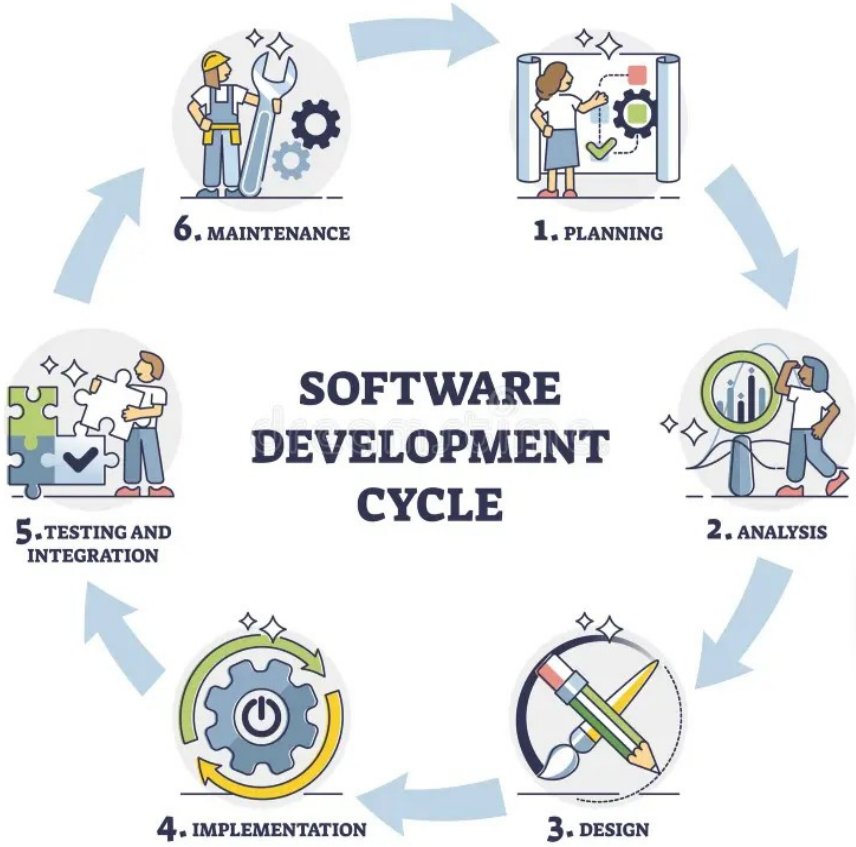
UT1 INTRODUCCIÓN AL DESARROLLO DEL SOFTWARE
Curso 2025/2026 ENTORNOS DE DESARROLLO 1º DAW



UT1 INTRODUCCIÓN AL DESARROLLO DEL SOFTWARE
ÍNDICE

ÍNDICE

UT1 INTRODUCCIÓN A ARQUITECTURAS WEB
RESULTADOS DE APRENDIZAJE EVALUADOS

RESULTADO DE APRENDIZAJE
RA 1: Reconoce los elementos y herramientas que intervienen en el desarrollo de un programa informático, analizando sus características y las fases en las que actúan hasta llegar a su puesta en funcionamiento.
CRITERIOS DE EVALUACIÓN
a) Se ha reconocido la relación de los programas con los componentes del sistema informático: memoria, procesador, periféricos, entre otros.
b) Se han identificado las fases de desarrollo de una aplicación informática.
c) Se han diferenciado los conceptos de código fuente, objeto y ejecutable.
d) Se han reconocido las características de la generación de código intermedio para su ejecución en máquinas virtuales.
e) Se han clasificado los lenguajes de programación.

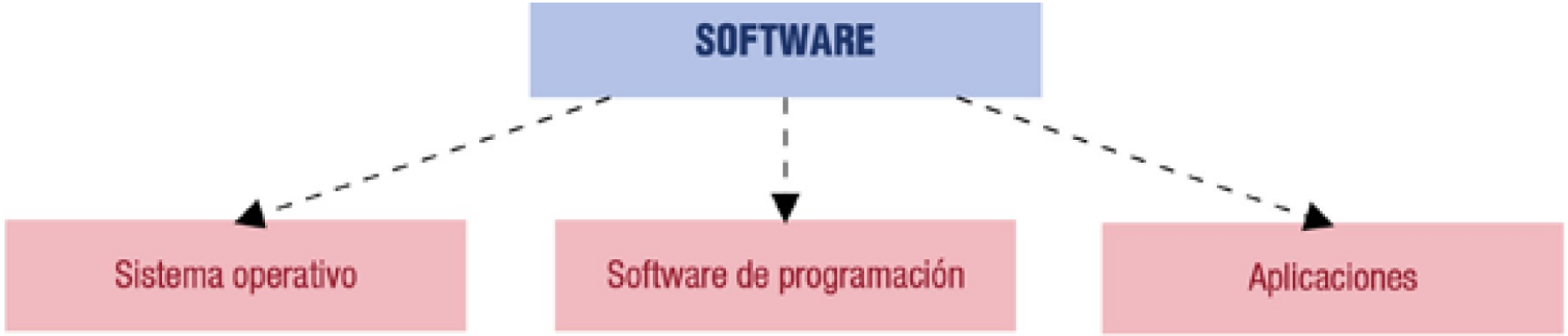
En este tema se califica el Resultado de Aprendizaje 1 (RA1) que se muestra en la tabla

1. Software y programa. Tipos de software.

Es de sobra conocido que el ordenador se compone de dos partes bien diferenciadas: **hardware** y **software**.

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

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



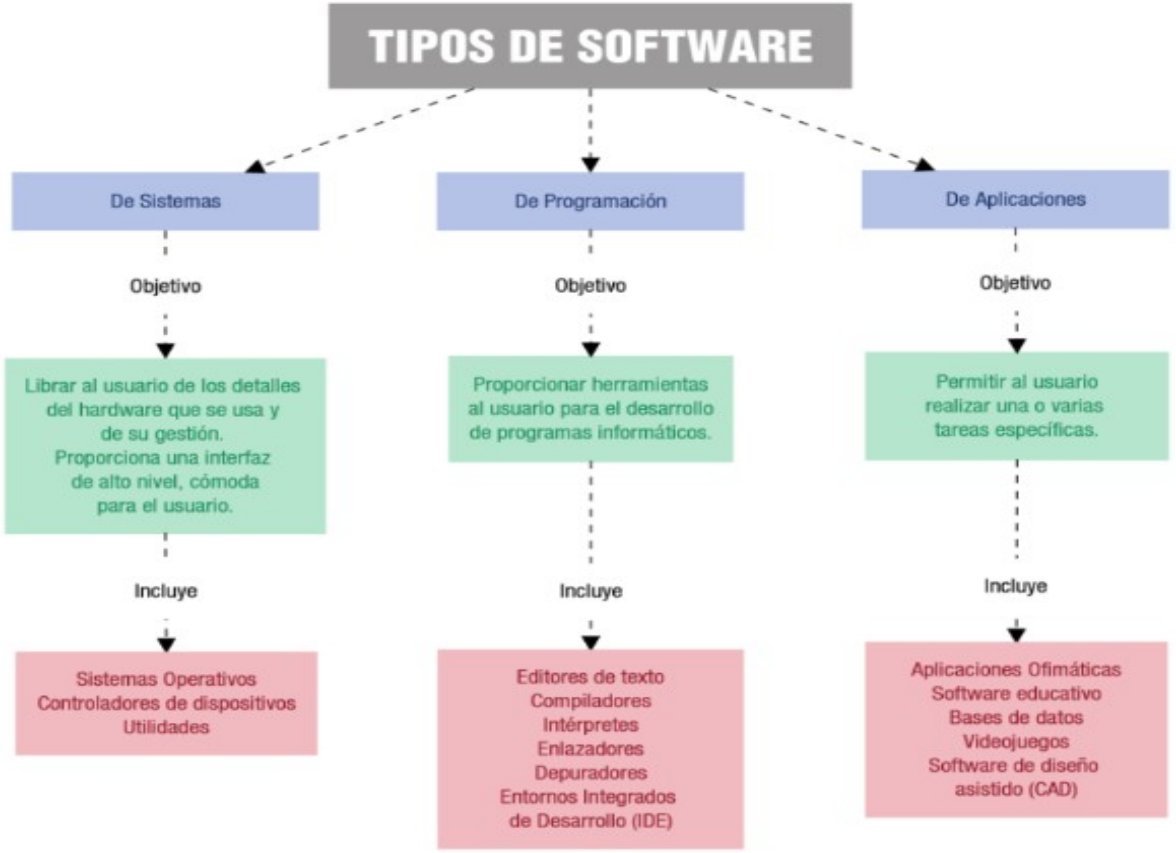
1. Software y programa. Tipos de software.

El *sistema operativo* es el software base que ha de estar instalado y configurado en nuestro ordenador para que las aplicaciones puedan ejecutarse y funcionar. Son ejemplos de sistemas operativos: Windows, Linux, Mac OS X ...

El *software de programación* es el conjunto de herramientas que nos permiten desarrollar programas informáticos, y las *aplicaciones informáticas* son un conjunto de programas que tienen una finalidad más o menos concreta. Son ejemplos de aplicaciones: un procesador de textos, una hoja de cálculo, el software para reproducir música, un videojuego, etc.

Un programa es un conjunto de instrucciones escritas en un lenguaje de programación.

1. Software y programa. Tipos de software.



En este tema, nuestro interés se centra en las **aplicaciones informáticas**: cómo se desarrollan y cuáles son las fases por las que necesariamente han de pasar.

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

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

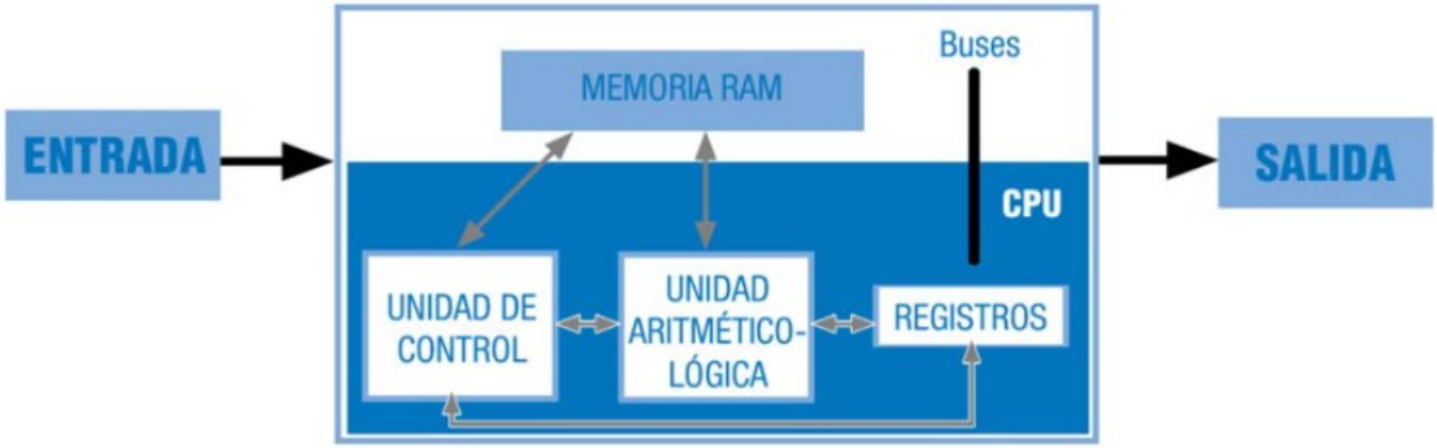
2. Relación hardware-software.

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

Existe una relación indisoluble entre éste y el software, ya que necesitan estar instalados y configurados correctamente para que el equipo funcione.

El software se ejecutará sobre los dispositivos físicos.

La primera arquitectura hardware con programa almacenado se estableció en 1946 por John Von Neumann:



2. Relación hardware-software.

Esta relación software-hardware la podemos poner de manifiesto desde dos puntos de vista:

a) Desde el punto de vista del sistema operativo

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

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

2. Relación hardware-software.

b) Desde el punto de vista de las aplicaciones

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

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

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

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

2. Relación hardware-software.

Autoevaluación

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

A)Hay que escribir las instrucciones en código binario para que las entienda el hardware.

B)Sólo es necesario escribir el programa en algún lenguaje de programación y se ejecuta directamente.

C)Hay que escribir el programa en algún Lenguaje de Programación y contar con herramientas software que lo traduzcan a código binario.

D)Los programas informáticos no se pueden escribir: forman parte de los sistemas operativos.

2. Relación hardware-software.

Autoevaluación

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

A)Hay que escribir las instrucciones en código binario para que las entienda el hardware.

B)Sólo es necesario escribir el programa en algún lenguaje de programación y se ejecuta directamente.

C)Hay que escribir el programa en algún Lenguaje de Programación y contar con herramientas software que lo traduzcan a código binario.

D)Los programas informáticos no se pueden escribir: forman parte de los sistemas operativos.

C) Hay que escribir el programa en algún Lenguaje de Programación y contar con herramientas software que lo traduzcan a código binario.

3. Desarrollo de software.

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

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

Como veremos con más detenimiento a lo largo de la mitad de la unidad, el desarrollo es un proceso que conlleva una serie de pasos. Genéricamente, estos pasos son los siguientes;

3. Desarrollo de software.

Etapas en el desarrollo de software:

Como vamos a ver en el siguiente punto, según el orden y la forma en que se lleven a cabo las etapas hablaremos de diferentes ciclos de vida del software.

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

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

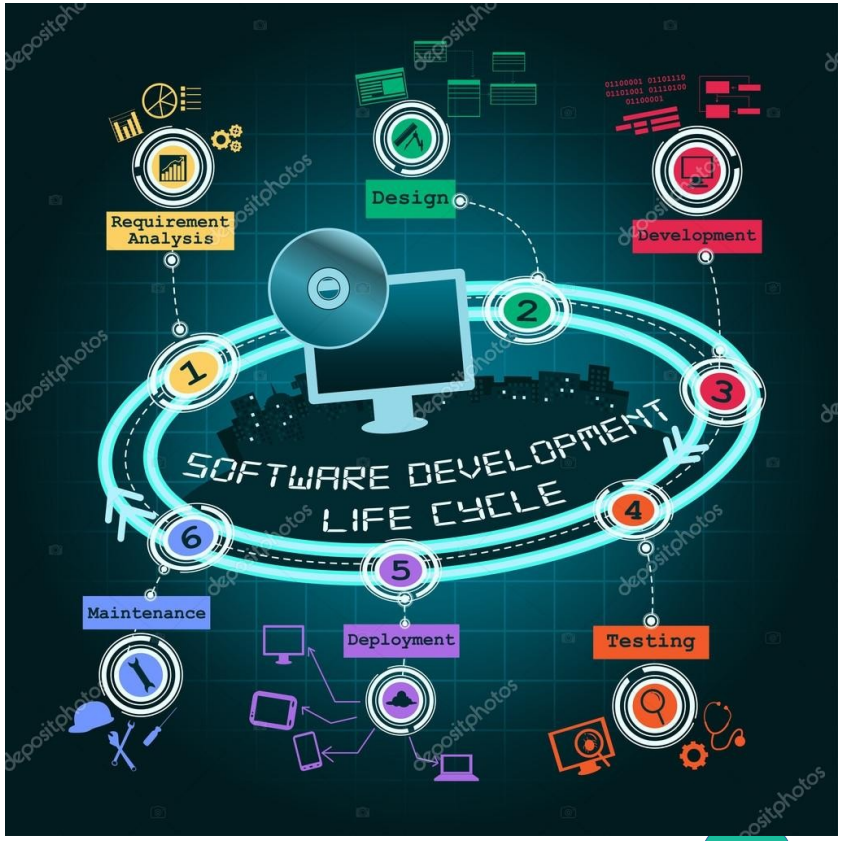
3.1 CICLOS DE VIDA DEL SOFTWARE

El ciclo de vida del desarrollo de software (SDLC del ingles Software Development Life Cycle) es el proceso paso a paso que siguen los equipos para crear aplicaciones de software, desde la primera chispa de una idea hasta el producto final en manos de los usuarios.

Cada etapa vendrá explicada con más detalle en el punto 5..

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

Siempre se debe aplicar un modelo de ciclo de vida al desarrollo de cualquier proyecto software.



3.1.1 MODELOS DE CICLOS DE VIDA DEL SOFTWARE CLÁSICOS

Existen numerosos modelos de ciclo de vida del desarrollo de software, cada uno con sus propias fortalezas y debilidades. La selección del modelo depende de factores como el tamaño y complejidad del proyecto, los requisitos del cliente, los recursos disponibles y la tolerancia al riesgo. Una mala elección puede llevar a retrasos, sobrecostos y un producto final insatisfactorio. A continuación se comparan tres modelos representativos.

Modelo	Ventajas	Desventajas	Aplicación típica
Cascada	Simple, fácil de entender y gestionar; requerimientos bien definidos al inicio.	Poco flexible ante cambios; detección tardía de errores; riesgo de entrega tardía o producto no satisfactorio.	Proyectos pequeños con requisitos bien definidos y estables, como sistemas embebidos con funcionalidades limitadas.
Espiral	Alta capacidad de adaptación a cambios; gestión de riesgos proactiva; apropiado para proyectos complejos.	Puede ser costoso y complejo de gestionar; requiere una planificación cuidadosa y expertos en gestión de riesgos.	Desarrollo de sistemas críticos donde la seguridad y la fiabilidad son primordiales, como software aeroespacial o médico.
Incremental	Entrega temprana de funcionalidad; mayor flexibilidad ante cambios; reduce el riesgo global del proyecto.	Requiere una planificación cuidadosa de los incrementos; la integración de los incrementos puede ser compleja.	Desarrollo de sistemas grandes y complejos donde la entrega temprana de funcionalidades clave es importante, como un sistema de gestión de contenido empresarial (CMS).

Para más información puedes ir a este **enlace**

3.1.1 MODELOS DE CICLOS DE VIDA DEL SOFTWARE CLÁSICOS

1. Modelo en Cascada

Es el modelo de vida clásico del software. Existen tres variantes:

1.- En el las etapas para el desarrollo del software tienen un orden y para empezar una etapa es necesario finalizar la etapa anterior, después de cada etapa se realiza una revisión para comprobar si se puede pasar a la siguiente.

3.1.1 MODELOS DE CICLOS DE VIDA DEL SOFTWARE CLÁSICOS

1. Modelo en Cascada

2.- En este modelo se permite hacer iteraciones, por ejemplo durante la etapa de mantenimiento del producto el cliente requiere una mejora, esto implica que hay qe modificar algo en el diseño, lo cual significa que habrá que hacer cambios en la codificación y se tendrán que realizar de nuevo las pruebas, es decir, si se tiene que volver a una etapa de las etapas anteriores hay que recorrer de nuevo el resto de las etapas.

3.1.1 MODELOS DE CICLOS DE VIDA DEL SOFTWARE CLÁSICOS

1. Modelo en Cascada

3.- Otra variante es el *Modelo en Cascada con Realimentación* siendo esta una de las más utilizadas. Por ejemplo supongamos que la etapa de Análisis (captura de requisitos) ha finalizado y se puede pasar a la de Diseño. Durante el desarrollo de esta etapa se detectan fallos (los requisitos han cambiado, han evolucionado, ambigüedades en la definición de los mismos, etc), entonces será necesario retomar la etapa anterior, realizar los ajustes pertinentes y continuar de nuevo con el Diseño. A esto se le conoce como realimentación, pudiendo volver de una etapa a la anterior o incluso de varias etapas a la anterior.

3.1.1 MODELOS DE CICLOS DE VIDA DEL SOFTWARE CLÁSICOS

1. Modelo en Cascada

Ventajas:

- Fácil de comprender, planificar y seguir.
- La calidad del producto resultante es alta.
- Permite trabajar con personal poco cualificado.

Inconvenientes:

- La necesidad de tener todos los requisitos definidos desde el principio (algo que no siempre ocurre ya que pueden surgir necesidades imprevistas).
- Es difícil volver atrás si se cometen errores en una etapa.
- El producto no está disponible para su uso hasta que no está completamente terminado

Se recomienda cuando:

- El proyecto es similar a alguno que ya se haya realizado con éxito anteriormente.
- Los requisitos son estables y están bien comprendidos.
- Los clientes no necesitan versiones intermedias

3.1.1 MODELOS DE CICLOS DE VIDA DEL SOFTWARE CLÁSICOS

2. Modelos Evolutivos

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

El modelo en cascada asume que se va a entregar un producto completo, en cambio los modelos evolutivos permiten desarrollar versiones cada vez mas completas hasta llegar al producto final deseado. En estos modelos se asume que las necesidades del usuario no estñan completas y se requiere una vuelta a planificar y diseñar después de cada implantación de los entregables.

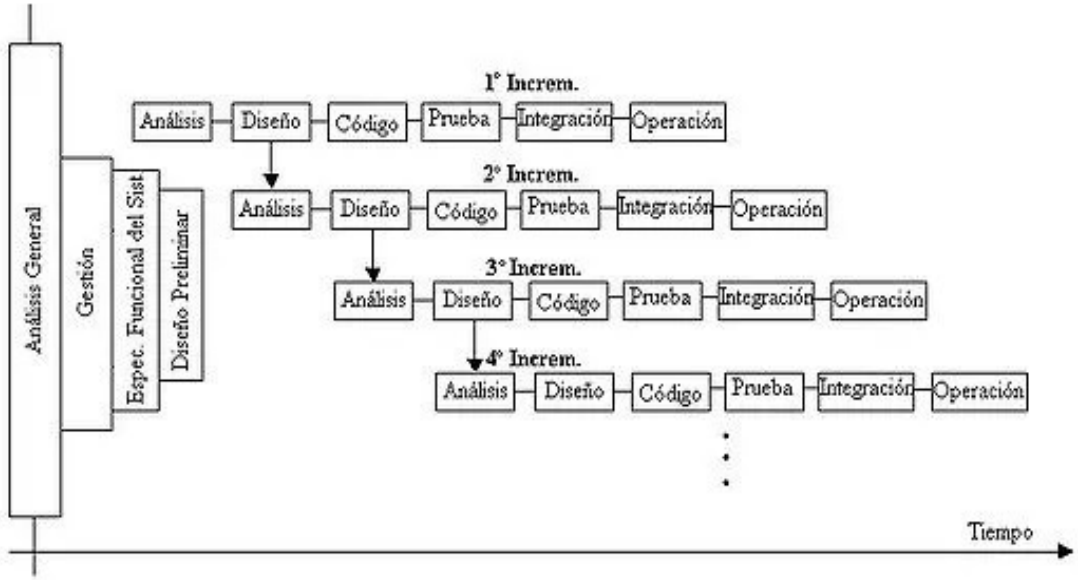
Los Modelos Evolutivos más conocidos son: el Iterativo Incremental y el Espiral

3.1.1 MODELOS DE CICLOS DE VIDA DEL SOFTWARE CLÁSICOS

2. Modelos Evolutivos.

2.1 Modelo Iterativo Incremental

Está basado en varios ciclos en cascada realimentados aplicados repetidamente. El modelo incremental entrega el software en partes pequeñas, pero utilizables, llamadas “incrementos”. En general, cada incremento se construye sobre aquél que ya ha sido entregado. Como se puede ver en la figura se muestra un diagrama del modelo bajo un esquema temporal, se observa de forma iterativa el modelo en cascada para la obtención de un nuevo incremento mientras progresa el tiempo en el calendario.



3.1.1 MODELOS DE CICLOS DE VIDA DEL SOFTWARE CLÁSICOS

2. Modelos Evolutivos.

2.1 Modelo Iterativo Incremental

Ventajas:

- No se necesita conocer todos los requisitos al comienzo.
- Permite la entrega temprana al cliente de partes operaticas del software.
- La entregas facilitan la realimentación de los próximos entregables.

Inconvenientes:

- Es difícil estimar el esfuerzo y el coste final necesario.
- Se tiene el riesgo de no acabar nunca.
- No recomendable para desarrollo de sistemas de tiempo real, de alto nivel de seguridad, de procesamiento distribuido, y/o de alto índice de riesgos.

Se recomienda cuando:

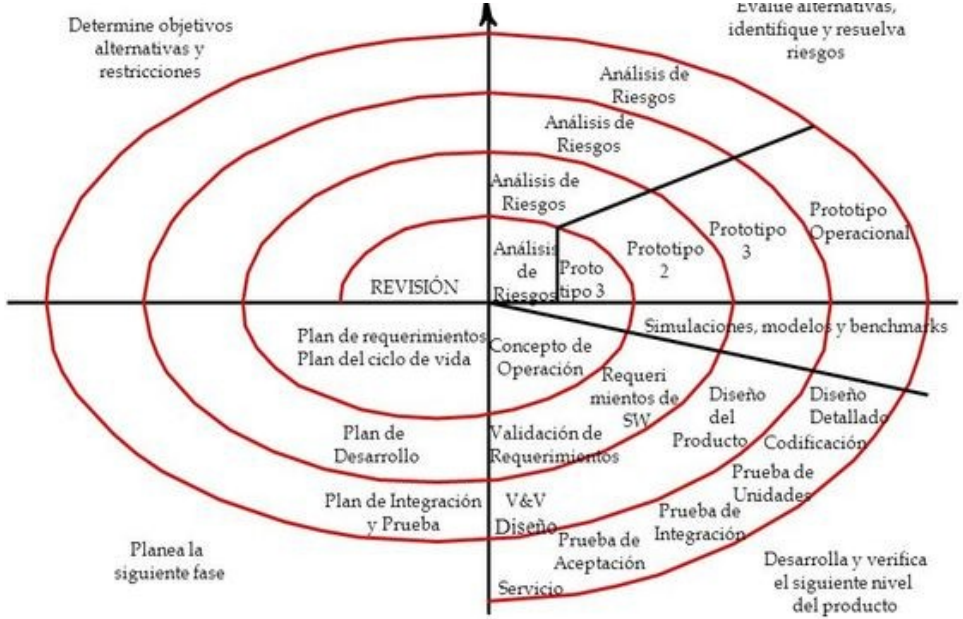
- Los requisitos o el diseño no están completamente definidos y es posible que haya grandes cambios
- Se están provando o introduciendo nuevas tecnologías.

3.1.1 MODELOS DE CICLOS DE VIDA DEL SOFTWARE CLÁSICOS

2. Modelos Evolutivos.

Se trata de una propuesta que ***combina*** las propiedades de ***los modelos en cascada con el modelos iterativos incrementales*** de construcción de prototipos. Se fundamenta en un proceso de desarrollo en el cual se hacen entregas del producto -cada una más evolucionada o completa que la anterior- teniendo en cuenta los riesgos que pueden afectar el proceso. Cada ciclo del espiral representa una etapa del ciclo de vida del software.

2.2 Modelo en Espiral



3.1.1 MODELOS DE CICLOS DE VIDA DEL SOFTWARE CLÁSICOS

2. Modelos Evolutivos.

2.2 Modelo en Espiral

Durante los primeros ciclos la versión incremental podría ser maquetas en papel o modelos de pantallas (prototipos de interfaz); en el último ciclo se tendría un prototipo de operacional que implementa algunas funciones del sistema. Para cada ciclo, los desarrolladores siguen estas fases:

1. **Determinar objetivos:** Cada ciclo de la espiral comienza con la identificación de los objetivos, las alternativas para alcanzarlos (diseño A, diseño B, realización, compras, etc.), y las restricciones impuestas para realizar estas alternativas (costos, plazos, interfaz, etc.).
2. **Análisis del riesgo:** A continuación hay que evaluar las alternativas en relación con los objetivos y las limitaciones. Con frecuencia en este proceso se identifican los riesgos y (si es posible) la manera de resolverlos. Un riesgo puede ser cualquier cosa: requerimientos no comprendidos, mal diseño, errores en la implementación, etc. Utiliza la construcción de prototipos como mecanismo para la reducción de riesgos.
3. **Desarrollar y probar:** Desarrollar la solución al problema en este ciclo, y verificar que es aceptable.
4. **Planificación:** Revisar y evaluar todo lo que se ha hecho, y con ello decidir si se continúa, entonces hay qe planificar las fases del ciclo siguiente

3.1.1 MODELOS DE CICLOS DE VIDA DEL SOFTWARE CLÁSICOS

2. Modelos Evolutivos. 2.2 Modelo en Espiral

Ventajas:

- No se requiere una definición completa de los requisitos para empezar a funcionar.
- Análisis de riesgo en todas las etapas.
- Reduce riesgos del proyecto.
- Incorpora objetivos de calidad.

Inconvenientes:

- Es difícil de evaluar los riesgos.
- El coste del proyecto aumenta a medida que la espiral pasa por sucesivas iteraciones.
- El éxito del proyecto depende en gran medidad de la fase de análisis de riesgos.

Se recomienda cuando:

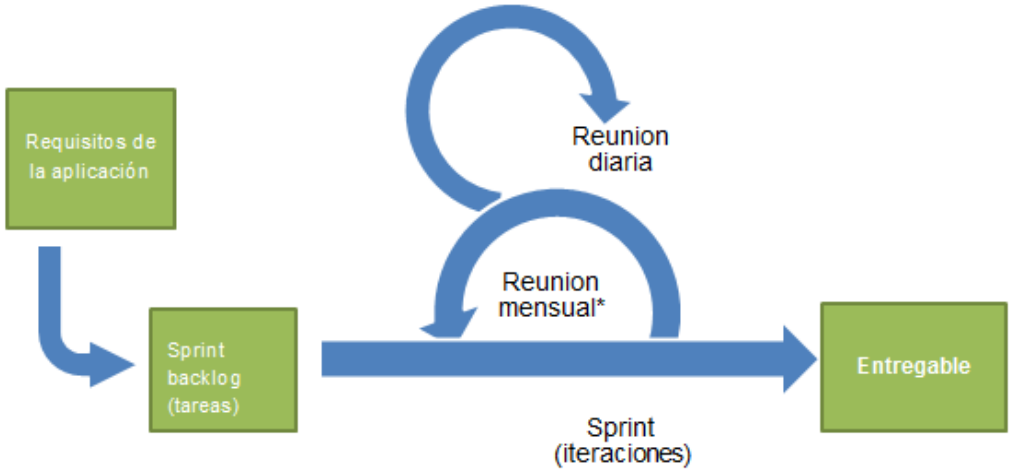
- Proyectos de gran tamaño y que necesita constantes cambios.
- Proyectos donde sea importante el factor riesgo.

3.1.2 MODELOS DE CICLOS DE VIDA DEL SOFTWARE MODERNOS

2. Scrum (dentro de Agile)

Marco de trabajo iterativo en sprints (ciclos de 2-4 semanas).
Roles definidos: Product Owner, Scrum Master, Equipo de desarrollo.
Se hacen reuniones diarias (Daily Stand-up).
Ejemplos reales:

- Usado en startups y grandes empresas como Google, Microsoft, IBM.
- Muy común en desarrollo web y apps.



3.2 HERRAMIENTAS DE APOYO AL DESARROLLO DEL SOFTWARE

En el desarrollo de software, la eficiencia, la colaboración y la calidad del producto final son más importantes que nunca. Por estas razones se hace cada vez más importante usar las herramientas adecuadas para llegar a buen puerto.

Las **herramientas CASE** (Computer Aided Software Engineering – Ingeniería de Software Asistida por Computadora) *son aplicaciones que ayudan a los desarrolladores a planificar, diseñar, documentar y mantener sistemas de software de manera más eficiente.*, representa una solución poderosa que *permite automatizar y optimizar las distintas etapas del ciclo de vida del software*.

3.2 HERRAMIENTAS DE APOYO AL DESARROLLO DEL SOFTWARE

¿Qué son las herramientas CASE?

Una herramienta CASE integra diversos módulos como editores de modelos, generadores de código, repositorios, y controladores de versiones, todos con el fin de mejorar el desarrollo de software.

Componentes de CASE

1) Modeladores visuales
2) Repositorio de datos
3) Generadores de código
4) Herramientas de prueba
5) Control de versiones

Integración de herramientas

La integración entre herramientas CASE permite que todo el equipo trabaje de manera sincronizada. Se puede lograr de forma vertical (entre fases) u horizontal (entre herramientas de la misma fase), usando un repositorio común.

3.2 HERRAMIENTAS DE APOYO AL DESARROLLO DEL SOFTWARE

Beneficios de las herramientas CASE

1)Aumentan la productividad del equipo.
2)Mejoran la calidad del software.
3)Automatizan tareas repetitivas.
4)Facilitan la documentación y el mantenimiento.

3.2 HERRAMIENTAS DE APOYO AL DESARROLLO DEL SOFTWARE

Clasificación de herramientas CASE

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

U-CASE (Upper CASE):

- Se usan en las fases iniciales: ayuda en planificación y análisis de requisitos.
- Permiten modelar procesos, diagramas de flujo de datos, casos de uso.
- Ejemplos: Enterprise Architect: modelado UML, Visual Paradigm: diseño de diagramas, modelado de base de datos, Lucidchart / Draw.io: diagramas de procesos y sistemas.

M-CASE (Middle CASE):

- Se emplean en las fases de análisis y diseño.
- Ayudan a crear diagramas UML (clases, secuencia, etc.), modelos de datos, diseño de arquitectura.
- Ejemplos: Visual Paradigm, PowerDesigner, ERwin Data Modeler.

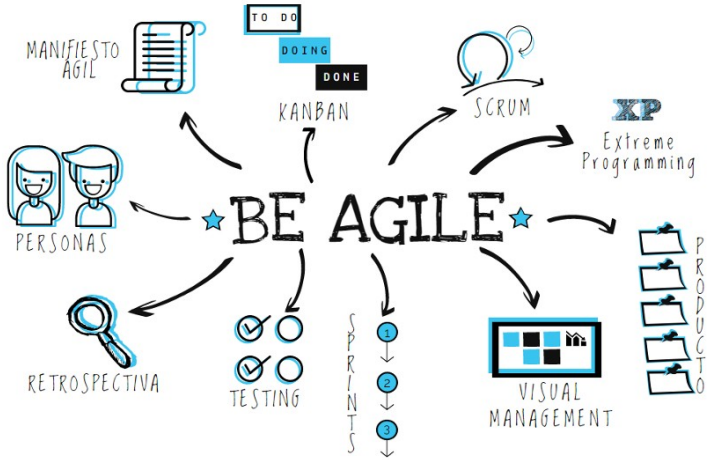
L-CASE (Lower CASE):

- Apoyan en las fases finales: implementación, pruebas, depuración y documentación.
- Generan código, gestionan bases de datos, ejecutan pruebas automáticas, crean reportes.
- Ejemplos: Eclipse, Visual Studio, Selenium, MySQL Workbench.

3.1.2 MODELOS DE CICLOS DE VIDA DEL SOFTWARE MODERNOS

1. Modelo Ágil (Agile) Para más información vete a este enlace

- Filosofía basada en el Manifiesto Ágil (2001).
- Se centra en:
  - Entregas rápidas e incrementales.
  - Colaboración estrecha con el cliente.
  - Adaptación a cambios en requisitos.
- Ejemplos reales:
  - Desarrollo de apps móviles (Uber, Airbnb).
  - Empresas como Spotify y Netflix aplican metodologías ágiles.



4.1 CONCEPTO Y CLASIFICACIÓN

Podemos definir un Lenguaje de Programación como un idioma creado de forma artificial, formado por un conjunto de símbolos y normas que se aplican sobre un alfabeto para obtener un código, que el hardware de la computadora pueda entender y ejecutar.

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

Cada lenguaje tiene su propia sintaxis, semántica y conjunto de reglas que definen cómo se deben escribir las instrucciones. Existen diversos tipos de lenguajes y formas de organizarlos:

- Por generación
- Por ámbito de aplicación
- Por tipado de datos
- Por paradigma de programación

4.1.1 LENGUAJES DE PROGRAMACIÓN POR GENERACIÓN

1.- Lenguajes de primera generación (Lenguaje Máquina): Son lenguajes de bajo nivel que se asemejan al lenguaje de máquina y se utilizan para la programación de hardware.

- Son **instrucciones escritas directamente en código binario (0 y 1)**.
- El procesador las entiende de forma inmediata.
- Ejemplo: 10110000 01100001 (sumar, mover, etc.).
- Ventaja: máxima eficiencia.
- Desventaja: muy difícil de programar y mantener.

2.- Lenguajes de segunda generación (Lenguaje Ensamblador): Son lenguajes de bajo nivel que ofrecen mayor abstracción que los de primera generación.

- Usa **mnemónicos** (abreviaturas simbólicas) en lugar de binario.
- Cada instrucción corresponde casi directamente a una instrucción máquina.
- Ejemplo: MOV AX, 1
- Ventaja: más legible que binario, aún rápido.
- Desventaja: dependiente del procesador y difícil de portar.

4.1.1 LENGUAJES DE PROGRAMACIÓN POR GENERACIÓN Video

3.- Lenguajes de tercera generación (Lenguaje de Alto Nivel): Son lenguajes de alto nivel diseñados para ser más legibles y portables.

- Se parecen más al lenguaje humano, independientes del hardware.
  Ejemplos: C, C++, Java, Python, Pascal, Fortran.
- Ventaja: más fáciles de aprender y usar, portables entre sistemas.
- Desventaja: menor control directo sobre el hardware.

4.- Lenguajes de cuarta generación (Lenguaje de muy Alto Nivel): Son lenguajes de muy alto utilizados para aplicaciones específicas, como bases de datos y procesamiento de datos. Ejemplos incluyen SQL y MATLAB.

- Orientados a resolver problemas específicos con menos líneas de código.
- Se enfocan en bases de datos, consultas y generación de informes.
- Ejemplos: SQL, MATLAB, Oracle Forms, R, ABAP.
- Ventaja: mayor productividad.
- Desventaja: menos flexibles para usos generales.

4.1.2 LENGUAJES DE PROGRAMACIÓN POR ÁMBITO DE APLICACIÓN

Clasificación de lenguajes de programación según su propósito

- **Lenguajes de programación web:** Utilizados principalmente para el desarrollo de aplicaciones web. Ejemplos: JavaScript, PHP, Ruby.
- **Lenguajes de programación científica y matemática:** Diseñados para resolver problemas numéricos, científicos y de cálculo avanzado. Ejemplos: MATLAB, R.
- **Lenguajes de programación de sistemas:** Usados para desarrollar sistemas operativos y software de bajo nivel. Ejemplos: C, C++.
- **Lenguajes de programación de bases de datos:** Orientados a consultar, definir y manipular datos en bases de datos. Ejemplo: SQL.
- **Lenguajes de programación de scripting:** Diseñados para escribir scripts o programas pequeños que automatizan tareas específicas. Ejemplos: Python, Perl, Bash.
- **Lenguajes de programación para sistemas embebidos:** Orientados a dispositivos con recursos limitados (microcontroladores, tiempo real). Ejemplos: C, Ada.

4.1.3 LENGUAJES DE PROGRAMACIÓN POR TIPADO DE DATOS

Cuando hablamos de tipado en lenguajes de programación, nos referimos a cómo manejan los tipos de datos (números, cadenas, booleanos, etc.) y dependiendo del momento en que se comprueban estos tipo tenemos la siguiente clasificación.

- **Tipado estático:** El tipo de cada variable se conoce y comprueba en tiempo de compilación.
  - Ejemplos: C, C++, Java, C#, Rust, Go.
  - Ventaja: Detecta errores antes de ejecutar.
  - Desventaja: Menos flexible.
- **Tipado dinámico:** El tipo se comprueba en tiempo de ejecución.
  - Ejemplos: Python, JavaScript, Ruby, PHP.
  - Ventaja: Más flexible y rápido de escribir.
  - Desventaja: Los errores de tipo pueden aparecer en ejecución.

4.1.5 LENGUAJES DE PROGRAMACIÓN POR PARADIGMA DE PROGRAMACIÓN

Un paradigma de programación es un enfoque o estilo de programación que define cómo se deben escribir, estructurar y organizar los programas de software.

Cada paradigma establece un conjunto de reglas, principios y características que guían la forma en que se desarrollan las aplicaciones.

Principales paradigmas de programación:

- **Programación Imperativa:** El programador indica paso a paso cómo resolver un problema. Se basa en instrucciones secuenciales que modifican el estado del programa. Ejemplos: C, Pascal, Fortran.
- **Programación Orientada a Objetos (POO):** Organiza el software en objetos que tienen atributos (datos) y métodos (funciones). Facilita la reutilización y el mantenimiento del código. Ejemplos: Java, C++, Python, C#.

4.1.5 LENGUAJES DE PROGRAMACIÓN POR PARADIGMA DE PROGRAMACIÓN

- **Programación Funcional:** Se centra en el uso de funciones matemáticas puras (sin efectos secundarios). Favorece la inmutabilidad y la recursión. Ejemplos: Haskell, Lisp, Scala, Elixir.
- **Programación Lógica:** Basada en el uso de reglas lógicas y hechos para deducir conclusiones. Útil en inteligencia artificial y sistemas expertos. Ejemplo: Prolog.
- **Programación Basada en Eventos:** El flujo del programa depende de eventos externos (clics, teclas, mensajes). Muy usada en interfaces gráficas y desarrollo web. Ejemplos: JavaScript (event listeners), Visual Basic.
- **Programación Estructurada:** Variante de la imperativa que promueve dividir el programa en bloques y funciones, evitando el uso excesivo de goto. Ejemplos: C, Pascal.
- **Programación Concurrente y Paralela:** Diseñada para ejecutar múltiples procesos o hilos al mismo tiempo. Es clave en aplicaciones modernas y sistemas distribuidos. Ejemplos: Java (multithreading), Erlang, Go, Rust.

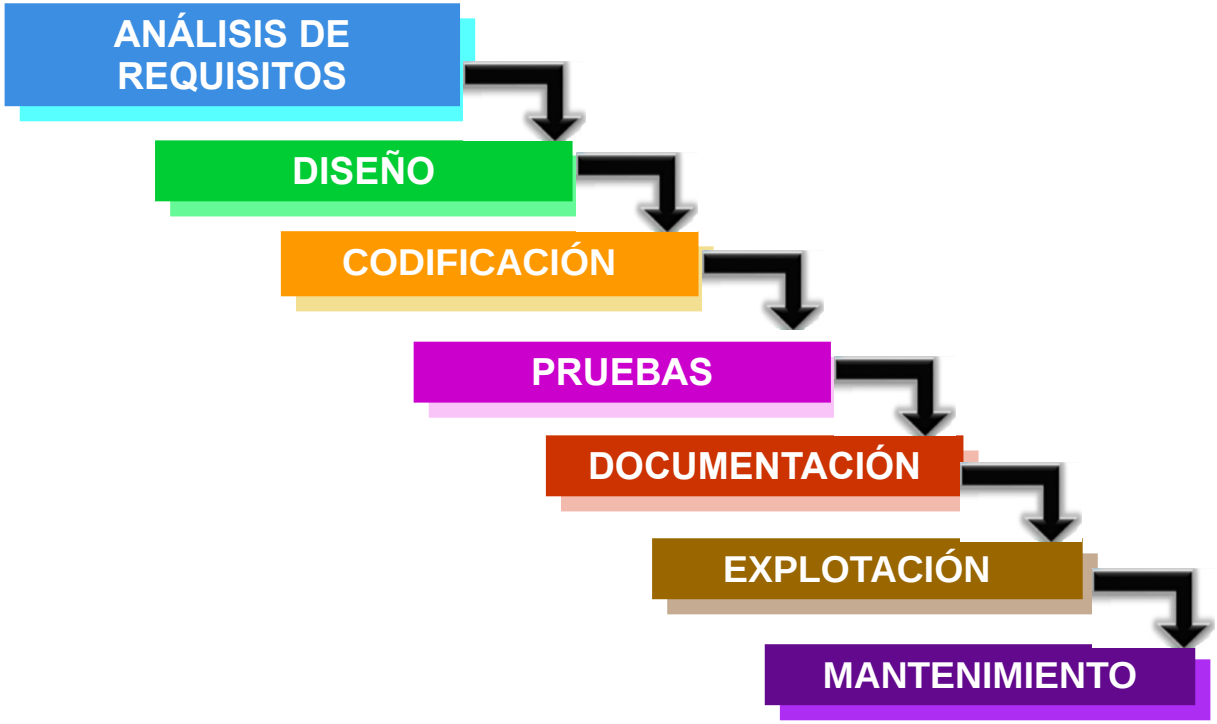
5. Fases en el desarrollo y ejecución del software.

Fases en el Desarrollo de Software

Detallan todo el proceso que ocurre desde que se concibe una idea hasta que un programa está implantado en el ordenador y funcionando.

Ya hemos visto en puntos anteriores que debemos elegir un modelo de ciclo de vida para el desarrollo de nuestro software.

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



5. Fases en el desarrollo y ejecución del software.

5.1 ANÁLISIS

ANÁLISIS DE REQUISITOS

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

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

5.1 ANÁLISIS

ANÁLISIS DE REQUISITOS

¿Qué se hace en esta fase?

Se especifican y analizan los requisitos funcionales y no funcionales del sistema.

Requisitos:

- **Funcionales**: Qué funciones tendrá que realizar la aplicación. Qué respuesta dará la aplicación ante todas las entradas. Cómo se comportará la aplicación en situaciones inesperadas.
- **No funcionales**: Tiempos de respuesta del programa, legislación aplicable, tratamiento ante la simultaneidad de peticiones, etc.

5.1 ANÁLISIS

ANÁLISIS DE REQUISITOS

La culminación de esta fase es el documento ERS (Especificación de Requisitos Software).

En este documento quedan especificados:

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

5.2 DISEÑO

DISEÑO

Durante esta fase, donde ya sabemos lo que hay que hacer, el siguiente paso es **¿Cómo hacerlo?**

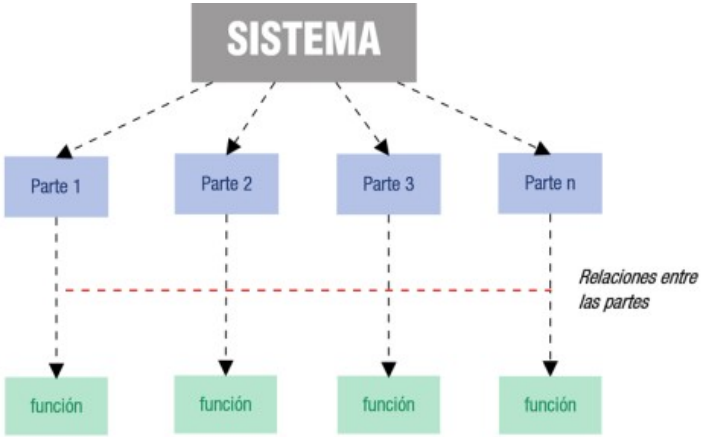
Se debe dividir el sistema en partes y establecer qué relaciones habrá entre ellas.

Decidir qué hará exactamente cada parte.

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

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

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



5.3 CODIFICACIÓN. TIPOS DE CÓDIGO

CODIFICACIÓN

Durante la fase de codificación **se realiza el proceso de programación.** Consiste en elegir un determinado lenguaje de programación, codificar toda la información anterior y llevarlo a código fuente.

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

Las características deseables de todo código son:

- **Modularidad:** que esté dividido en trozos más pequeños.
- **Corrección:** que haga lo que se le pide realmente.
- **Fácil de leer:** para facilitar su desarrollo y mantenimiento futuro.
- **Eficiencia:** que haga un buen uso de los recursos.

5.3 CODIFICACIÓN. TIPOS DE CÓDIGO

CODIFICACIÓN

Durante esta fase de codificación el código pasa por diferentes estados.

- **Código Fuente:** es el escrito por los programadores en algún editor de texto. Se escribe usando algún lenguaje de programación de alto nivel y contiene el conjunto de instrucciones necesarias.
- **Código Objeto:** es el código binario resultado de compilar el código fuente.

  La compilación es la traducción de una sola vez del programa, y se realiza utilizando un compilador. La interpretación es la traducción y ejecución simultánea del programa línea a línea.

  El código objeto no es directamente inteligible por el ser humano, pero tampoco por la computadora. Es un código intermedio entre el código fuente y el ejecutable y sólo existe si el programa se compila, ya que si se interpreta (traducción línea a línea del código) se traduce y se ejecuta en un solo paso.
- **Código Ejecutable:** Es el código binario resultante de enlazar los archivos de código objeto con ciertas rutinas y bibliotecas necesarias. El sistema operativo será el encargado de cargar el código ejecutable en memoria RAM y proceder a ejecutarlo. También es conocido como código máquina y ya sí es directamente inteligible por la computadora.

Actividad

Vas a realizar una infografía en la que expliques de forma visual y clara los estados del código en un programa escrito en Java:

1) Código fuente
   - ➢ Qué es y en qué archivo se guarda.
   - ➢ Ejemplo con HolaMundo.java.
2) Código objeto (Bytecode en Java)
   - ➢ Cómo se obtiene al compilar con javac.
   - ➢ Archivo .class generado.
3) Código ejecutable
   - ➢ Cómo se ejecuta en la JVM con java HolaMundo.
   - ➢ Diferencia con otros lenguajes que generan .exe.

Requisitos de la infografía
- ✔ Debe incluir título llamativo.
- ✔ Un esquema visual del proceso (flechas, pasos, iconos o dibujos).
- ✔ Explicación breve en texto para cada estado.
- ✔ Un ejemplo práctico basado en el programa HolaMundo.java.
- ✔ Diseñada en Canva, Genially, PowerPoint, o la herramienta que prefieras.

Entregable: Archivo de la infografía en formato imagen (PNG/JPG) o PDF.

5.4.1 CÓDIGO FUENTE

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

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

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

Para saber más ve a este **enlace**

5.4.1 CÓDIGO FUENTE

Para obtener el código fuente de una aplicación informática:

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

La culminación de la obtención de código fuente es un documento con la codificación de todos los **módulos** *(Cada parte, con una funcionalidad concreta, en que se divide una aplicación)*, **funciones** *(Parte de código muy pequeña con una finalidad muy concreta.)*, **bibliotecas y procedimientos** *(Igual que las función, pero al ejecutarse no devuelven ningún valor.)* necesarios para codificar la aplicación.

5.4.1 CÓDIGO FUENTE

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

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

1) Código fuente abierto. Es aquel que está disponible para que cualquier usuario pueda estudiarlo, modificarlo o reutilizarlo.
2) Código fuente cerrado. Es aquel que no tenemos permiso para editarlo.

Autoevaluación

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

A) Se elige el Lenguaje de Programación más adecuado y se codifica directamente.

B) Se codifica y después se elige el Lenguaje de Programación más adecuado.

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

5.4.2 CÓDIGO OBJETO

El código objeto es un **código intermedio**. Es el resultado de traducir código fuente a un código equivalente f**ormado por unos y ceros que aún no puede ser ejecutado** directamente por la computadora. Es decir, **es el código resultante de la compilación del código fuente.**

Consiste **en un bytecode** (Código binario resultante de la traducción de código de alto nivel que aún no puede ser ejecutado.) que **está distribuido en varios archivos**, cada uno **de** los cuales corresponde a **cada programa fuente compilado**.

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

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

5.4.2 CÓDIGO OBJETO

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

A) Compilación: El proceso de traducción se realiza sobre todo el código fuente, en un solo paso. Se crea código objeto que habrá que enlazar. El software responsable se llama compilador (Software que traduce, de una sola vez, un programa escrito en un lenguaje de programación de alto nivel en su equivalente en lenguaje máquina.).

B) Interpretación: El proceso de traducción del código fuente se realiza línea a línea y se ejecuta simultáneamente. No existe código objeto intermedio. El software responsable se llama intérprete (Software que traduce, instrucción a instrucción, un programa escrito en un lenguaje de alto nivel en su equivalente en lenguaje máquina). El proceso de traducción es más lento que en el caso de la compilación, pero es recomendable cuando el programador es inexperto, ya que da la detección de errores es más detallada.

5.4.2 CÓDIGO EJECUTABLE

El **código ejecutable**, resultado de enlazar los archivos de código objeto, consta de **un único archivo** que puede ser **directamente ejecutado por la computadora**. **No necesita ninguna aplicación externa**. Este archivo **es ejecutado y controlado por el sistema operativo.**

Para obtener un sólo archivo ejecutable, habrá que enlazar todos los archivos de código objeto, a través de un software llamado linker (Enlazador. Pequeño software encargado de unir archivos para generar un programa ejecutable.) y obtener así un único archivo que ya sí es ejecutable directamente por la computadora.

5.4.2 CÓDIGO EJECUTABLE

En el esquema de generación de código ejecutable, vemos el proceso completo para la generación de ejecutables.

A partir de un editor, escribimos el lenguaje fuente con algún Lenguaje de programación. (En el ejemplo, se usa Java).

A continuación, el código fuente se compila obteniendo código objeto o bytecode.

Ese bytecode, a través de la máquina virtual (se verá en el siguiente punto), pasa a código máquina, ya directamente ejecutable por la computadora.

5.5 MÁQUINAS VIRTUALES

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

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

Con el uso de máquinas virtuales podremos desarrollar y ejecutar una aplicación sobre cualquier equipo, independientemente de las características concretas de los componentes físicos instalados. **Esto garantiza la portabilidad** *(capacidad de un programa para ser ejecutado en cualquier arquitectura física de un equipo)* **de las aplicaciones.**

5.5 MÁQUINAS VIRTUALES

Las funciones principales de una máquina virtual son las siguientes:

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

5.5 MÁQUINAS VIRTUALES

CARACTERÍSTICAS DE LA MÁQUINA VIRTUAL

Cuando el **código fuente** se **compila** se obtiene **código objeto** (bytecode, código intermedio). **Para ejecutarlo en cualquier máquina** se requiere tener independencia respecto al hardware concreto que se vaya a utilizar.

Para ello, **la máquina virtual aísla la aplicación de los detalles físicos del equipo en cuestión.**

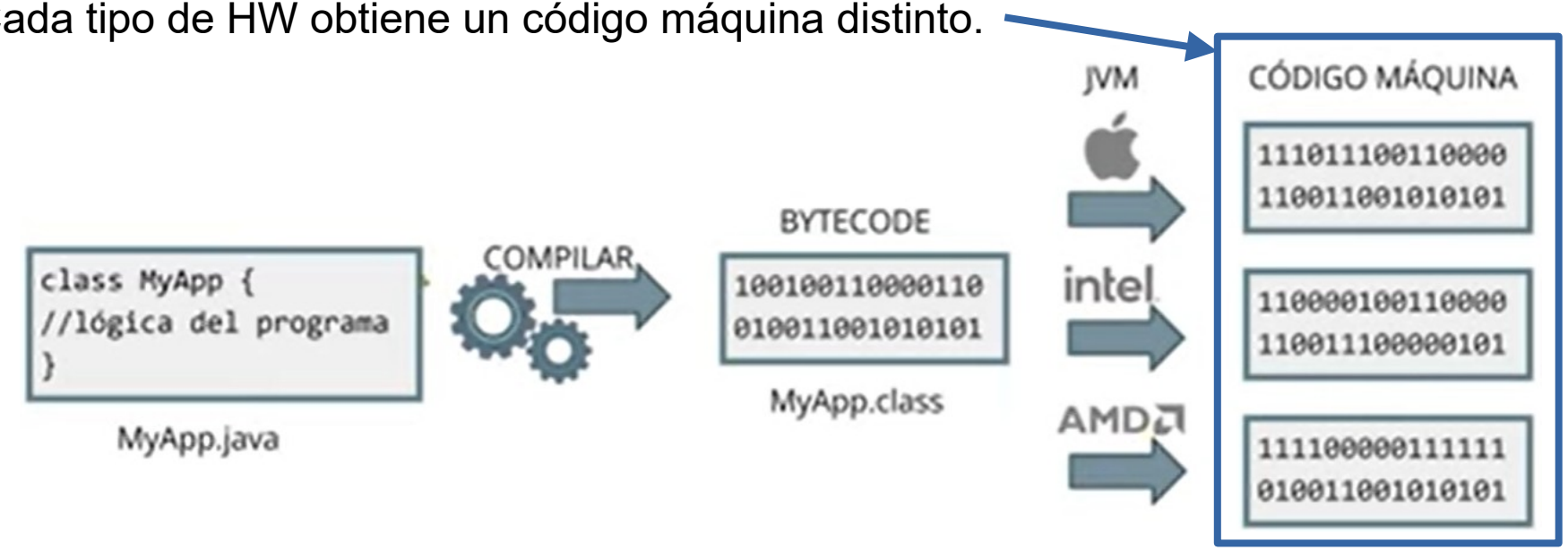
Funciona como una capa de software de bajo nivel y **actúa como puente entre el bytecode de la aplicación y los dispositivos físicos del sistema**.

La Máquina Virtual verifica todo el bytecode antes de ejecutarlo.

5.5 MÁQUINAS VIRTUALES

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

Cada tipo de HW obtiene un código máquina distinto.



5.5.1 Frameworks

Un **framework** (también llamado plataforma, entorno o marco de trabajo para el desarrollo rápido de aplicaciones) es una herramienta que facilita la labor del programador, ya que permite crear proyectos sin necesidad de empezar completamente desde cero.

Un framework es un entorno de software que incluye programas de soporte, bibliotecas, lenguajes interpretados y otros recursos que ayudan a construir, organizar y conectar los distintos módulos o componentes de un proyecto.

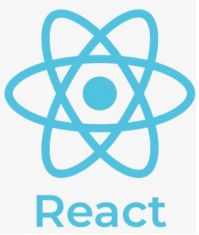
5.5.1 Frameworks

Un **framework** (también llamado plataforma, entorno o marco de trabajo para el desarrollo rápido de aplicaciones) es una herramienta que facilita la labor del programador, ya que permite crear proyectos sin necesidad de empezar completamente desde cero.

Se trata de un entorno de software que incluye programas de soporte, bibliotecas, lenguajes interpretados y otros recursos que ayudan a construir, organizar y conectar los distintos módulos o componentes de un proyecto.

Algunos ejemplos destacados de frameworks populares incluyen:

Bootstrap: Un framework **frontend** ampliamente utilizado que ofrece una variedad de componentes preestablecidos y estilos CSS para facilitar el diseño responsivo.



React: Un framework de JavaScript desarrollado por Facebook, altamente eficiente para crear interfaces de usuario interactivas.

React: Un framework de JavaScript desarrollado por Facebook, altamente eficiente para crear interfaces de usuario interactivas.

Un framework **backend** en PHP que sigue el patrón MVC y proporciona herramientas poderosas para el desarrollo rápido y seguro de aplicaciones web.

5.5.1 Frameworks

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

Ventajas de utilizar un framework:

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

Inconvenientes:

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

5.5.2 Entornos de Ejecución (Runtime Environments)

Un entorno de ejecución es simplemente el “lugar” donde un programa funciona. Incluye todo lo que necesita para ejecutarse: el sistema operativo, la memoria, librerías, e incluso, a veces, una máquina virtual.

- ➔Con máquina virtual: el entorno de ejecución es como un “servicio” que simula una computadora dentro de otra (por ejemplo, la JVM para Java).
- ➔Sin máquina virtual: el programa usa directamente el hardware y el sistema operativo (por ejemplo, un programa en C en Windows).

Un entorno de ejecución es la base de software y recursos que permite que un programa se ejecute correctamente

5.5.2 Entornos de Ejecución (Runtime Environments)

Durante la ejecución de un programa, el entorno de ejecución se encarga de:

- Ejecutar o interpretar el código.
- Gestionar la memoria (asignar y liberar espacio).
- Manejar entradas/salidas (pantalla, archivos, red).
- Controlar errores y excepciones.
- Proveer acceso a librerías y funciones del sistema.

5.5.3 Java Runtime Environments (JRE)

El Java Runtime Environment (JRE) es el entorno que permite ejecutar aplicaciones escritas en Java.

¿Qué es?

- Es un conjunto de herramientas y librerías que necesita el sistema operativo para que un programa Java pueda funcionar.
- No **sirve** para programar en Java (para eso está el JDK), **solo para ejecutar**.

¿Qué contiene el JRE?

- Java Virtual Machine (JVM): interpreta el bytecode (el código compilado en .class) y lo ejecuta en tu sistema.
- Librerías estándar de Java: colecciones de clases (como java.util, java.io, java.net, etc.) que los programas Java usan.
- Archivos de configuración y recursos que necesita el sistema de ejecución.

¿Cómo funciona?

- El programador compila el código Java con el compilador (javac), lo que genera archivos .class con bytecode.
- Ese bytecode no es código máquina, por lo que necesita la JVM.
- La JVM, dentro del JRE, traduce el bytecode al lenguaje del sistema operativo (Windows, Linux, macOS, etc.) y lo ejecuta.

5.6 PRUEBAS

PRUEBAS

¿Qué son las pruebas en el ciclo de vida del software?

Una vez que el software está construido (es decir, ya se ha programado), no se puede entregar directamente.

Primero debe **probarse** para comprobar que:

- Funciona como se espera (**verificación**).
- Cumple con lo que el cliente pidió (**validación**).
- **No presenta fallos** en situaciones normales ni en casos límite.

Por eso se utilizan datos de prueba, que incluyen:

- Valores normales (que deberían funcionar bien).
- Valores límite (muy altos, muy bajos, vacíos, etc.).
- Valores incorrectos (para comprobar que el sistema responde con mensajes de error adecuados).

5.6 PRUEBAS

PRUEBAS

Tipos principales de pruebas

Pruebas unitarias:

Consisten en probar cada parte individual del software (funciones, métodos, clases o módulos) para verificar que cumplen correctamente su función, de forma aislada e independiente del resto del sistema.

El objetivo de estas pruebas unitarias no es otro que el detectar errores cuanto antes, en las piezas más pequeñas del programa, antes de combinarlas con otras.

En Java el entorno de pruebas es JUnit.

Pruebas de integración

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

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

5.7 DOCUMENTACIÓN

DOCUMENTACIÓN

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

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

Para dar toda la información a los usuarios de nuestro software y poder acometer futuras revisiones del proyecto.

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

Distinguimos tres grandes documentos en el desarrollo de software:

5.7 DOCUMENTACIÓN

DOCUMENTACIÓN

Distinguimos tres grandes documentos en el desarrollo de software:

Documentos a elaborar en el proceso de desarrollo de software			
	Guía Técnica	**Guía de Uso**	**Guía de Instalación**
Quedan reflejados	• El diseño de la aplicación. • La codificación de los programas. • Las pruebas realizadas.	• Descripción de la funcionalidad de la aplicación. • Forma de comenzar a ejecutar la aplicación. • Ejemplos de uso del programa. • Requerimientos software de la aplicación. • Solución de los posibles problemas que se pueden presentar.	Toda la información necesaria para: • Puesta en marcha. • Explotación. • Seguridad del sistema.
¿A quién va dirigido?	Al personal técnico en informática (analistas y programadores).	A los usuarios que van a usar la aplicación (clientes).	Al personal informático responsable de la instalación, en colaboración con los usuarios que van a usar la aplicación (clientes).
¿Cuál es su objetivo?	Facilitar un correcto desarrollo, realizar correcciones en los programas y permitir un mantenimiento futuro.	Dar a los usuarios finales toda la información necesaria para utilizar la aplicación.	Dar toda la información necesaria para garantizar que la implantación de la aplicación se realice de forma segura, confiable y precisa.

5.8 EXPLOTACIÓN

EXPLOTACIÓN

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

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

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

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

5.8 EXPLOTACIÓN

EXPLOTACIÓN

Es recomendable que los futuros clientes estén presentes en este momento e irles comentando cómo se va planteando la instalación. En este momento, se suelen llevan a cabo las Beta Test, que son las últimas pruebas que se realizan en los propios equipos del cliente y bajo cargas normales de trabajo.

Una vez instalada, pasamos a la fase de configuración.

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

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

5.9 MANTENIMIENTO

MANTENIMIENTO

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

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

La etapa de mantenimiento se define como el proceso de control, mejora y optimización del software. Es la etapa más larga de todo el ciclo de vida del software.

Por su naturaleza, el software es cambiante y deberá actualizarse y evolucionar con el tiempo.

Deberá ir adaptándose de forma paralela a las mejoras del hardware en el mercado y afrontar situaciones nuevas que no existían cuando el software se construyó.

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

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

5.9 MANTENIMIENTO

MANTENIMIENTO

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

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

- **Perfectivos**: Para mejorar la funcionalidad del software.
- **Evolutivos**: El cliente tendrá en el futuro nuevas necesidades. Por tanto, serán necesarias modificaciones, expansiones o eliminaciones de código.
- **Adaptativos**: Modificaciones, actualizaciones... para adaptarse a las nuevas tendencias del mercado, a nuevos componentes hardware, etc.
- **Correctivos**: La aplicación tendrá errores en el futuro (sería utópico pensar lo contrario).