

TEMA 1 - DESARROLLO DE SOFTWARE

(SOFTWARE DEVELOPMENT)

- 1.1 – CONCEPTOS BÁSICOS
- 1.2 – TRADUCTORES: INTÉRPRETES, COMPILADORES Y 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



1.1 – INTRODUCCIÓN. CONCEPTOS BÁSICOS

Es de sobra conocido que en informática se distinguen dos componentes diferenciados de un ordenador: hardware y software.

El **hardware** lo constituyen todas las partes meramente maquinales: el ordenador en su conjunto, cada una de las piezas que lo conforman, los periféricos, etc...

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

Pero conviene aclarar que, según su función se distinguen **tres tipos de software**:

- sistema operativo ó software básico
- software para el desarrollo de programación
- aplicaciones de usuario final

Veamos de qué hablamos en cada caso particular:

1- Sistema operativo (*operating system*) es el software base 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, Windows Phone, etc...

2- 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, Microsoft Project, o Trello para la planificación de tareas, o VS Code e IntelliJ, para la escritura, depuración y testeo de código, GIT para el control de versiones, toda la gama de productos Rational de IBM, etc...

3- Aplicaciones informáticas (*computer or mobile applications*) son programas que tienen una utilidad 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 participar de una red social.

En este tema 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.

A lo largo de esta primera unidad vamos a aprender los conceptos fundamentales en torno al software y las fases del llamado **ciclo de vida de una aplicación informática** (*application lifecycle*).

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

A lo largo de la asignatura, estudiaremos a fondo los distintos tipos de **herramientas de software** para el desarrollo de software, y dedicaremos bastante tiempo en instalar configurar y manejar algunos de los más utilizados en el entorno productivo.

Por otra parte, el software básico será objeto de estudio en otros módulos de este ciclo formativo.

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 ejecutables por un ordenador**. También se suele decir que un programa es un **algoritmo codificado en un lenguaje que entienden los ordenadores**.

Algoritmo (*algorithm*)

Un algoritmo es un conjunto de órdenes o pasos ordenados y finitos que describen cómo 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 que entienda una máquina.

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 forman 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**, o pequeñas aplicaciones.

NOTA: Normalmente, en el lenguaje cotidiano no se distingue entre aplicación y programa; en nuestro caso entenderemos que la aplicación es un software completo que cumple la función completa para la que fue diseñado, mientras que un programa es el resultado de ejecutar un cierto código entendible por el ordenador, y que podría ser perfectamente un único componente o también uno de los muchos que conforman en conjunto una aplicación.

Lenguajes de programación (*programming languages*)

Para el desarrollo o construcción de un programa se utiliza 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 ya definida.

Esa sintaxis está diseñada para posibilitar la transmisión de instrucciones a la CPU, por lo que no contiene ambigüedades, y suele ser bastante exigente.

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 comprensible por el microprocesador.





















Los lenguajes de programación se han clasificado históricamente en dos grandes grupos:

- **Lenguajes de bajo nivel** (*low-level language*): dependen absolutamente de la arquitectura de cada máquina
 - o Lenguajes máquina (*machine language*): prácticamente ya no se utilizan, ni siquiera en electrónica. Consiste en programar directamente en 0's y 1's
 - o Lenguaje Ensamblador (*assembly language*): se sustituyen operandos y operaciones por nombres simbólicos, que deben ser posteriormente traducidos a 0's y 1's
- **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](https://index.tio.be/) podremos saber en cada momento la popularidad y evolución en el uso de los lenguajes de programación de alto nivel.

En la fecha actual (septiembre de 2025), encabezan el ranking, por este orden: Python, C++, C, Java, C#, JavaScript, Visual Basic, Go, Delphi y Perl. Este índice se basa en las consultas que realizan los usuarios en buscadores y foros.

Sep 2025	Sep 2024	Change	Programming Language	Rating	Change
1	1		 Python	25.98%	+5.81%
2	2		 C++	8.80%	-1.94%
3	4	▲	 C	8.65%	-0.24%
4	3	▼	 Java	8.35%	-1.09%
5	5		 C#	6.38%	+0.30%
6	6		 JavaScript	3.22%	-0.70%
7	7		 Visual Basic	2.84%	+0.14%
8	8		 Go	2.32%	-0.03%
9	11	▲	 Delphi/Object Pascal	2.26%	+0.49%
10	27	▲	 Perl	2.03%	+1.33%
11	9	▼	 SQL	1.86%	-0.08%
12	10	▼	 Fortran	1.49%	-0.29%
13	15	▲	 R	1.43%	+0.23%
14	26	▲	 Ada	1.27%	+0.56%
15	13	▼	 PHP	1.25%	-0.20%
16	17	▲	 Scratch	1.18%	+0.07%
17	21	▲	 Assembly language	1.04%	+0.05%
18	14	▼	 Rust	1.01%	-0.31%
19	12	▼	 MATLAB	0.98%	-0.49%
20	18	▼	 Kotlin	0.95%	-0.14%

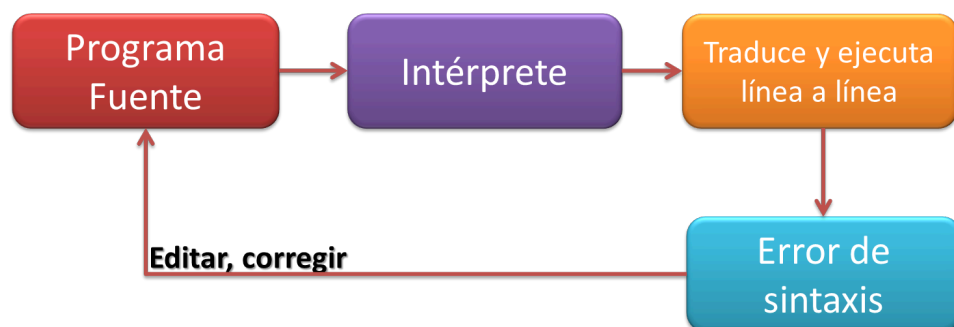
1.2 – TRADUCTORES: INTÉRPRETES, COMPILADORES Y JAVA

Los traductores son programas que traducen los programas fuente (código fuente) 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), 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 en lenguaje máquina y ejecuta cada orden una vez que se traduce sin almacenar el resultado de esa traducción, las instrucciones binarias.

En cierto sentido, su trabajo es similar al de un traductor humano que intermedia 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 suelen tener unas reglas sintácticas más relajadas.

Otra ventaja que se les suele atribuir a estos lenguajes, es la relativa portabilidad del código: al ser fuente, no está comprometido con ninguna arquitectura hardware particular.

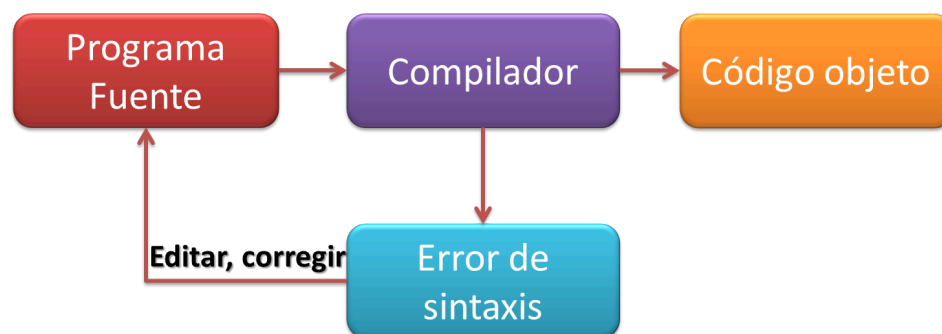
El inconveniente principal es la lentitud de ejecución, ya que cada vez que se quiere ejecutar un programa, éste debe ser traducido de nuevo línea a línea.

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, a no ser que resida en el lado servidor en un entorno de ejecución web. 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 alguna adecuación más.



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 de sus módulos mediante un programa montador o enlazador (*Linker*). Este proceso genera un programa en código máquina directamente ejecutable.

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

1. Escritura del programa fuente mediante un editor y almacenaje en memoria secundaria o en la nube.
2. Compilación
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 (*linker*) y obtención de programa ejecutable.
6. Ejecución del programa (si no existen errores en la lógica del programa, se tendrá la salida adecuada)

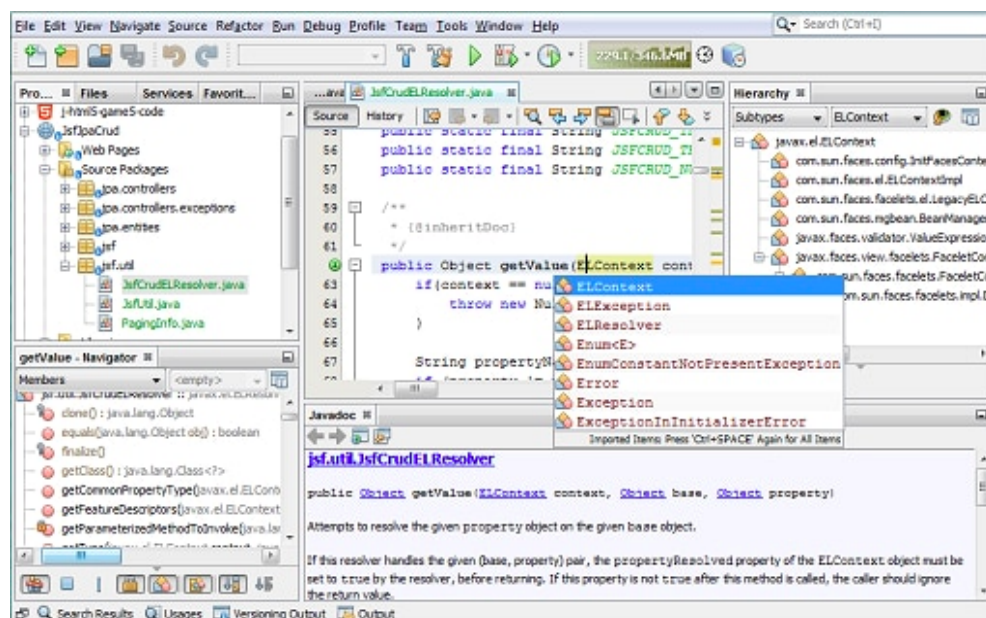
La ventaja que ofrece esta forma de traducción es básicamente la rapidez en la ejecución del programa final, pues una vez obtenido el código ejecutable, la máquina puede proceder directamente a ello.

El inconveniente es que la fase de depuración puede resultar, en algunos casos, bastante más penosa, y la sintaxis de los lenguajes compilados suele ser mucho más estricta y más exigente en cuanto a la declaración de datos.

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 desde un **IDE** o *Integrated Development Environment* (Entorno de Desarrollo Integrado), éste consta de un editor con las funciones necesarias para la compilación y una serie de funciones que facilitan la depuración de los programas.

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



Cuando un programa sin errores sintácticos (*syntax error*), y ya compilado, obtiene una salida que no es la esperada, se dice que posee **errores en su lógica de programación** (*logic error*).

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, de ahí que hoy en día en la mayoría de las aplicaciones se distribuyen parches o *patches* para subsanar errores no encontrados en la fase de depuración.

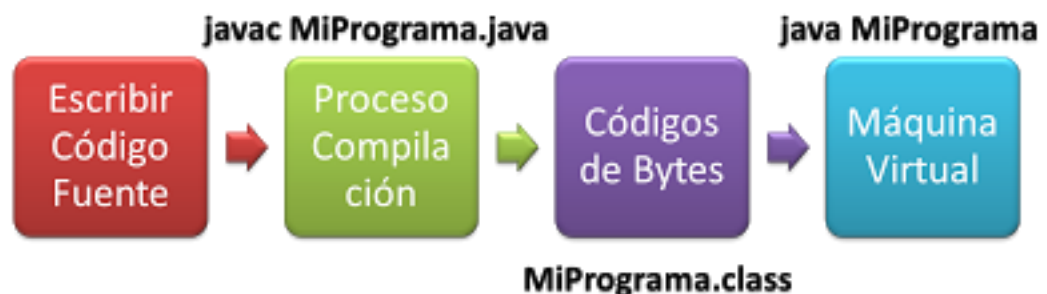
La labor del programador, además de codificar, es la de prevenir, predecir, encontrar y subsanar (si es posible) o al menos controlar (si se dan cuando el producto es entregado), los errores del software que produzca.

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

El caso Java

Java no es un lenguaje ni interpretado ni compilado, sino que es traducido 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.

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 a simplificar por un lado la sintaxis, alejándose del código máquina y acercándose al lenguaje humano, y por otro a la creación de instrucciones potentes que puedan ser traducidas fácil y cómodamente en un código máquina eficiente.

Primera Generación (1GL) – Años 40-50



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

Se trataba de lenguajes absolutamente dependientes de la máquina en particular. El “oficio” de programador no existía, pues eran los propios ingenieros constructores de estos ordenadores los encargados de codificar las órdenes binarias.

Los lenguajes ensambladores supusieron un avance en cuanto a que, en lugar de secuencias de 0 's y 1' s, se describen tanto las operaciones como los datos a través de símbolos.

Aparecen los primeros 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 sintaxis similar a las palabras en una oración. El proceso de traducción es 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, 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, de propósito general, se hizo muy popular al quedar asociado a la informática personal. Fue el primer producto creado por Microsoft.



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 eficiencia y reusabilidad del código, dio lugar a la creación de lenguajes que corregía los problemas presentados por los no estructurados.

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

Ésta es también la época en que se define el **modelo relacional de bases de datos**, aún 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, obteniendo información mediante interrogaciones. Ejemplos de estos lenguajes son: **SQL**, generadores de aplicaciones, **herramientas CASE**.

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**, acordes con los entornos gráficos

Quinta generación

La caracterización de los lenguajes de quinta generación es algo más difusa. Se trata de lenguajes 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 la puesta al día de lenguajes pioneros en su momento, como Lisp, Prolog y también podríamos incluir en este grupo a los lenguajes funcionales, como **Haskel**, **Python** y **Ruby**.

No todos los expertos tienen por tanto claro que esta generación exista claramente diferenciada de las anteriores.

<h2>1.4 – TÉCNICAS DE PROGRAMACIÓN</h2>

Tal y como lo 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, nos referiremos en exclusiva a estos en lo sucesivo.

F | H | A



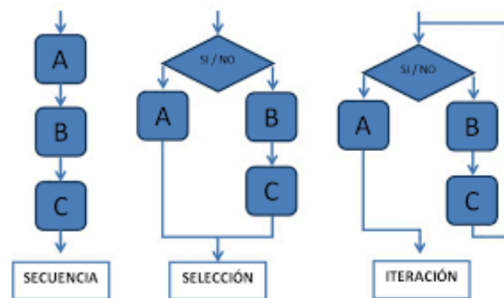
(C) Fachhochschule Aargau für
Technik, Wirtschaft und Gestaltung
Nordwestschweiz

10

1. Un **alfabeto** (*alphabet*): 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.

- **Lenguajes de Programación Estructurados** (*Structured programming language*):

13

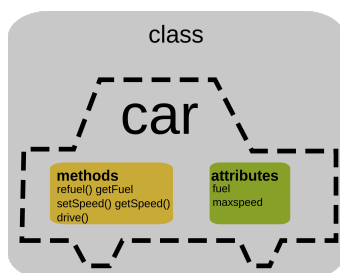


La programación estructurada evolucionó hacia la **programación modular** (*Modular programming*), que divide el programa en porciones de código llamados módulos, con una cohesión interna y funcionalidad concreta, que simplifican la resolución del problema por el conocido método de “divide y vencerás”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

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

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

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



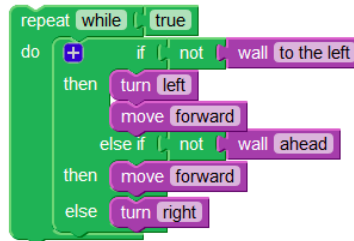
acciones.

La visión que se tiene de los programas en la Orientación a Objetos no es la de un conjunto ordenado de instrucciones, sino de **un conjunto de objetos que representan un modelo del mundo real**, con una serie de características y un comportamiento, que colaboran e interactúan entre ellos para realizar

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

Basados en las técnicas anteriores, permiten programar visualmente, arrastrando y pegando componentes en 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 algo secundario, a una actividad crítica y central en el desarrollo de la informática.

En un principio, la industria focaliza 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, y finalmente, doméstico, con la aparición de los ordenadores personales, y actualmente generalizado a cualquier actividad con los móviles y otros **smart devices** (dispositivos inteligentes). Estamos en la era del Internet de las cosas (*IoT, Internet of Things*) y de la IA.

Comienza 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.

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 en 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 de 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, que cumple con unos estándares de desarrollo aceptados, debidamente documentados y medibles.

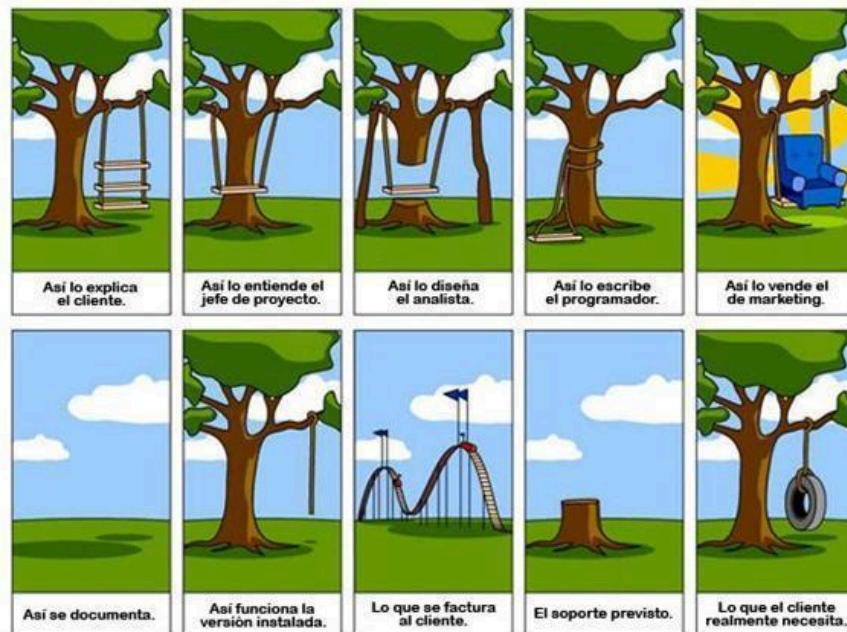
Se trata de esta manera de que el desarrollo de software sea mucho más metodológico y estructurado, disminuyendo de esta manera notablemente los fallos, las imprevisiones, y las correcciones costosas.

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 estos problemas, ya que eliminarlos totalmente es una pretensión ilusoria.

Para expresarlo de un modo gráfico, con las técnicas de la Ingeniería del Software se trata de que no ocurra lo siguiente:

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
- **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:
 - o Diseño arquitectónico: identifica los componentes y su interrelación *grosso modo*
 - o Diseño detallado: describe cada componente en detalle
- **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 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 pruebas, en que en aquellas, las técnicas utilizadas son dinámicas, ya que requieren la ejecución del software.
- **Mantenimiento** (*maintenance*): modificaciones a un producto de software previamente liberado para prevenir fallos (preventivo), corregirlos (correctivo), mejorar su funcionamiento (perfectivo) o adaptarlo a cambios del entorno (adaptativo)
- **Administración 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** (*methods and tools*): 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 y se pueden agrupar en los llamados *workbench*. Los métodos de ingeniería de software establecen una estructura para sistematizar las actividades con el objetivo de aumentar las posibilidades de éxito.

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, en tareas.

Las etapas pueden variar, pero las que se suelen considerar son:

- **Análisis** (de los requerimientos)
- **Diseño** de las estructuras de datos, la arquitectura del software, la interfaz de usuario y los procedimientos. También se elije el lenguaje y/o el gestor de bases de datos a utilizar

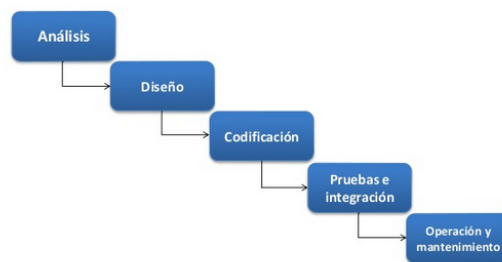
- **Codificación**, que consiste en la programación
- **Pruebas**, en las que se comprueba el correcto funcionamiento del sistema
- **Mantenimiento**, que ocurre después de la entrega del software al cliente. También recibe el nombre de soporte.

Además, cabe resaltar que una de las tareas más importantes en cada una de estas etapas es la **generación de documentación**.

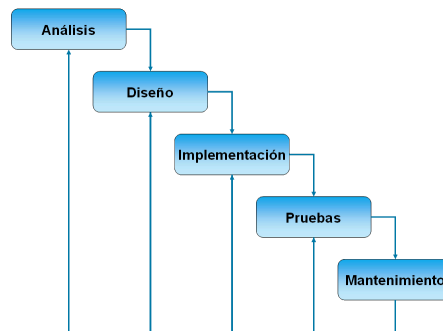
Modelos de ciclo de vida del software

Aunque el consenso en la existencia de estas etapas necesarias es generalizado, existen diversos modelos de ciclo de vida que se pueden tomar como referente en la generación de software. Los veremos tal y como aparecieron históricamente.

1. **Modelo en Cascada o lineal** (*waterfall lifecycle*): Es el modelo de vida clásico del software. Consiste en recorrer en secuencia los pasos anteriormente citados. Es prácticamente imposible que se pueda utilizar a día de hoy, ya que da por hecho que se conocen de antemano todos los requisitos del sistema. Sólo es aplicable a pequeños desarrollos, porque las etapas pasan de una a otra sin retorno posible. (se presupone, por tanto que no habrá errores ni variaciones del software, algo muy temerario).

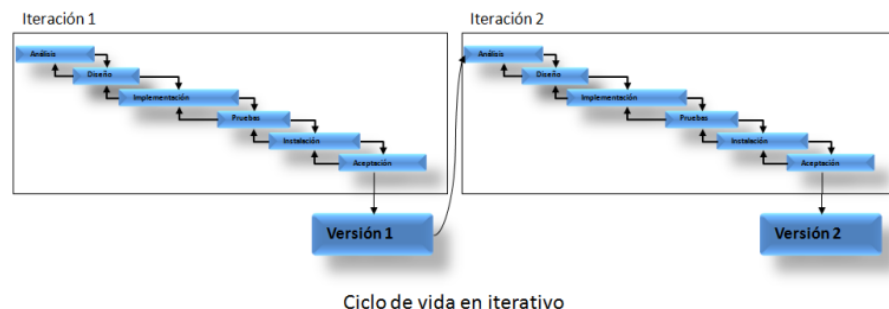


2. **Modelo en Cascada con Realimentación** (*waterfall lifecycle with feedback*): Es uno de los modelos más utilizados. Proviene del modelo anterior, pero se introduce una realimentación entre etapas, de forma que podamos volver atrás en cualquier momento para corregir, modificar o depurar algún aspecto. No obstante, si se prevén muchos cambios durante el desarrollo no es el modelo más idóneo. Es el modelo perfecto si el proyecto es rígido (pocos cambios, poco evolutivo) y los requisitos están claros.



3. **Modelos Evolutivos:** Son más modernos que los anteriores. Tienen en cuenta la naturaleza cambiante y evolutiva del software. Distinguimos dos variantes:

- 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 por partes, para después integrarlas a medida que se completan. A cada vuelta se agregan nuevas funcionalidades al sistema
 - Es **iterativo**, por que a cada iteración se revisa, evalúa y mejora el producto, lo que incide en su calidad final

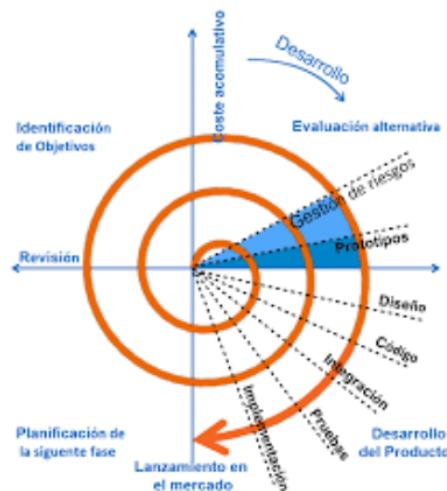


Por tanto, de la unión del ciclo de vida iterativo y el incremental al final de cada iteración se consigue una versión más estable del software, de más calidad, y añadiendo además nuevas funcionalidades respecto a versiones anteriores.

- **Modelo en Espiral** (*spiral lifecycle model*): En él, el software se va construyendo repetidamente **en forma de versiones**, comenzando por la de riesgo más asumible, y se hace una iteración en espiral, que consiste en nuevas versiones cada vez mejores, debido a que incrementan la funcionalidad. 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.

- **Ventaja:** como el software evoluciona a medida que progresa el proceso, el desarrollador y el cliente comprenden y reaccionan mejor ante los riesgos.
- **Inconveniente:** la flexibilidad conlleva incertidumbre de los plazos temporales. No se sabe cuando el proceso acabará



4. **Modelo Ágil:** aparece como evolución natural de los anteriores, 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 encorsetar. 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 que la de seguimiento de planes pre-establecidos.



Manifiesto for Agile Software Development

We are uncovering better ways of developing software by doing it and helping others do it.
Through this work we have come to value:

Individuals and interactions over processes and tools

Working software over comprehensive documentation

Customer collaboration over contract negotiation

Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

Como principales **ventajas** de esta filosofía, que es la que más se utiliza en el presente y futuro inmediato, se pueden indicar la extrema flexibilidad en el proceso, el tiempo récord en que se puede tener una versión temprana del producto funcionando, la interacción y realimentación continua con el cliente que inciden en una mejora en la calidad del producto y en la satisfacción de éste. 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 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.



Está basado directamente en el 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** es un tanto difuso, pues abarca un poco 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, **utilizando una serie de herramientas** determinadas, articuladas en una **sucesión de pasos**, y que genera una

documentación estandarizada, encaminada a organizar el desarrollo de software mediante una serie de “buenas prácticas”.

Por tanto, una metodología particular, refleja un estilo de hacer las cosas, organiza de forma clara el trabajo a realizar e indica detalles del proceso, y de la documentación que debe generarse.

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

Pues bien, una metodología en particular nos especifica **cómo 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.

Y así en cada fase 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 y sistemáticas**, estas metodologías tienen una serie de pasos y generan un tipo de documentación que adolece de una marcada rigidez.

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

Nosotros, a lo largo del desarrollo de esta asignatura, estudiaremos en particular los caso de **Scrum/Kanban**, **RUP** y de **Métrica v3**, que pese a pertenecer al primer y más antiguo grupo, sigue vigente en España (en una versión actualizada, la v3) por ser la que utiliza la Administración Pública.

Por último, no debemos confundir lo que es una metodología de desarrollo otras cosas, por ejemplo, un paradigma de programación (estructurada, orientada a objetos, visual), o un tipo de ciclo de vida (cascada, iterativo, espiral,...), ni con otro tipo de creaciones como el

lenguaje UML, aunque 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).

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.

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 RUP ó Métrica utilizan diagramas UML como herramientas para expresar estructuras e ideas.

1.8 – LICENCIAS DE SOFTWARE. SOFTWARE LIBRE Y PROPIETARIO

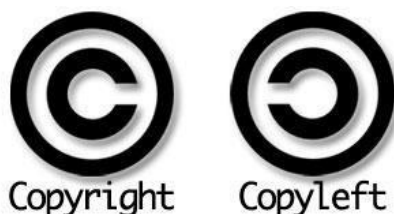
Una **licencia de software** (*software license*) es un contrato que se establece entre el desarrollador de un software sometido a propiedad intelectual y a derechos de autor, y el usuario, en el que se definen con precisión los derechos y deberes de ambas partes. Es el desarrollador, o aquel a quien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



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, 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 lo

puede utilizar, e incluso cabe desarrollar una oferta propietaria sobre la base de un código que se encuentra en el dominio público.

¿Qué es 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 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 el 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 – Compartirlgual (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 – Compartirlgual (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.

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 emula uno o varios sistemas operativos completos y permiten que se ejecuten en una sola máquina física mediante un software llamado **hypervisor**.

El hypervisor dialoga con el sistema operativo real (no virtual), que es 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 sobre un ordenador Apple, por ejemplo, pudiendo de esta manera correr aplicaciones nativas 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 así. Están también las denominadas **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 precompilado sobre cualquier sistema operativo.



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 los API's (bibliotecas de clases estándar, necesarias para que la aplicación, escrita en algún lenguaje de programación pueda ser ejecutada). Estos dos componentes se suelen distribuir conjuntamente, porque necesitan ser compatibles entre sí.

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



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 todo 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 sí, 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. El contenedor actúa de esta forma, como una funda para el software que lo habilita para funcionar dentro de un nuevo entorno ajeno.

Dependiendo del contexto y de la situación particular, puede ser más recomendable utilizar una solución u otra.



Docker es el formato de contenedor más usado en la actualidad, exclusivo para kernel Linux. Solo necesitamos tener instalado el software docker en la máquina para poder ejecutar cualquier aplicación empaquetada en un contenedor de este tipo.

Kubernetes es un orquestador de contenedores para aplicaciones 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, chequeo, etc. Permite escalar el número de contenedores en ejecución, ampliando o reduciendo su número.

Frameworks

Se suele utilizar el término inglés, pero 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, una determinada organización predefinida (o arquitectura) que se considera óptima para el desarrollo de ese lenguaje en particular, y herramientas para el desarrollo rápido de código.

Se pretende de esta manera ayudar a automatizar y ayudar a desarrollar y unir los diferentes componentes de un proyecto. Se trata de no partir de 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** para PHP, y **.NET** de Microsoft. **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 poética de la propia red de redes, que se suele representar como tal en los diagramas de topologías de red.

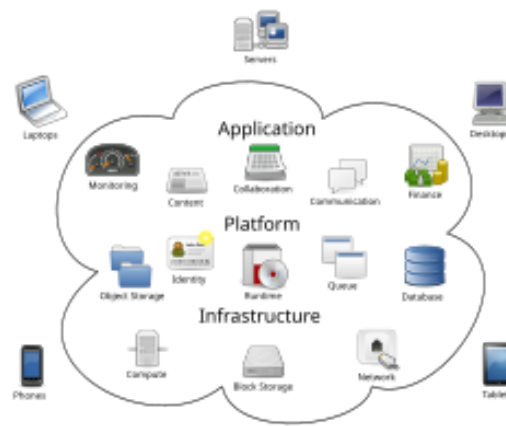
Tanto los usuarios particulares como las empresas pueden, a través de estos servicios, y sin disponer de infraestructuras locales especialmente costosas, gestionar bases de datos o hacer uso de aplicaciones sin necesidad de instalarlas en el computador. Solo es necesario disponer de una conexión a Internet, 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 máquinas 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 máquinas. El hardware resulta completamente transparente, todo lo manejamos de forma virtual.



La principal ventaja de utilizar computación en la nube es también su principal inconveniente: la independencia de una instalación local particular, la hace 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 seguridad.

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), IDEs (Eclipse Che), herramientas de gestión de proyectos (como Trello, Jira, Slack), repositorios (como GitHub, Docker Hub), etc...

Devops (Development-Operation)

Finalmente, tenemos que hablar de un concepto que hasta hace unos años 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
- **Compartición de información** entre desarrollo y operación

La irrupción de la IA en el desarrollo de software

La inteligencia artificial (IA) está transformando el desarrollo de software al automatizar tareas, mejorar la calidad del código y optimizar procesos.

Las herramientas de IA pueden generar código, detectar errores, y acelerar la depuración, permitiendo a los desarrolladores enfocarse en tareas más complejas y creativas.

Algunos de los beneficios del uso de la IA en el desarrollo de software pueden ser:

- **Automatización de tareas repetitivas:** La IA puede generar código, identificar errores y optimizar el rendimiento, liberando a los desarrolladores de tareas tediosas.
- **Mejora de la calidad del software:** Herramientas de IA pueden analizar código, detectar vulnerabilidades y sugerir mejoras, lo que resulta en software más robusto y confiable.
- **Mayor eficiencia y productividad:** La IA acelera el desarrollo al automatizar tareas y proporcionar herramientas de análisis, permitiendo a los equipos entregar proyectos más rápido y con menos recursos.
- **Toma de decisiones más rápida:** La IA puede analizar datos y ofrecer información valiosa para la toma de decisiones en la gestión de proyectos y el diseño de software.
- **Experiencia de usuario mejorada:** La IA puede personalizar la experiencia del usuario al analizar datos y adaptar la interfaz de la aplicación.
- **Innovación y creatividad:** La IA puede ser una herramienta poderosa para la innovación, generando nuevas ideas y soluciones a problemas complejos.

Como ejemplos del uso de la IA en el desarrollo de software, podemos destacar:

- **Asistentes de codificación:** herramientas como GitHub Copilot y Tabnine pueden generar código, completar líneas y sugerir mejoras basadas en el contexto del proyecto.
- **Pruebas automatizadas:** La IA puede generar casos de prueba, ejecutar pruebas y analizar resultados, lo que acelera el proceso de pruebas y mejora la calidad del software.
- **Análisis de código:** herramientas de IA pueden analizar código en busca de errores, vulnerabilidades y problemas de

rendimiento, ayudando a los desarrolladores a corregir problemas antes de que afecten al usuario final.

- **Optimización de código:** la IA puede analizar el código y sugerir mejoras en rendimiento, legibilidad y mantenibilidad.
- **Generación de documentación:** La IA puede ayudar a generar documentación técnica del código, lo que facilita la comprensión y el mantenimiento del software.