comprometido con ninguna arquitectura hardware particular al más bajo nivel.

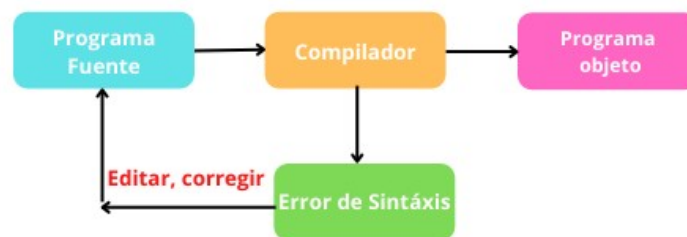
Su inconveniente principal es la **lentitud en la ejecución**, ya que cada vez que se quiere correr un código, éste debe ser traducido de nuevo línea a línea.

Ejemplos de lenguajes interpretados son: **JavaScript, PHP, Ruby, Python y Smalltalk**.

Finalmente, nótese que si el código fuente siempre está disponible a la hora de traducir-ejecutar, es **prácticamente imposible proteger el producto con copyright**, a no ser que el fuente resida en el lado servidor, en un entorno de ejecución web, y se sirva al cliente, el código ya traducido. Es el caso, por ejemplo, de PHP.

Compiladores (*Compilers*)

Un compilador es un programa que traduce un programas escrito en un lenguaje de alto nivel (llamado **programa fuente** o *source program*) a un código en lenguaje máquina (llamado **programa objeto** u *object program*). Éste programa obtenido no es directamente ejecutable pues es simplemente la traducción línea a línea del fuente y aún necesita de algún enlace y adecuación adicional.



Fases de la Compilación

Una vez obtenido un código objeto mediante la compilación, y tal y como hemos comentado previamente, se hace necesario el montaje o ensamblaje de sus módulos mediante un software específico denominado **montador o enlazador** (*Linker*). Este proceso genera un programa en código máquina directamente ejecutable.

Por tanto, y recapitulando, los pasos para convertir un programa escrito en programa ejecutable son:

- 1 Escritura del programa fuente mediante un editor y almacenaje en disco.
- 2 Compilación o traducción del código completo
- 3 Verificación de errores de compilación (listado de errores, en algunos casos)
- 4 Obtención del programa objeto, una vez libre de errores sintácticos.
- 5 Montaje mediante un enlazador y obtención de programa ejecutable.
- 6 Ejecución del programa. Si no existen errores en la lógica del programa, se obtendrá la salida adecuada a partir de los datos introducidos

La ventaja que ofrece esta forma de traducción es básicamente la **rapidez en la ejecución del programa final**, pues una vez obtenido el código ejecutable, la máquina puede proceder directamente a ello. Y también, en el caso de los productos comerciales, el hecho de que el código fuente quede oculto.

Los inconvenientes principales son, que la fase de depuración puede resultar, en algunos casos bastante más penosa, y que la sintaxis de los lenguajes compilados

suele ser mucho más estricta y más exigente en cuanto a la declaración de datos y la estructuración del código.

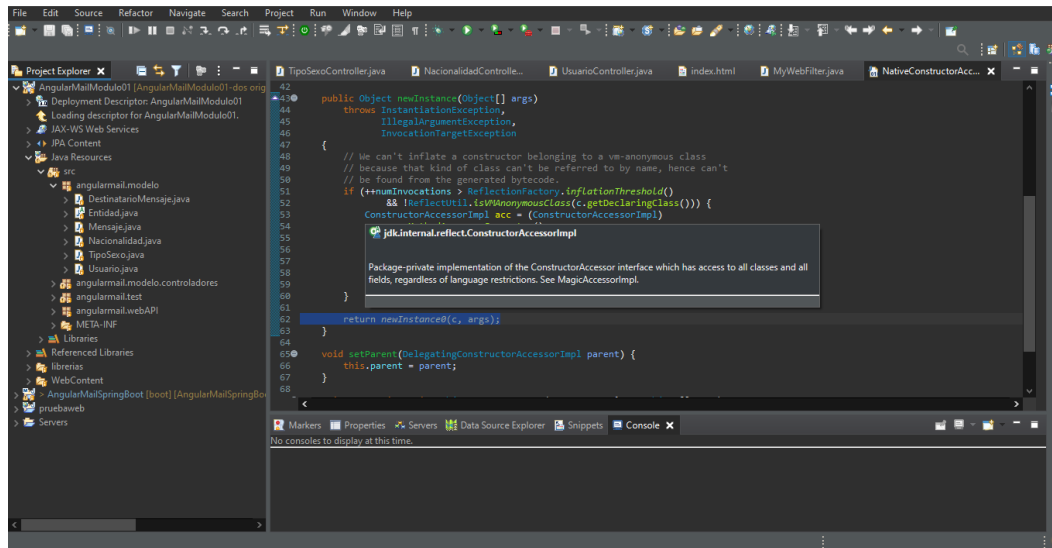
Ejemplos de lenguajes compilados son: **C**, **C++** y **VisualBasic**.

Tanto si un lenguaje es interpretado como compilado, en la actualidad, casi siempre se suele trabajar el proceso completo desde un **IDE** o **Integrated Development Environment** (Entorno de Desarrollo Integrado).

Éste consta de un editor con las funciones necesarias para la traducción, depuración, testing, control de versiones de los programas, etc.

Es el caso de **Eclipse**, **IntelliJ**, **Visual Studio Code**, **Sublime Text**, etc...

Dado que es la herramienta de trabajo principal de todo programador/a, le dedicaremos especial atención en este módulo de Entornos de Desarrollo.



Entorno de desarrollo integrado Eclipse

Cuando un programa sin errores sintácticos (**syntax errors**), y ya compilado, en el proceso de pruebas o testing, obtiene salidas no adecuadas, se dice que posee **errores en su lógica de programación (logic errors)**.

Los errores son uno de los caballos de batalla de los programadores ya que **siempre están presentes** y a veces son muy difíciles de detectar, y en el caso de aplicaciones medianas y/o complejas, se asume que éstos aparecerán por mucho que intenten evitarse, de ahí que en la mayoría de las aplicaciones comerciales se distribuyan parches o **patches** posteriores al despliegue (**deployment**) para subsanar errores no encontrados en la fase de depuración.

Una de las labores del programador, además de codificar, es la de prevenir, predecir, encontrar y subsanar en la medida de lo posible o al menos controlar (si se dan cuando el producto esté entregado), los errores del código que produzca, aunque

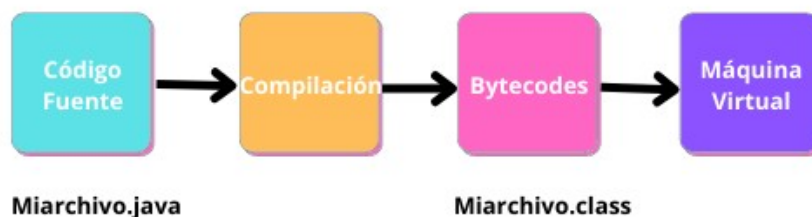
se cuente, como ya veremos, con la existencia de **testers** e **Ingenieros de QA** (*Quality Assurance Engineers*) dedicados a esta labor.

Una mala gestión de errores causa experiencias poco gratas al usuario de una aplicación.

El caso Java

Java no es un lenguaje ni interpretado ni compilado, sino que sufre un proceso de traducción a código máquina en dos fases:

- **Primera fase:** se compila el código fuente a un código que no es el código máquina, sino un lenguaje intermedio denominado **Bytecode**. Este código no es directamente ejecutado por ninguna máquina real en particular, sino por la denominada **JVM** o *Java Virtual Machine*.
- **Segunda fase:** en una arquitectura de computador determinada, el Bytecode es interpretado y ejecutado **para el código máquina de esa máquina en particular**. Evidentemente, es necesario que el ordenador tenga previamente instalado su propia versión de JVM que es la que “conoce” su código máquina específico.



Esta forma tan peculiar de traducción permite la portabilidad prácticamente total de un mismo bytecode. Según el lema de Java: *"Write once, run anywhere"* (escribe una vez y ejecuta en cualquier lugar) y a esto se debe, en definitiva, el principal factor del éxito del lenguaje.

A esto y a que el bytecode “protege” el secreto de cómo está hecho el programa, pues el código fuente no se encuentra a la vista en una distribución.

1.3 – Generaciones, lenguajes y ordenadores

A lo largo de la historia de la informática, desde los años 40 del pasado siglo hasta hoy, se cuentan cinco generaciones de lenguajes de programación. Los pasos de esta evolución, han ido encaminados desde el punto de vista del Software a facilitar por un lado la sintaxis, alejándola del código máquina y acercándola al lenguaje humano, y por otro a la creación de instrucciones potentes que puedan ser traducidas

fácil y cómodamente en un código máquina eficiente y en cualquier plataforma hardware.

Primera Generación (1GL) – Años 40-50

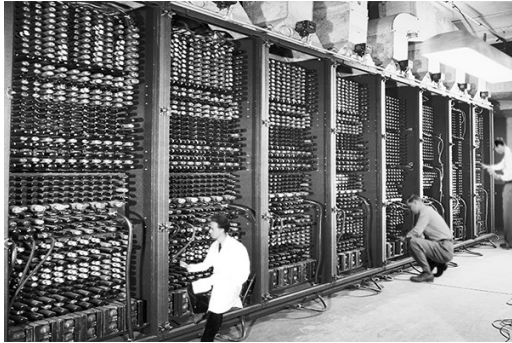


Photo from tutorialandexample.com

Los lenguajes de los primeros ordenadores (en realidad, prototipos) son conocidos como **lenguajes máquina**. Consistían enteramente en una secuencia de 0s y 1s que los controles de la computadora interpretaban como instrucciones.

Se trataba de lenguajes absolutamente dependientes de cada máquina en particular. El oficio de programador no existía como tal, eran los propios ingenieros constructores de estos ordenadores los encargados de codificar las ordenes binarias.

Los **lenguajes ensambladores** (*assembly languages*) supusieron un primer avance en cuanto a que, en lugar de secuencias de 0's y 1's, se describían tanto las operaciones como los datos a través de símbolos. Además, marcaron el camino a seguir a partir de ahí

Aparecen los primeros y primitivos **traductores de lenguajes**, que convertían las instrucciones ensamblador en código máquina.

Segunda Generación (2GL) – Finales de los 50

Aparecen los primeros **lenguajes de alto nivel**. Un lenguaje de alto nivel tiene una gramática y una sintaxis similar a las palabras en una oración. El proceso de traducción se vuelve más complejo cada vez.

Desde la actualidad, y en perspectiva, se denomina a estos lenguajes como de **alto nivel no estructurados**. Los primeros fueron: Fortran, Cobol y Basic

- ✓ El primero, **Fortran** (*Formula Translator*) es un lenguaje creado para aplicaciones científicas y matemáticas.
- ✓ El segundo, **Cobol** (*Common Business Oriented Language*), fue en cambio diseñado para aplicaciones de gestión de grandes volúmenes de datos almacenados, en un principio, en cintas magnéticas, y posteriormente, en discos.
- ✓ Basic (*Beginner's All-purpose Symbolic Instruction Code*), de propósito general, se hizo muy popular al quedar asociado a la informática personal. Fue el primer producto patentado por Microsoft. Pertenece a la tipología de

esta generación, aunque cronológicamente, aparece en el mercado durante la siguiente generación, con la creación de **ordenadores personales**



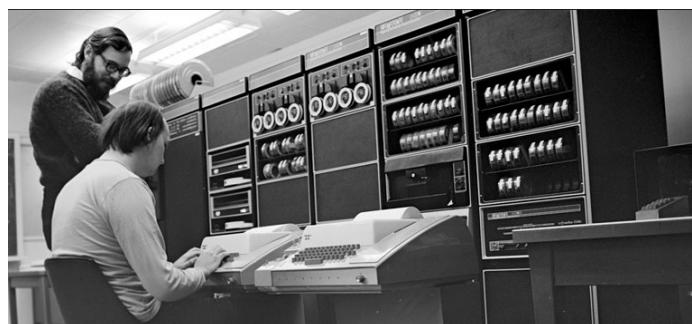
Photo from [Flickr](#)

Tercera Generación (3GL) – Años 70 en adelante

La tercera generación de lenguajes de programación se conoce como **lenguajes de alto nivel estructurados** (*structured high-level language*). La experiencia acumulada de los errores cometidos en las décadas anteriores en cuanto a planificación, eficiencia y reusabilidad del código, dio lugar a la creación de estos lenguajes tras la llamada **Primera crisis del software**.

Ejemplos de lenguajes estructurados de esta época son: **Algol**, **Pascal**, **C** y **ADA**. Aparecen también lenguajes experimentales muy específicos, como: **Lisp** y **Prolog** (para lógica e inteligencia artificial) y **Smalltalk**, el primer lenguaje orientado a objetos (*LOO – Object Oriented Language*), que incluía un enfoque y un nivel de abstracción conceptual desconocido hasta el momento.

Esta es también la época en que se define el **Modelo Relacional de Bases de Datos**, aun vigente en la actualidad.



Cuarta generación (4GL) – Años 80 en adelante

La cuarta generación de lenguajes de programación avanza en la sintaxis utilizada, intentando utilizar un lenguaje cada vez más natural. Los lenguajes de la cuarta generación se crean, típicamente, para acceder a bases de datos,

obteniéndose información mediante interrogaciones. Ejemplos de estos lenguajes son: **SQL**, generadores automáticos de aplicaciones y **herramientas CASE**.

En esta época, se afianza la informática personal, abriéndose con ello un nuevo mercado para el software.

Otra tendencia avanzada por **Smalltalk** que se consolida, es la de los Lenguajes Orientados a Objetos, como **C++**, **Eiffel** y **Java**, y los lenguajes de programación visual (*visual programming language*): **Visual Basic** y **Visual C**, que explotan las facilidades de interacción con los entornos gráficos.

Quinta generación

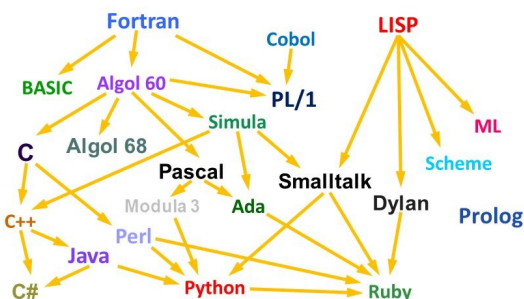
La caracterización de los lenguajes de quinta generación es algo más difusa. Se trata de lenguajes para la implementación de algoritmos que intentan imitar el resultado del razonamiento humano.

Son lenguajes orientados hacia la **inteligencia artificial (AI – Artificial Intelligence)** y el manejo de redes neuronales (*Neural Networks*). En realidad se trata de una revisión y puesta al día de lenguajes pioneros en su momento, como Lisp, Prolog y también podríamos incluir en este grupo a los **lenguajes funcionales**, como **Haskell**, **Python** y **Ruby**.

No todos los expertos tienen claro que esta generación exista claramente diferenciada de las anteriores, ya que a partir de este momento, se sucede continuamente la aparición de nuevos lenguajes.

1.4 – Técnicas de programación

A family tree of languages



Tal y como definimos en la introducción de este tema, podemos distinguir entre lenguajes de alto y bajo nivel. Dado que los primeros son los que se utilizan actualmente para programar ordenadores, tablets y smartphones, nos referiremos en exclusiva a estos en lo sucesivo.

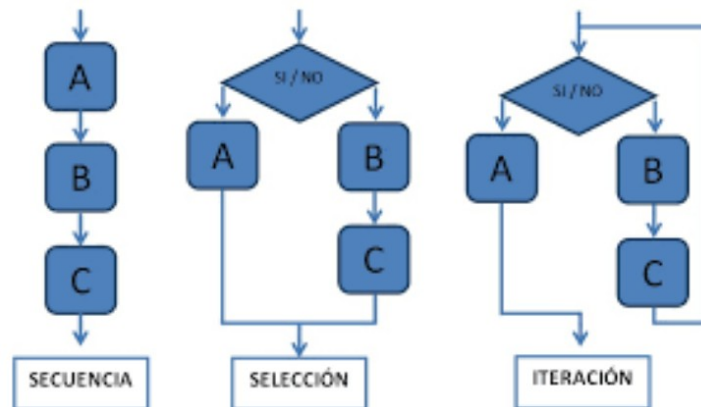
Un lenguaje de programación está constituido por:

- 1 Un **alfabeto** (*alphabet*): o conjunto de símbolos permitidos.
- 2 Una **sintaxis** (*syntax*): normas de construcción permitidas de los símbolos del lenguaje.
- 3 Una **semántica** (*semantic*): significado de las construcciones para hacer acciones válidas.

Además, según la técnica de programación utilizada (en realidad relacionadas con las distintas generaciones), los lenguajes de alto nivel se pueden clasificar en tres grupos:

- **Lenguajes de Programación Modulares y Estructurados** (*Modular & Structured programming language*):

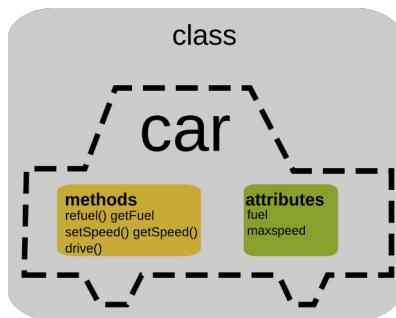
La programación estructurada se define como una técnica para escribir programas que permite sólo el uso de tres tipos de sentencias o estructuras de control: secuenciales, alternativas o condicionales y repetitivas, iteraciones o bucles (*sequencies, selections & loops*). Ejemplos de estos lenguajes son **Pascal** y **C**:



La programación estructurada evolucionó al mismo tiempo hacia la **programación modular**, que divide el programa en porciones de código llamados módulos, con una alta cohesión interna y funcionalidad concreta, que simplifican la resolución del problema por el conocido método de “divide y vencerás” (*divide & conquer*) podrán ser reutilizares.

Aunque los requerimientos actuales de software son bastante más complejos de lo que la técnica de programación estructurada es capaz de representar, sigue siendo necesario conocer sus bases, ya que a partir de ellas se evolucionó hasta otros lenguajes y técnicas más completas (orientada a eventos, orientada a objetos) que son las que se usan actualmente. Por tanto, normalmente, cuando se aprende a manejar un lenguaje de programación, se empieza por aquí, y se va avanzando en abstracción hacia los siguientes paradigmas.

- **Lenguajes de Programación Orientados a Objetos** (*Object Oriented Programming Languages*):

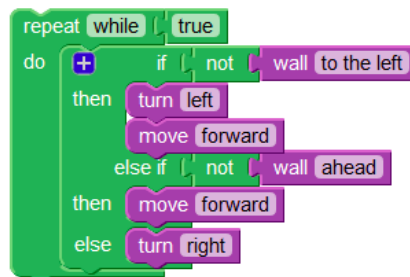


Usan la técnica de programación orientada a objetos (**OOP**), como **C++**, **Java**, **Ada**, **Delphi**, etc.

La visión que se tiene del código en la Orientación a Objetos no es la de un conjunto ordenado de instrucciones que llevan a obtener un resultado paso a paso, sino de la descripción de **un conjunto de objetos que representan una modelización del mundo real**, cada uno de ellos con una serie de características y un comportamiento, que colaboran e interactúan entre ellos para realizar acciones.

- **Lenguajes de Programación Visuales** (*Visual Programming Languages*):

Basados en las técnicas anteriores, permiten programar interactuando con un entorno gráfico, arrastrando y pegando componentes en un bloque, siendo el código correspondiente generado de forma automática en segundo plano, como ocurre con **Visual Basic.Net**, **Borland Delphi**, **Scratch**, etc.



Como curiosidad, se puede ver en helloworldcollection.de el aspecto que tiene el programa "Hello world!" en casi 600 lenguajes de programación diferentes.

1.5 – Ingeniería del Software

Primera Crisis del Software: el problema

Tal y como hemos visto en el apartado anterior, desde la construcción de los primeros ordenadores hasta la actualidad, la tarea de programar ha pasado de ser contemplada como algo secundario, a una actividad crítica y central en el desarrollo de la Informática.

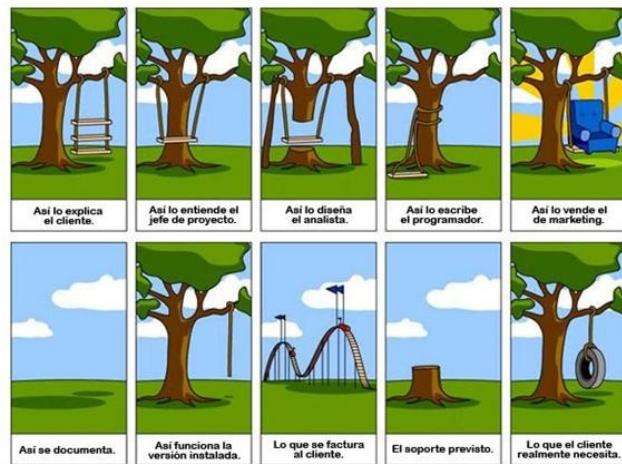
En un principio, la industria focalizaba su atención sobre el hardware: tecnologías, precio final del producto, rendimiento... y aunque todo eso sigue siendo

importante, actualmente se considera que **la mayor parte del valor del negocio de la informática está en el software y en los datos.**

Esta forma de ver las cosas cambia cuando el ordenador pasa de ser una herramienta para científicos e ingenieros y se convierte en una máquina de uso extensivo, primero empresarial, después, doméstico con la aparición de los ordenadores personales, y actualmente generalizado a cualquier actividad con los móviles y los **Smart devices** (dispositivos inteligentes). Estamos en la era del Internet de las cosas (**IoT, Internet of Things**).

Comienza ya en un primer momento, a tomarse conciencia de que la tarea de programar, en un principio vista como una actividad artesanal y dependiente de la capacidad y la intuición del programador, debe planificarse y organizarse entorno a unos estándares de diseño y documentación que permitan que el código creado por un programador o equipo de programadores pueda ser retomado, reinterpretado, adaptado, ampliado y reusado por otro.

Para expresarlo de un modo gráfico, se trata de evitar que no ocurra lo siguiente:



En definitiva, la creación de software debe someterse a las mismas exigencias y requerimientos que cualquier otra rama de la ingeniería "física": mecánica, naval, arquitectura, etc...

El punto de inflexión entre una visión y otra es conocido como **Primera Crisis del Software**. Ocurre al final de los años 60, y el primero que la enuncia como tal es **Dijkstra**, uno de los padres de la futura nueva disciplina denominada Ingeniería del Software.

El término expresa las dificultades del desarrollo de software frente al rápido crecimiento de la demanda, la complejidad creciente de los problemas a ser resueltos y la inexistencia de técnicas establecidas para el desarrollo de sistemas que funcionaran adecuadamente o pudieran ser validados.

Ingeniería del Software (*Software Engineering*): la solución

La Ingeniería del Software nace como una nueva disciplina que pretende, principalmente, obtener software de calidad, entendiendo como tal a aquel que cumple con todos los requisitos funcionales y de eficiencia previamente establecidos, y con unos estándares de desarrollo aceptados, debidamente documentados y medibles.

Se trata de esta manera de que el desarrollo de software sea mucho mas metodológico y estructurado, disminuyendo de esta manera notablemente los fallos, las imprevisiones, y las correcciones costosas.

Hay que reseñar, que pese a todos los estándares establecidos, los problemas imprevistos siempre ocurren, como en cualquier ingeniería, por lo que se habla de “reducir al mínimo” estos problemas, ya que eliminarlos totalmente es una pretensión ilusoria.

Áreas de conocimiento de la Ingeniería del Software

Entonces ¿qué conocimientos debe tener un ingeniero del software?. Según el **SWEBOK** ó *Software Engineering Body of Knowledge*, proyecto de la IEEE que trata de caracterizar el alcance de esta disciplina, se definen las siguientes áreas de conocimiento:

- **Requerimientos (*Requirements*)**: necesidades y restricciones que debe satisfacer un producto para contribuir a la solución de un problema real: obtener, cuantificar, negociar, clasificar, priorizar, modelar, documentar y validar los requerimientos de software. Implica una comunicación clara y sin ruido con el cliente
- **Diseño (*Design*)**: juega un papel crítico en el desarrollo de software. Se generan modelos que constituyen los planos para la construcción del Software. Se suele hablar de dos niveles de diseño:
 - ✓ **Diseño arquitectónico**: identifica los componentes y su interrelación “desde arriba”, al nivel más alto de abstracción. Se toman decisiones importantes que no podrán ser replanteadas posteriormente sin un alto coste adicional
 - ✓ **Diseño detallado**: describe en detalle, cada componente establecido en el diseño arquitectónico
- **Construcción (*Construction*)**: es la actividad relativa a la programación propiamente dicha, depuración de código, pruebas e integración de componentes. El código generado debe cumplir estándares de buenas prácticas (*best practices*) para que sea entendible y extensible
- **Pruebas de Software (*Software testing*)**: es la verificación del comportamiento de un programa en funcionamiento real en comparación con lo

esperado. Se seleccionan una serie de datos de prueba y sesiones de trabajo típicas y atípicas para detectar fallas. Pueden ser de diferentes tipos: de funcionalidad, de confiabilidad, de eficiencia, etc... Para una evaluación objetiva, se deben adoptar una serie de métricas estandarizadas.

- **Calidad del Software (*Software quality*)**: aplicación de técnicas estáticas para evaluar y mejorar la calidad del software. Difiere de las anteriores, en que en éstas, las técnicas utilizadas son dinámicas, ya que requieren la ejecución del software terminado, y se prueban requisitos no funcionales de calidad: seguridad, carga, estrés, etc...

IMPORTANTE: tanto el testing como el aseguramiento de la calidad es una actividad que debe introducirse desde las primeras fases (requerimientos), ya que es crítica la detección de errores e inexactitudes, cuanto más tempranamente mejor

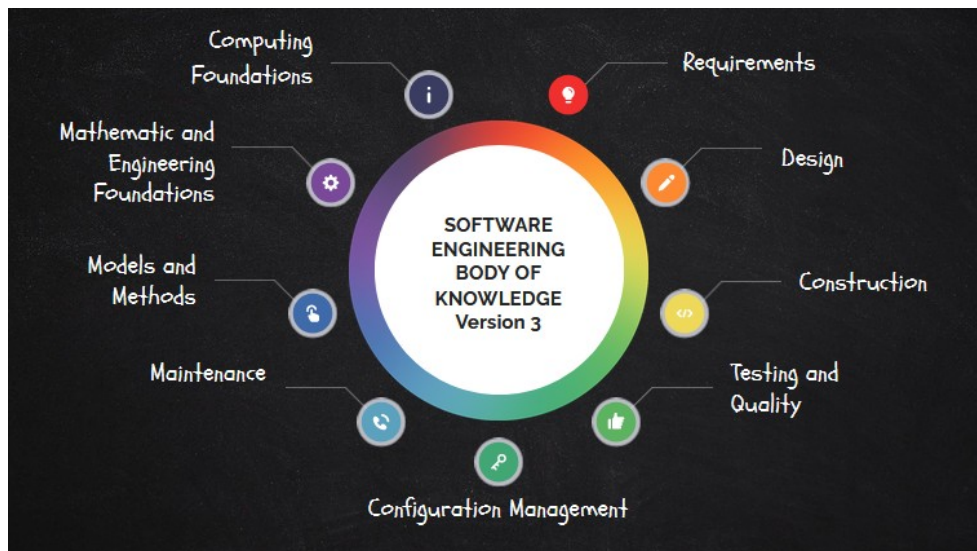
- **Mantenimiento (*Software maintenance*)**: modificaciones a un producto de software previamente liberado (*released*) para:
 - ✓ prevenir fallos (mantenimiento preventivo)
 - ✓ corregirlos (mantenimiento correctivo)
 - ✓ mejorar su funcionamiento (mantenimiento perfectivo)
 - ✓ adaptarlo a cambios del entorno (mantenimiento adaptativo)
- **Gestión de la configuración (*Configuration Management*)**: estricto control de los cambios realizados sobre todo producto obtenido durante cualquiera de las etapas del desarrollo, manteniendo siempre disponible una versión estable. Esto incluye: código ejecutable, código fuente, documentación, modelos de datos, planes de prueba, etc.

En un momento como el actual en que los cambios se producen de manera continua, esta actividad es crítica.

- **Gestión de la Ingeniería de Proyectos (*Project Management*)**. Es la aplicación de actividades administrativas para asegurar que el desarrollo y mantenimiento de software se lleva a cabo de manera sistemática, disciplinada y cuantificable. Se realizan:
 - ✓ planeación de proyectos
 - ✓ estimación de esfuerzo
 - ✓ asignación de recursos
 - ✓ administración de riesgo
 - ✓ manejo de proveedores
 - ✓ manejo de métricas
 - ✓ evaluación, y cierre de proyectos

- **Manejo de herramientas y métodos de Ingeniería de Software (*tools and methods*)**: las herramientas permiten la automatización de tareas. Las hay especializadas para asistir todas las áreas de conocimiento, desde la administración de requerimientos hasta las pruebas automatizadas. Los métodos de ingeniería de software establecen una estructura para sistematizar las actividades con el objetivo de aumentar las posibilidades de éxito.
- **Fundamentos de Computación, Matemáticos y de Ingeniería general (*Mathematic, Engineering and Computing foundations*)**

Para más información, se puede [descargar la SWEBOK Version 3](#) de la página oficial



1.6 – Ciclo de vida del software

Se denomina como **ciclo de vida del software (*software lifecycle*)** al marco de referencia que contiene los procesos, las actividades y las tareas involucradas en el desarrollo, la explotación y el mantenimiento de un producto de software, abarcando la vida del sistema, desde la definición de los requisitos, hasta la finalización de su uso.

Comprende, por tanto, 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 cada etapa, a su vez, en tareas.

Las etapas **pueden variar y combinarse de diferentes formas**, según las metodologías, pero las que suelen considerarse siempre son:

- **Análisis (Analysis)** de los requerimientos de los usuarios
- **Diseño (Design)** de las estructuras de datos, la arquitectura del software, la interfaz de usuario y los procedimientos. También se elige el lenguaje y/o el gestor de bases de datos a utilizar
- **Codificación (Codification or Construction)**, que consiste en la programación en sí, en la escritura de código, depuración, traducción y pruebas unitarias
- **Pruebas (Testing)**, en las que se comprueba el correcto funcionamiento del sistema formado por módulos ya codificados
- **Mantenimiento (Maintenance)**, que ocurre después de la entrega del software al cliente. También recibe el nombre de **soporte (Software Support)**. Actualmente, la tendencia es a considerarlo más como un **despliegue continuo** de nuevas modificaciones y funcionalidades

Además, cabe resaltar que una de las tareas más importantes presente en cada una de estas etapas es la **generación de documentación** que siempre debe acompañar a cualquier producto de ingeniería.

Modelos de ciclo de vida del software

Aunque el consenso en la existencia de estas etapas básicas es generalizado, existen diversos modelos de ciclo de vida que se pueden tomar como referente en la construcción de software.

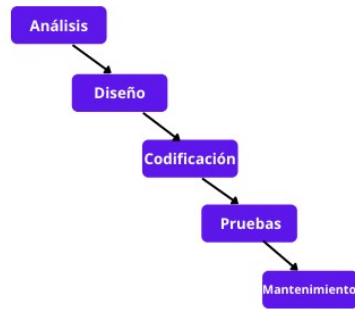
Los veremos en el mismo orden en que aparecieron históricamente, aunque hay que reseñar que, salvo el primero, todos permanecen vigentes:

1 Modelo en Cascada o lineal (*waterfal lifecycle*):

Es el modelo de vida clásico del software. Consiste en recorrer **en secuencia** los pasos anteriormente citados. Se cierra una etapa, y se pasa a la siguiente.

Es muy difícil que se pueda utilizar a día de hoy, ya que da por hecho que se conocen de antemano todos los requisitos del sistema, y que sólo dispondremos del producto al finalizar la última fase.

Sólo es aplicable a pequeños desarrollos y en entornos en que la incertidumbre sea inexistente, porque las etapas pasan de una a otra sin retorno posible. Se presupone, por tanto que no habrá errores ni variaciones en los requerimientos del software, algo muy temerario y no suele cumplirse nunca:

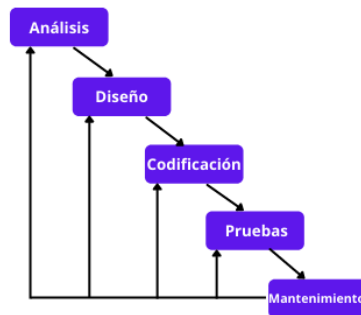


2 Modelo en Cascada con Realimentación (*waterfall lifecycle with feedback*)

Es uno de los modelos más utilizados.

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

Es el modelo de ciclo de vida perfecto, si el proyecto es rígido (pocos cambios, poco evolutivo) y los requisitos están claros desde el primer momento, de forma incluso contractual. Es típico en aplicaciones de la administración pública y/o estratégicas.



3 Modelos Evolutivos

Son más modernos que los anteriores. Tienen en cuenta la naturaleza cambiante y evolutiva del software en cuanto a requisitos. Distinguimos dos variantes:

3.a Modelo Iterativo e Incremental (*iterative and incremental lifecycle*)

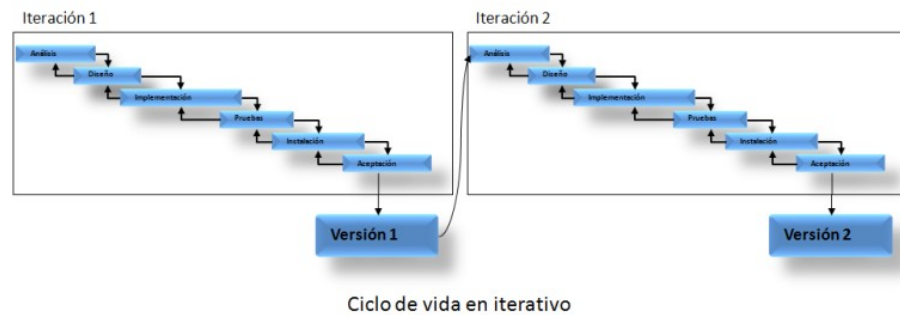
Está basado en el modelo en cascada con realimentación, donde las fases se repiten y refinan, y van propagando su mejora a las fases siguientes.

- Es **incremental**, porque se desarrolla el producto dividiéndolo en partes o funcionalidades al abordarlo, para después integrarlas a medida que se completan. Podemos tener una aplicación funcional a las pocas

iteraciones, pero a cada vuelta se agregan nuevas funcionalidades al sistema

- Es **iterativo**, por que a cada iteración se revisa, evalúa y mejora el producto completo, lo que incide en su calidad final

Por tanto, de la unión del ciclo de vida iterativo y el incremental al final de cada iteración se consigue una versión más estable del software, de más calidad, y añadiendo además nuevas funcionalidades respecto a versiones anteriores.



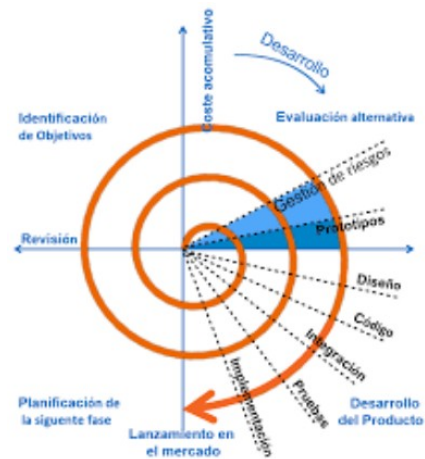
3.b Modelo en Espiral (*spiral lifecycle model*)

En él, el software se va construyendo repetidamente **en forma de versiones**, comenzando por la de riesgos más asumibles o también, el mínimo producto viable (**MVP**), y se hace una iteración en espiral, que consiste en nuevas versiones cada vez mejores, debido a que incrementan y/mejoran las funcionalidades ya implementadas.

El proceso puede continuar más o menos indefinidamente si es que el cliente quiere seguir haciendo mejoras en el software. A cada vuelta se vuelven a evaluar las diferentes alternativas, hasta que el producto software desarrollado sea aceptado por el cliente.

0.1 **Ventaja:** como el software evoluciona a la vista del cliente a medida que progresa el proceso, el desarrollador comprende mejor el producto y reacciona mejor ante los riesgos.

0.2 **Inconveniente:** la flexibilidad conlleva incertidumbre de los plazos temporales y en el presupuesto económico. No se sabe cuando el proceso acabará, ni lo que acabará costando.



4 Modelo Agile

Aparece como evolución natural de los anteriores (iterativo e incremental y espiral), y responde al siguiente *Manifiesto*, firmado en 2001 por 17 expertos en el desarrollo de software, críticos con la forma tradicional en que se desarrollaba su trabajo:

“Estamos descubriendo formas mejores de desarrollar software tanto por nuestra propia experiencia como ayudando a terceros. A través de este trabajo hemos aprendido a valorar:

Individuos e interacciones sobre procesos y herramientas
Software funcionando sobre documentación extensiva
Colaboración con el cliente sobre negociación contractual
Respuesta ante el cambio sobre seguir un plan

Esto es, aunque valoramos los elementos de la derecha, valoramos más los de la izquierda”



Interpretando un poco lo que significa cada afirmación del manifiesto:

- Los procesos deben ser una ayuda para guiar el trabajo, pero no encorsetarlo y ser un obstáculo. Es vital que **las personas con experiencia propongan iniciativas de cambio**, aunque supongan salirse del marco de referencia
- Aunque los documentos son necesarios y permiten la transferencia de conocimiento, no pueden sustituir, ni pueden ofrecer la riqueza que se logra con la **comunicación directa entre las personas** y a través de la **interacción de éstas con los prototipos**

- En el desarrollo ágil, **el cliente es un miembro más del equipo**, que se integra y colabora en el grupo de trabajo. Los modelos de contrato por obra no encajan.
- Para entornos inestables, que cambian y evolucionan rápidamente, resulta mucho más valiosa la **capacidad de respuesta frente al cambio**, que la de seguimiento de planes pre-establecidos.

Como principales **ventajas** de esta metodología, que es la que más se utiliza en el presente y futuro inmediato, se pueden indicar:

- ✓ extrema flexibilidad en el proceso
- ✓ tiempo récord en que se puede tener una versión temprana del producto funcionando (mínimo producto viable)
- ✓ interacción y realimentación continua con el cliente que inciden en una mejora en la calidad del producto y en la satisfacción de éste (validación desde el principio)
- ✓ Cuando menos claramente definido esté el producto inicialmente, mayor será el beneficio.

En cuanto a los **inconvenientes**, el principal es que al no existir un plan concreto, no hay certezas absolutas en la planificación del proyecto, ni en los plazos de entrega ni en los presupuestos.

Además todas las partes implicadas, especialmente el cliente debe mentalizarse en la necesidad de emplear mucho tiempo en reuniones e intercambios de contenido, para lo cual es incluso necesario recibir una formación específica.



Finalmente, insistir en el hecho de que este modelo de desarrollo supone una evolución directa del modelo en espiral, pero prestando una especial atención a la comunicación permanente cliente-desarrollador.

1.7 – Principales metodologías de desarrollo

El concepto de **Metodología de Desarrollo de Software** abarca un poco a todo lo visto anteriormente, bajo un determinado enunciado y formalismo que determina todos los procesos y pasos.

Se trata en realidad de:

- una **colección de procedimientos** y técnicas
- encaminada a organizar el desarrollo de software mediante una serie de **buenas prácticas** (*best practices*)
- **utilizando una serie de herramientas** determinadas
- articuladas según un modelo determinado de ciclo de vida
- que genera una **documentación de acuerdo a un formato estandarizado**

Por tanto, una metodología particular:

- refleja un estilo de hacer las cosas
- organiza de forma clara el trabajo a realizar
- indica detalles del proceso, y de la documentación que debe generarse

Acogerse a una metodología particular supone recorrer un camino ya trazado y beneficiarse de la experiencia previa de un grupo de personas expertas.

Básicamente, sea cual sea la metodología adoptada, los procesos son los mismos (análisis, diseño,...), pero la forma de organizar el trabajo y los focos de atención, son diferentes.

Por ejemplo, en el desarrollo de cualquier software es necesario hacer un **análisis de requisitos**, que conlleva recolectar información del cliente y/o usuarios finales sobre lo que se desea y necesita implementar, y sobre las funcionalidades generales y específicas del producto.

Pues bien, una metodología en particular nos especifica **cómo y cuando desempeñar esta función concretamente**, por ejemplo, nos indicará que tenemos que realizar entrevistas con determinadas personas del ámbito en el que se desenvolverá el software generado por nosotros, como hay que preparar las preguntas y **que forma tendrá el documento que tendremos que elaborar a partir de la información recabada**, o tal vez lo indique de una manera completamente diferente, según las buenas prácticas que guíen dicha metodología: mediante fichas de “historias de usuario”, utilizando técnicas de Design Thinking, etc...

Y así en cada una de las fases del proyecto.

Las metodologías, como todo en informática, han sufrido una evolución a lo largo del tiempo. Hay toda una serie de ellas anteriores a los años 90, que si bien

gozaron de mucho prestigio, en la actualidad se consideran obsoletas. Es el caso de **Jackson, Yourdon, Merise**,...

Absolutamente **planificadoras, sistemáticas, y orientadas a** la descripción y regulación del **proceso**, estas metodologías tienen una serie de pasos muy estructurada y generan un tipo de documentación que adolece de una marcada rigidez. Como ya dijimos anteriormente, sólo tienen cabida en entornos en los que la incertidumbre está muy controlada o debe ser inexistente, y los plazos y métodos responden a pautas marcadas contractualmente.

A partir del año 2000, aparecen una serie de metodologías más modernas, como **Rational Unified Process** (RUP), y otras que siguen un enfoque más acorde con el manifiesto ágil y sus propuestas. Es el caso de **Scrum, Extreme Programming** (XP), y **Agile Unified Process** (AUP).

Éstas metodologías, al contrario de lo que sucedía con las clásicas, se adaptan a entornos cambiantes y con necesidades de respuesta rápida, más acordes con las necesidades del mercado tecnológico actual.

Nosotros, a lo largo del desarrollo de este módulo, estudiaremos en particular una de cada tipo: son el caso de **Scrum, RUP** y de **Métrica v3**, que pese a pertenecer esta última al tipo de las metodologías clásicas, sigue vigente en España (en una versión actualizada, la v3) por ser la que utiliza la Administración Pública española en sus proyectos de informatización.

Por último, **no debemos confundir lo que es una metodología de desarrollo con otras cosas**, por ejemplo:

- un paradigma de programación (estructurada, orientada a objetos, visual),
- un tipo de ciclo de vida (cascada, iterativo, espiral,...)
- ni con otro tipo de creaciones como el lenguaje gráfico UML

Aunque hay que tener claro que en realidad, toda metodología integra estos conceptos anteriores (describen un ciclo de vida, son orientadas a programar de acuerdo con un determinado paradigma, y pueden utilizar diferentes diagramas basados en el lenguaje UML) dentro de un marco de actuación y con unas determinadas herramientas y buenas prácticas específicas.

Y aunque sea un poco prematuro, ya que le dedicaremos mucho más tiempo y espacio en este módulo, pasaremos a describir someramente qué es UML.



UML (Unified Modeling Language) es un **lenguaje gráfico**, esto es, una forma estandarizada de representar partes de la construcción de un sistema de software (diseño, comportamiento, arquitectura, etc.), con diagramas gráficos que siguen unas determinadas convenciones estandarizadas.

No es una metodología, pues no indica la manera de realizar las tareas en un proyecto concreto, sino un conjunto de **representaciones gráficas** utilizadas por diferentes metodologías. De hecho, por ejemplo, dos metodologías tan distintas como RUP ó Métrica utilizan diagramas UML como herramientas para expresar estructuras e ideas en las distintas fases del desarrollo de software.

1.8 – Licencias de software. Software libre y propietario

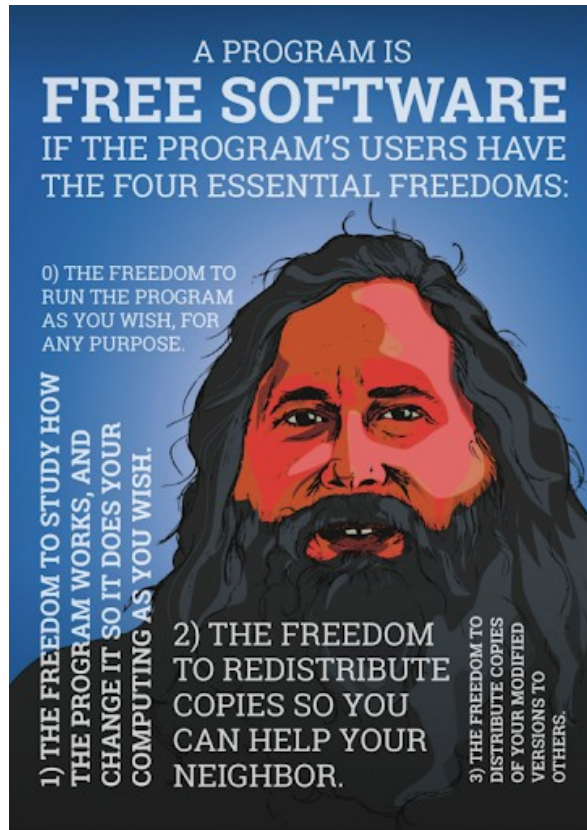
Una **licencia de software** (*software license*) es un contrato que se establece entre el desarrollador de un software sometido a propiedad intelectual y a determinados derechos de autor, y el usuario, en el que se definen con precisión los derechos y deberes de ambas partes.

Es el desarrollador, o aquel a quien éste haya cedido los derechos de explotación, quien elige la licencia según la cual distribuye el software.



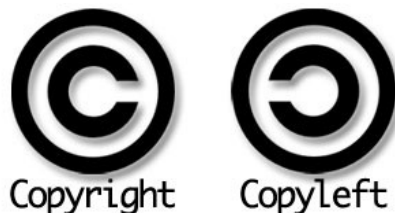
El **software libre** (*free software*) es aquel en el cual el autor cede una serie de derechos (denominados libertades) en el marco de una licencia, y que son las siguientes:

- 1 Libertad de utilizar el programa con cualquier fin en cuantos ordenadores se desee
- 2 Libertad de estudiar cómo funciona el programa y de adaptar su código a necesidades específicas. Para ello, como condición previa, es necesario poder acceder al código fuente
- 3 Libertad de distribuir copias a otros usuarios (con o sin modificaciones)
- 4 Libertad de mejorar el programa (ampliarlo, añadir funciones) y de hacer públicas y distribuir al público las modificaciones. Para ello, como condición previa, es necesario también poder acceder al código fuente



La licencia más completa que se puede dar en los productos y desarrollos de software libre (pero no la única) es la **licencia GPL (GNU General Public License** – Licencia Pública General) que da derecho al usuario a usar y modificar el programa con la obligación de hacer públicas las versiones modificadas de éste.

A partir de la licencia GPL, han aparecido toda una familia de licencias, en las que se restringe alguna de las libertades: **EUPL** (European Union PL), **EPL** (Eclipse PL), **MPL** (Mozilla PL), etc...



En el otro extremo, a diferencia del software libre y de fuente abierta, el **software propietario (proprietary software)** es aquel que se distribuye habitualmente en formato binario (o bytecode), sin posibilidad de acceso al código fuente y según una licencia en la cual el propietario prohíbe todas o algunas de las siguientes posibilidades: redistribución, modificación, copia, uso

en varias máquinas simultáneamente, transferencia de la titularidad, difusión de fallos y errores que se pudieran descubrir en el programa, etc.

Finalmente, un **software de dominio público** (*public domain software*) es aquel que carece de cualquier tipo de licencia o no hay forma de determinarla pues se desconoce al autor. Esta situación se produce cuando el propietario abandona los derechos que lo acreditan como titular, o bien cuando se produce la extinción de la propiedad por la expiración del plazo de la misma. El software de dominio público no pertenece a una persona concreta, sino que todo el mundo lo puede utilizar, e incluso cabe desarrollar una oferta propietaria sobre la base de un código que se encuentra en el dominio público.

Licencias Creative Commons

Creative Commons es una organización sin ánimo de lucro, con sede en Mountain View, California, dedicada a la ampliación del repertorio de licencias disponibles **sobre cualquier obra creativa**, (música, gráfica, audiovisual, informática,...) que le otorga a un autor el poder de decidir sobre los límites de uso y explotación de su trabajo o creaciones en Internet.

Esta organización ha publicado diversos derechos de la licencias de autor conocidas como **Licencias Creative Commons** gratuitamente para su adopción por cualquier autor. Mediante el uso de estas licencias, se les permite a los creadores comunicar los derechos que se reservan y los derechos a los que renuncian en beneficio de los destinatarios o de otros creadores. Se utilizan una serie de símbolos visuales asociados, que explican los detalles específicos de cada licencia de Creative Commons.



Las licencias Creative Commons permiten al autor cambiar fácilmente los términos y condiciones de derechos de autor de su obra, en algún punto intermedio entre del extremo “todos los derechos reservados” a “algunos derechos reservados”, situando su obra en algún punto intermedio entre el copyright y copyleft.

Existen 6 modelos de licencia:



Reconocimiento (by): Se permite cualquier explotación de la obra, incluyendo una finalidad comercial, así como la creación de obras derivadas, la distribución de las cuales también está permitida sin ninguna restricción.



Reconocimiento – NoComercial (by-nc): Se permite la generación de obras derivadas siempre que no se haga un uso comercial. Tampoco se puede utilizar la obra original con finalidades comerciales.



Reconocimiento – NoComercial – CompartirIgual (by-nc-sa): No se permite un uso comercial de la obra original ni de las posibles obras derivadas, la distribución de las cuales se debe hacer con una licencia igual a la que regula la obra original.



Reconocimiento – NoComercial – SinObraDerivada (by-nc-nd): No se permite un uso comercial de la obra original ni la generación de obras derivadas.



Reconocimiento – CompartirIgual (by-sa): Se permite el uso comercial de la obra y de las posibles obras derivadas, la distribución de las cuales se debe hacer con una licencia igual a la que regula la obra original.



Reconocimiento – SinObraDerivada (by-nd): Se permite el uso comercial de la obra pero no la generación de obras derivadas.

Podemos observar, que como mínimo, todas incluyen el **reconocimiento de la autoría**, esto es, se debe indicar claramente quien es el autor al utilizar el producto.

En el ámbito informático, **Arduino**, **Wikipedia** y **Mozilla** son ejemplo de creaciones exitosas y muy extendidas, que adoptaron en su momento licencias Creative Commons.

1.9 – Algunos conceptos adicionales

Si bien en los apartados anteriores hemos hecho un repaso *grosso modo* sobre todo lo relativo al desarrollo de software, hay una serie de conceptos importantes por los que hemos pasado de puntillas o directamente se nos han quedado en el tintero. Pasemos a revisarlos a continuación:

Máquinas virtuales (*Virtual Machines*)

Una máquina virtual es un software que simula el comportamiento de un software base determinado y que es capaz de ejecutarse “sobre” otro software básico, que es el sistema operativo real (no virtual), el que realmente se comunica con el hardware.

Hay dos tipos de máquinas virtuales: **de sistemas y de procesos**.

Una máquina virtual de sistemas simula el comportamiento de un sistema operativo (el virtual) sobre otro sistema operativo (el real). La ventaja que ofrece es permitir la portabilidad de programas y/o aplicaciones que serían incompatibles por sí mismos. El principal inconveniente es la ralentización de los procesos que corren sobre ellas, debido a que tienen que atravesar capas adicionales de software.

De esta manera, se puede tener instalada una máquina virtual Windows o Android sobre un sistema operativo Mac OS en un ordenador Apple, por ejemplo, pudiéndose de esta manera correr aplicaciones nativas que sin este recurso serían, en principio, incompatibles.

Algunas de las máquinas virtuales más utilizadas son **Virtual Box** y **VMWare**, ambas disponibles gratuitamente en la actualidad.



Pero no todas las máquinas virtuales son de este tipo. Están también las denominadas **MV de proceso o aplicación**. Es el caso de Java, como ya vimos anteriormente en este tema.

Permiten ejecutar aplicaciones que se comportan de forma igual en plataformas hardware diferentes, como Windows, Mac o Linux. La más utilizada actualmente es la **Máquina Virtual Java (JVM)**, que permite interpretar y ejecutar el bytecode ya pre compilado sobre cualquier sistema operativo.



PROXMOX

Finalmente, debemos nombrar a **Proxmox**, que es un software gratuito de código abierto funcionando sobre Debian GNU/Linux y que permite la gestión de máquinas virtuales desde un entorno cliente/servidor, con la ventaja adicional de ofrecer una interfaz web.

Su forma de trabajar la virtualización es radicalmente distinta a las anteriores. Se necesita disponer de un servidor de gran potencia que aloje y administre las máquinas virtuales. Desde los equipos conectados en red y autorizados, se podrá solicitar el acceso a una o más máquinas cada vez que necesiten utilizarla. No hay que instalar nada en el ordenador cliente.

Entornos de ejecución (*Execution environments*)

Un Entorno de ejecución está formado por la **máquina virtual y una serie de 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.



Un ejemplo de esto es el **JRE (Java Runtime Environment)**. Otro ejemplo a destacar sería el **Common Language Runtime de .NET**.

Contenedores de software (*Software containers*)

Un contenedor de software es un paquete de software que envuelve una aplicación en un completo sistema de archivos con **estrictamente sólo lo que hace falta para su ejecución**.

Representa, por tanto, un **entorno de ejecución completo**, un paquete de elementos que además de la aplicación en si, contendrá sus dependencias, así como las librerías y ficheros binarios y de configuración necesarios para el buen funcionamiento de esta.

El concepto de contenedor es similar al de la máquina virtual, salvo que en lugar de contener un sistema operativo completo, **sólo incluye lo imprescindible para garantizar la portabilidad de un software determinado**. El contenedor actúa de esta forma, como una “carcasa” para el software, que lo habilita para funcionar dentro de un nuevo entorno de software ajeno.

Dependiendo del contexto y de la situación particular, puede ser más recomendable utilizar una solución u otra, pero en general, un contenedor suele ser más compacto y eficiente. Además, el contenedor es el formato utilizado para desplegar aplicaciones escalables en la nube.

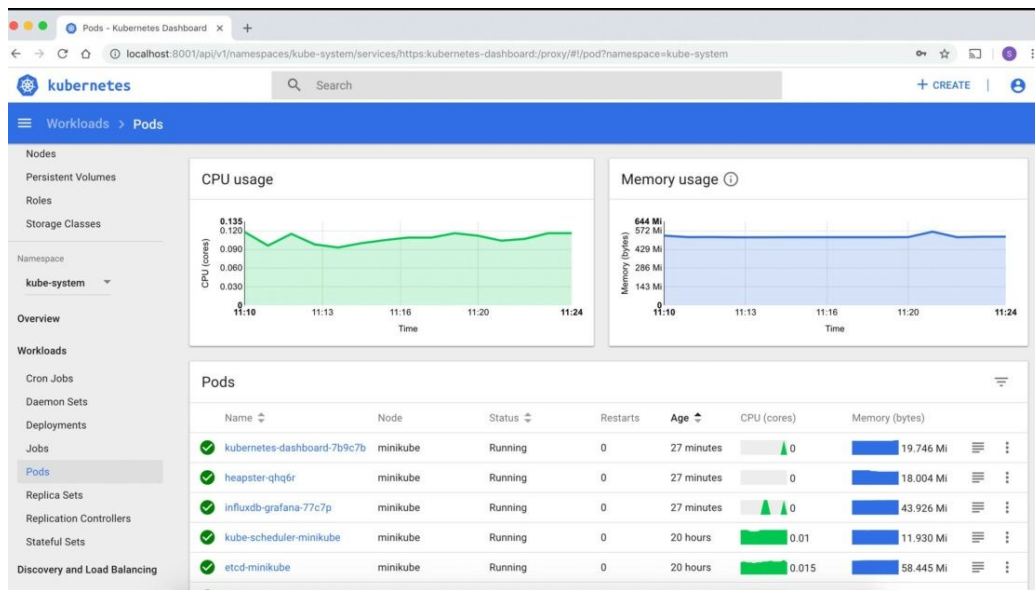


Docker es el formato de contenedor más usado en la actualidad. Sólo necesitamos tener instalado el software docker sobre nuestro sistema operativo para poder ejecutar cualquier aplicación empaquetada en un contenedor de este tipo.

Kubernetes es un orquestador de contenedores, un gestor que controla el despliegue de aplicaciones en la nube en diferentes servidores o **hosts**. Es un proyecto de software libre creado por Google y The Linux Foundation.

Los servicios que ofrece son:

- gestión del ciclo de vida de los contenedores,
- monitorización de red y chequeo
- escalado del número de contenedores en ejecución, ampliando o reduciendo su número.



Frameworks

Se suele utilizar el término inglés, aunque se puede traducir como **plataforma o entorno de trabajo**. Se trata de una estructura módulos concretos de software, que a su vez se utiliza como base para la organización y desarrollo de software en un lenguaje determinado.

Suele incluir:

- soporte de programas
- bibliotecas
- herramientas para el desarrollo rápido de código
- una determinada organización o arquitectura predefinida (organización en paquetes o carpetas) que se considera óptima para el desarrollo de ese lenguaje en particular, y.

Se pretende de esta manera ayudar automatizar y ayudar a desarrollar y unir los diferentes componentes de un proyecto. Se trata de no partir desde cero en cada

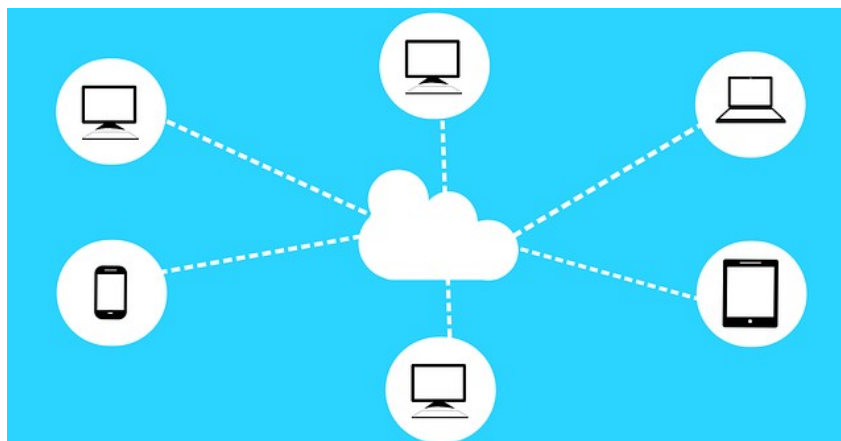
nuevo proyecto de software, habida cuenta de que hay siempre un gran número de módulos y operaciones básicas que son comunes a todos.

Algunos de los frameworks más populares son **Spring** e **Hibernate** para Java, **Ruby on Rails** para Ruby, **Django** para Python, **Symfony** y **Laravel** para PHP, y **.NET** de Microsoft. **Bootstrap**, **Tailwind** y **Compozer** son frameworks para desarrollo de sitios web e **ionic** es una herramienta framework, gratuita y open source, para el desarrollo de aplicaciones híbridas basadas en HTML5, CSS y JS.



Computación en la nube (*Cloud Computing*)

La computación en la nube consiste en la disponibilidad de servicios de información y aplicaciones basados en Internet y **Data Centers** remotos. La “nube” es una metáfora simbólica de la propia red de redes, que se suele representar como tal en los diagramas de topologías de red.



Picture from [Pixabay](#)

Tanto los usuarios particulares como las empresas pueden, a través de estos servicios, y sin disponer de infraestructuras locales de hardware especialmente costosas, gestionar bases de datos o hacer uso de aplicaciones sin necesidad de instalarlas en el computador. Solo es necesario de disponer de una conexión a Internet, un punto de acceso, y una cuenta, que puede ser una computadora o cualquier dispositivo móvil. En el otro extremo del servicio, se encuentra un Servidor, que puede ser gratuito (en realidad no lo es, nada lo es...) o de pago.

Entre estas distintas formas que puede adoptar la nube se encuentran:

- ✓ **SaaS - *Software-as-a-Service***: es cualquier servicio basado en la web, como por ejemplo Gmail, Google Docs, Dropbox, etc. A estos servicios nosotros se accede a través del navegador, y todo el desarrollo, mantenimiento, actualizaciones, copias de seguridad son responsabilidad del proveedor de servicios
- ✓ **PaaS - *Platform-as-a-Service***: consiste en el desarrollo de aplicaciones que se ejecutan en la nube. Nuestra única preocupación es la construcción de una aplicación, y la infraestructura nos la da la plataforma en la nube. Reduce bastante la complejidad a la hora de desplegar y mantener aplicaciones ya que la escalabilidad se gestiona automáticamente. Son ejemplos de Paas: Heroku, Google App Engine.
- ✓ **IaaS - *Infrastructure-as-a-Service***: permite mayor control que PaaS, a cambio de tener que encargarnos directamente de la gestión de infraestructura. A esta figura responde **Amazon Web Service (AWS)** que nos permite manejar maquinas virtuales en la nube o almacenamiento. Se basa en la creación y el despliegue de contenedores. Se puede elegir y gestionar qué tipo de instancias queremos usar, Linux o Windows, así como la capacidad de memoria o procesador de cada una de nuestras maquinas. El hardware resulta completamente transparente, todo lo manejamos de forma virtual.

Una principal ventaja de utilizar computación en la nube es también su principal inconveniente: la independencia de una instalación local particular, la hace, por contra, muy dependiente de la conexión a Internet. Si ésta última falla, todo se desmorona.

Otra ventaja fundamental es la reducción de la inversión en adquisición de software, así como la delegación de responsabilidades en cuanto a labores de respaldo y ciberseguridad.

Como ventaja adicional y muy interesante, cabe destacar que las aplicaciones SaaS posibilitan el trabajo colaborativo en equipo independiente de la localización geográfica. En particular, permite que determinadas herramientas de desarrollo de software hagan uso de esta característica,

Es el caso de herramientas que utilizaremos en este módulo para: diseño UML (**LucidChart**, **Visual Paradigm**), IDE's (**Eclipse Che**), herramientas de gestión de

proyectos (como **Trello**, **Jira**, **Slack**, **Teams**), repositorios (como **GitHub**, **Gitlab**, **DockerHub**), etc...

Devops (Development-Operation)

Finalmente, tenemos que hablar de un concepto que hasta hace muy poco se consideraba novedoso, pero que cada día se está implantando más en relación con la creación de Software.

Tradicionalmente, se han distinguido dos tipos de tareas realizadas por el departamento de informática o IT:

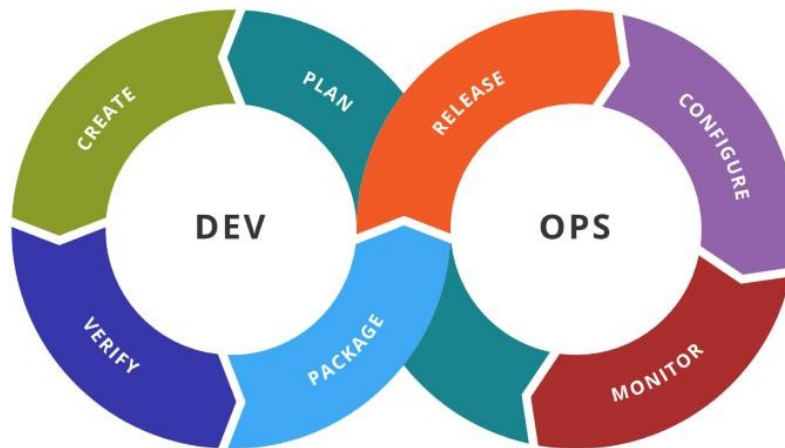
- ✓ **Desarrollo:** construcción de nuevas aplicaciones
- ✓ **Operación:** administración y explotación de las aplicaciones ya existentes

Es muy habitual que en un departamento de informática existan equipos de personas diferenciados tanto en formación como en cualificación para la realización de tareas de uno u otro tipo.

DevOps es una filosofía de trabajo consistente en poner en colaboración, comunicación y coordinación continuas ambas tareas, de manera que los límites entre una y otra queden difuminados, y se produzca de esta manera una realimentación y una sinergia que favorezca a su vez una mejora en el rendimiento y la velocidad de respuesta ante el cambio.

Tradicionalmente, el desarrollo de una aplicación conlleva una fase de despliegue y explotación que escapa de las competencias de los desarrolladores y a la inversa, los responsables de explotación y administración, se sitúan alejados de las actividades de desarrollo, teniéndose entre ambos equipos, como única vía de comunicación, el reporte de fallos e inexactitudes para la realización de tareas de mantenimiento del software.

La rápida evolución en el campo del desarrollo software y la necesidad de realización de continuas actualizaciones (**upgrades**), así como la implantación de metodologías ágiles, han desembocado de manera natural en una necesidad de sinergia entre ambos aspectos de los profesionales IT. Sirva como ejemplo el mundo de las Apps y de las redes sociales, como Facebook, Twitter, Instagram,... que están sometidas a mejoras y actualizaciones diarias



Los principales principios en los que se asienta DevOps son:

- ✓ **Cultura** : comportamiento en equipo
- ✓ **Automatización**: realizar la mayor parte del trabajo de manera automatizada
- ✓ **Métricas** del resultado para corregir y mantener la calidad
- ✓ **Compartición** de información entre desarrollo y operación