I. E. S. MARE NOSTRUM
Beato Fco. Castelló Aleu, s/n
☎ 965936520 • 965936521
03008 ALICANTE
iesmarenostrum.edu.gva.es
03012736@edu.gva.es

UNIDAD 1
DESARROLLO DE SOFTWARE

I. E. S. MARE NOSTRUM
Beato Fco. Castelló Aleu, s/n
☎ 965936520 • 965936521
03008 ALICANTE
iesmarenostrum.edu.gva.es
03012736@edu.gva.es

Contenido

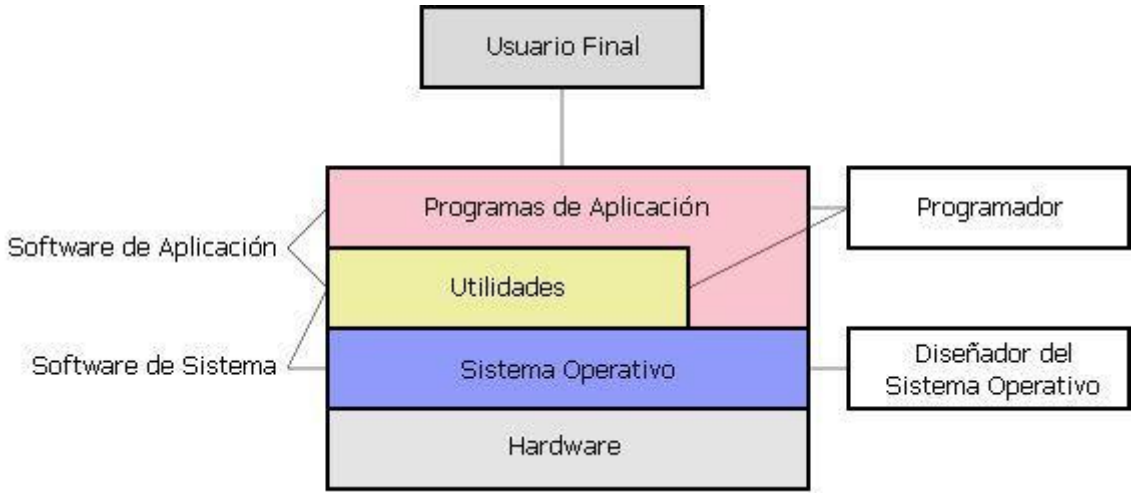
1 SOFTWARE. PROGRAMA. APLICACIÓN INFORMÁTICA.

Un programa informático es un conjunto de instrucciones que una vez ejecutadas realizarán una o varias tareas en una computadora. Sin programas, estas máquinas no pueden funcionar. Al conjunto de todos los programas se le denomina software, que más genéricamente se refiere al equipamiento lógico o soporte lógico de una computadora digital.

El software es el nexo de unión entre el hardware (computadora) y el hombre (usuario). La computadora, por sí sola, no puede comunicarse con el usuario y viceversa, ya que los separa la barrera del lenguaje. El software trata de eliminar esa barrera, estableciendo procedimientos de comunicación entre el hombre y la máquina; es decir, el software obra como un intermediario entre el hardware y el hombre.

Existen diferentes tipos de software: software de sistema (sistemas operativos) y software de aplicación.

Un sistema operativo (S.O.) es el conjunto mínimo de programas que permiten aprovechar toda la funcionalidad del ordenador (hardware), aunque sin realizar ningún tipo de función específica. El S.O. permite la utilización de periféricos (ratón, teclado, impresora, ...), el acceso a discos, etc. Ejemplos de S.O. son Microsoft Windows, MacOS o las diferentes distribuciones de Linux (Debian, Ubuntu, CentOS, RedHat, ...)



Una aplicación es un tipo de programa informático diseñado para facilitar al usuario la realización de un determinado tipo de trabajo. Las aplicaciones suelen diseñarse para la automatización de ciertas tareas complicadas o tediosas como pueden ser la contabilidad, la redacción de documentos, o la gestión de un almacén. Algunos ejemplos de programas de aplicación son los procesadores de textos, hojas de cálculo, y base de datos.

2 LENGUAJE DE PROGRAMACIÓN

Los ordenadores, a diferencia de los seres humanos, no son capaces de procesar el lenguaje natural ("guarda este archivo en mi usb extraíble"). Por el contrario, poseen un conjunto reducido de instrucciones que son capaces de ejecutar. A este conjunto reducido de instrucciones se les denomina "lenguaje máquina".

El principal problema del lenguaje máquina es su dificultad para entenderlo, ya que suele manejar información como direcciones de memoria o números hexadecimales. Por eso, los seres humanos hemos diseñado lenguajes que, siendo todavía cercanos a la máquina, permitan a un ser humano su escritura de manera sencilla. Estos lenguajes son los lenguajes de programación.

Un lenguaje de programación es un lenguaje diseñado para describir el conjunto de acciones consecutivas que un ordenador debe ejecutar. Por lo tanto, un lenguaje de programación es un modo práctico para que los seres humanos puedan dar instrucciones a un ordenador. Hay muchísimos, de toda clase, tipos y características, inventados para facilitar el abordaje de distintos problemas, el mantenimiento del software, su reutilización, mejorar la productividad, etc.

TIPOS DE LENGUAJES DE PROGRAMACIÓN

Los lenguajes de programación se pueden clasificar según varios criterios. Hay que tener en cuenta también, que, en la práctica, la mayoría de lenguajes no pueden ser puramente clasificados en una categoría, pues surgen incorporando ideas de otros lenguajes y de otras filosofías de programación, pero no importa al establecer las clasificaciones, pues el auténtico objetivo de las mismas es mostrar los rangos, las posibilidades y tipos de lenguajes que hay.

Nivel de abstracción.

Según el nivel de abstracción, es decir, según el grado de cercanía a la máquina:

- Lenguajes de **bajo nivel**: La programación se realiza teniendo muy en cuenta las características del procesador. Ejemplo: Lenguajes ensamblador.

- Lenguajes de **alto nivel**: Más parecidos al lenguaje humano. Manejan conceptos, tipos de datos, etc., de una manera cercana al pensamiento humano ignorando (abstrayéndose) el funcionamiento de la máquina. Ejemplos: C, Java, Ruby on Rails.

Evolución histórica.

Con el paso del tiempo, se va incrementando el **nivel de abstracción**, pero en la práctica, los de una generación no terminan de sustituir a los de la anterior:

- Lenguajes de **primera generación (1GL)**: Código máquina.
- Lenguajes de **segunda generación (2GL)**: Lenguajes ensamblador.
- Lenguajes de **tercera generación (3GL)**: La mayoría de los lenguajes modernos, diseñados para facilitar la programación a los humanos. Ejemplos: C, Java.
- Lenguajes de **cuarta generación (4GL)**: Diseñados con un propósito concreto, o sea, para abordar un tipo concreto de problemas. Ejemplos: NATURAL, Mathematica.
- Lenguajes de **quinta generación** (5GL): La intención es que el programador establezca qué problema ha de ser resuelto y las condiciones a reunir, y la máquina lo resuelve. Se usan en inteligencia artificial. Ejemplo: Prolog.

Modo de abordar la tarea a realizar.

Según la manera de abordar la tarea a realizar, pueden ser:

- Lenguajes **imperativos**: Indican cómo hay que hacer la tarea, es decir, expresan los pasos a realizar. Ejemplo: C.
- Lenguajes **declarativos**: Indican qué solución hay que obtener, pero no cómo llegar a ella. Ejemplos: Lisp, Prolog, SQL (para consultar bases de datos).

Paradigma de programación.

El **paradigma de programación** es el estilo de programación empleado. Algunos lenguajes soportan varios paradigmas, y otros sólo uno. Se puede decir que históricamente han ido apareciendo para facilitar la tarea de programar según el tipo de problema a abordar, o para facilitar el mantenimiento del software, o por otra cuestión similar, por lo que todos corresponden a lenguajes de alto nivel (o nivel medio), estando los lenguajes ensambladores "atados" a la arquitectura de su procesador correspondiente. Los principales son:

- Lenguajes de **programación procedural**: Divide el problema en partes más pequeñas, que serán realizadas por subprogramas (subrutinas, funciones, procedimientos), que se llaman unas a otras para ser ejecutadas. Ejemplos: C, Pascal.

- Lenguajes de **programación orientada a objetos**: Crean un sistema de clases y objetos siguiendo el ejemplo del mundo real, en el que unos objetos realizan acciones y se comunican con otros objetos. Ejemplos: C++, Java.

- Lenguajes de **programación funcional**: La tarea se realiza evaluando funciones, (como en Matemáticas), de manera recursiva. Ejemplo: Lisp.

- Lenguajes de **programación lógica**: La tarea a realizar se expresa empleando lógica formal matemática. Expresa qué computar. Ejemplo: Prolog.

LENGUAJES MÁS DIFUNDIDOS

Algunos de los lenguajes más difundidos son:

- **BASIC**, que durante mucho tiempo se ha considerado un buen lenguaje para comenzar a aprender, por su sencillez, aunque se podía tender a crear programas poco legibles. A pesar de esta "sencillez" hay versiones muy potentes, incluso para programar en entornos gráficos como Windows.

- **COBOL**, que fue muy utilizado para negocios (para crear software de gestión, que tuviese que manipular grandes cantidades de datos), aunque últimamente está bastante en desuso.

- **FORTRAN**, concebido para ingeniería y aplicaciones que requirieran un uso exhaustivo de operaciones matemáticas. También va quedando desplazado.

- **Ensamblador**, muy cercano al código máquina (es un lenguaje de "bajo nivel"), pero sustituye las secuencias de ceros y unos (bits) por palabras más fáciles de recordar, como MOV, ADD, CALL o JUMP.

- **C,** uno de los mejor considerados actualmente (junto con C++ y Java, que mencionaremos a continuación), porque no es demasiado difícil de aprender y permite un grado de control del ordenador muy alto, combinando características de lenguajes de alto y bajo nivel. Además, es muy transportable: existe un estándar, el ANSI C, lo que asegura que se pueden convertir programas en C de un ordenador a otro o de un sistema operativo a otro con bastante menos esfuerzo que en otros lenguajes.

- **C++**, un lenguaje desarrollado a partir de C, que permite Programación Orientada a Objetos, por lo que resulta más adecuado para proyectos de una cierta envergadura. Creado por Bjarne Stroustrup.

- **Java**, desarrollado a su vez a partir de C++, que elimina algunos de sus inconvenientes, y ha alcanzado una gran difusión gracias a su empleo en Internet.

- **PHP,** es un lenguaje de programación interpretado, diseñado originalmente para la creación de páginas web dinámicas. Se usa principalmente para la interpretación del lado del servidor (*server-side scripting*) pero actualmente puede ser utilizado desde una interfaz de línea de comandos o en la creación de otros tipos de programas.

- **Python,** es un lenguaje de programación de alto nivel cuya filosofía hace hincapié en una sintaxis muy limpia y que favorezca un código legible. Se trata de un lenguaje de programación multiparadigma ya que soporta orientación a objetos, programación imperativa y, en menor medida, programación funcional.

3 CÓDIGO FUENTE, OBJETO Y EJECUTABLE.

CÓDIGO FUENTE

El código fuente de un programa informático (o software) es un conjunto de líneas de texto que constituyen las instrucciones que debe seguir la computadora para ejecutar dicho programa. por tanto, en el código fuente de un programa está descrito por completo su funcionamiento.

El código fuente de un programa está escrito por un programador en algún lenguaje de programación, pero en este primer estado no es directamente ejecutable por la computadora, sino que debe ser traducido a otro lenguaje (el lenguaje máquina o código objeto) que sí pueda ser ejecutado por el hardware de la computadora. Para esta traducción se usan los llamados compiladores, ensambladores, intérpretes y otros sistemas de traducción.

CÓDIGO OBJETO

En programación, se llama código objeto al código que resulta de la compilación del código fuente. Este código por lo general no es directamente utilizable, teniendo que ser o bien enlazado o bien interpretado por alguna máquina virtual.

Consiste en lenguaje máquina o bytecode y se distribuye en varios archivos que corresponden a cada código fuente compilado. Para obtener un programa ejecutable se han de enlazar todos los archivos de código objeto con un programa llamado enlazador (linker).

CÓDIGO EJECUTABLE

El código ejecutable es el código resultante del proceso de enlazado (link), y es directamente interpretable por la máquina.

Generalmente se confunde con el código objeto, ya que al leer su estructura se comprende como símbolos. Pero en realidad, este código se encuentra empaquetado y listo para ser ejecutado en cualquier computadora. Generalmente vienen con la extensión EXE o COM, si los han de ejecutar computadoras con Sistema Operativo de Windows o con bits de marca que trae Linux para ser ejecutable.

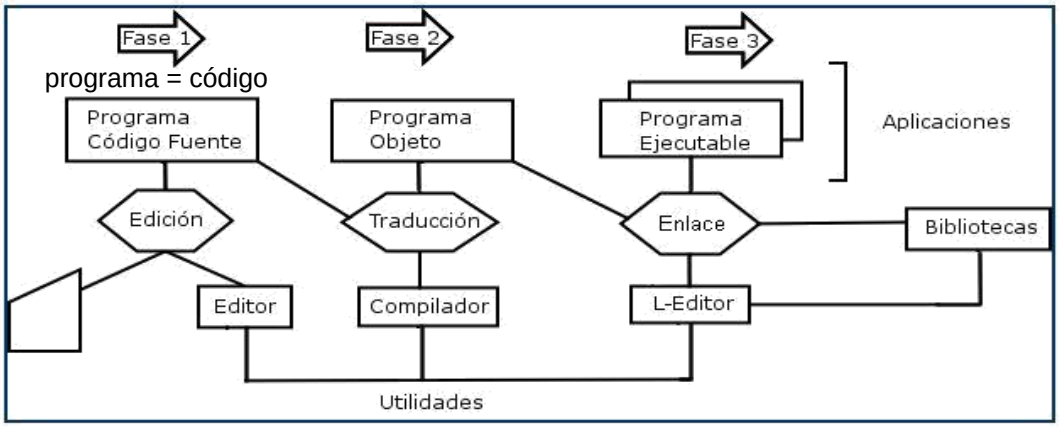
El beneficio que esto trae es que, al tener el código ejecutable, podemos saber que la compilación fue realizada correctamente y que el programa, si no tiene errores de manejo, puede funcionar correctamente, ya que está libre de errores de variables, signos y demás.

PROCESO DE OBTENCIÓN DE CÓDIGO EJECUTABLE A PARTIR DE CÓDIGO FUENTE

El proceso de obtención de código ejecutable a partir de código fuente se denomina *compilación*. El proceso de compilación incluye dos fases:

- La traducción del código fuente al código objeto, lo que propiamente denominamos compilación.

- La transformación del código objeto en código ejecutable, proceso que se denomina 'enlazado'.



Para la realización de este proceso se utilizan una serie de herramientas:

- Compilador
- Intérprete
- Enlazador
- Depurador

Compilador

El compilador es la herramienta, dentro de un entorno de desarrollo, encargada de traducir el código fuente a código objeto. Como una parte fundamental de este proceso de traducción, el compilador le hace notar al usuario la presencia de errores en el código fuente del programa.

Un aspecto fundamental es que el proceso de compilación no se detiene en el primer error que se encuentra en el código, sino que el compilador continúa la compilación para seguir detectando nuevos errores. Lógicamente, y aunque el proceso de detección de errores continúe, en el momento en que se detecta un error se cancela la generación de código objeto.

Un programa que ha sido compilado puede correr por sí sólo, pues en el proceso de compilación se transformó en otro lenguaje (lenguaje máquina).

Intérprete

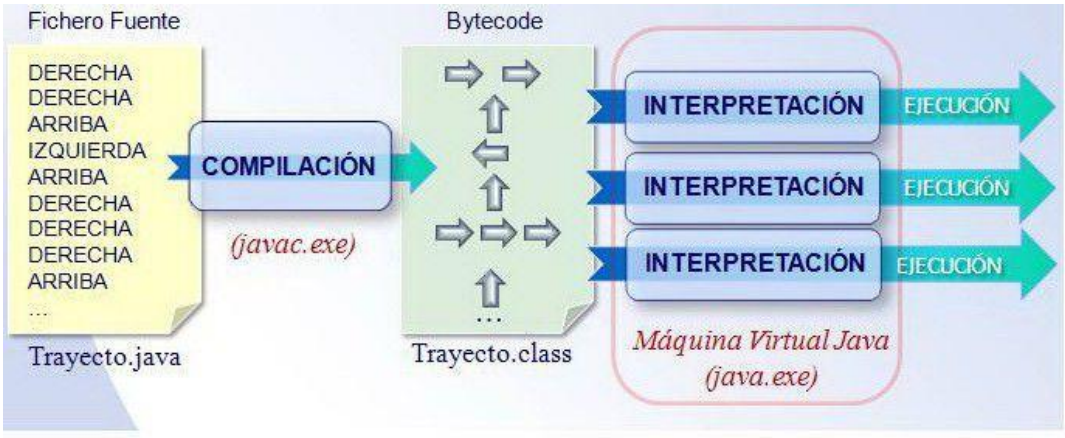
Los intérpretes, a diferencia de los compiladores, no producen un lenguaje objeto. Un intérprete lee el código como está escrito e inmediatamente lo convierte en acciones; es decir, lo ejecuta en ese instante. Por tanto, y a diferencia de los compiladores, un intérprete se detiene en cuanto encuentra un error en el código.

Un intérprete traduce el programa cuando lo lee, convirtiendo el código del programa directamente en acciones. La ventaja del intérprete es que dado cualquier programa se puede interpretar en cualquier plataforma (sistema operativo). En cambio, el archivo generado por el compilador solo funciona en la plataforma en donde se le ha creado. Sin embargo, hablando de la velocidad de ejecución, un archivo compilado es de 10 a 20 veces más rápido que un archivo interpretado.

Compilador + Intérprete

El programa es compilado la primera vez a un formato intermedio. El archivo resultante debe ser interpretado cada vez que desee ejecutarse.

El lenguaje Java, primero pasa por una fase de compilación en la que el código fuente se transforma en "bytecode", y este "bytecode" puede ser ejecutado luego (interpretado) en ordenadores con distintas arquitecturas (procesadores) que tengan todos instalados la misma "máquina virtual" Java.



Depuradores

El depurador es un programa independiente del editor, el compilador y el enlazador. Suele estar integrado con los otros tres, de modo que desde el entorno de

programación se puede lanzar cualquiera de los programas, pero también se puede usar por separado.

El depurador es una herramienta fundamental para localizar y corregir los errores en tiempo de ejecución.

4 CICLO DE VIDA DEL SOFTWARE

El ciclo de vida es el conjunto de fases por las que pasa el sistema que se está desarrollando desde que nace la idea inicial hasta que el software es retirado o remplazado (muere). También se denomina a veces paradigma.

Un ciclo de vida para un proyecto se compone de fases sucesivas compuestas por tareas que se pueden planificar. Según el modelo de ciclo de vida, la sucesión de fases puede ampliarse con bucles de realimentación, de manera que lo que conceptualmente se considera una misma fase se pueda ejecutar más de una vez a lo largo de un proyecto, recibiendo en cada pasada de ejecución aportaciones a los resultados intermedios que se van produciendo (realimentación).

Las fases o etapas del desarrollo de una aplicación son:

- Análisis.
- Diseño.
- Codificación y pruebas.
- Implantación o explotación.
- Mantenimiento.

Se retroalimentan.
No por haber pasado el Análisis, nunca vuelves a él.

ANÁLISIS

Suministrado por el CLIENTE

Es la fase en que se recogen los requisitos que debe cumplir la aplicación. Dichos requisitos deben ser suministrados por el usuario. En esta fase se establece el producto a desarrollar, siendo necesario especificar los procesos y estructuras de datos que se van a emplear. Debe existir una gran comunicación entre el usuario y el analista para poder conocer todas las necesidades que precisa la aplicación. En el caso de falta de información por parte del usuario se puede recurrir al desarrollo de prototipos para saber con más precisión sus requerimientos.

Requisitos funcionales y no funcionales

A grandes rasgos, podemos dividir los requisitos en funcionales y no funcionales. Los

requisitos funcionales son las operaciones deseadas de un programa o sistema. Normalmente, un requisito funcional es una funcionalidad básica o un comportamiento deseado documentado de forma clara y cuantitativa. Nos indicarán qué funciones tendrá que realizar la aplicación, qué respuesta dará la aplicación ante todas las entradas o cómo se comportará la aplicación en situaciones inesperadas. Por ejemplo, al describir los requisitos funcionales de un frasco, un requisito funcional sería que contenga un líquido y tenga una parte superior roscada para sellar la mermelada para una mejor conservación. Los requisitos funcionales usualmente quedan descritos en un DFD o un diagrama de casos de uso.

Los requisitos funcionales en la ingeniería de sistemas se complementan con requisitos técnicos, que también se conocen como requisitos no funcionales. Los requisitos no funcionales definen lo que se requiere para entregar la función o el comportamiento deseado de un sistema a los estándares de un usuario. Los requisitos técnicos pueden ser factores de rendimiento, accesibilidad, versatilidad, registro, control y respaldo. En el ejemplo anterior del tarro, requisitos no funcionales podrían ser tener cristal irrompible o coloreado, por ejemplo.

El gran problema en la recogida de requisitos suele ser la ambigüedad que se genera en el uso del lenguaje natural, que, por lo general, es el que empleará el cliente para comunicarlos. En el análisis estructurado se pueden emplear varias técnicas para la recogida de requisitos como:

- *Diagramas de flujo de datos:* Sirven para conocer el comportamiento del sistema mediante representaciones gráficas.
- *Modelos de datos:* Sirven para conocer las estructuras de datos y sus características. (Entidad relación y formas normales)
- *Diccionario de datos*: Sirven para describir todos los objetos utilizados en los gráficos, así como las estructuras de datos.
- *Definición de los interfaces de usuario:* Sirven para determinar la información de entrada y salida de datos.

Si utilizamos el paradigma de programación orientado a objetos (POO), es posible utilizar para esta etapa de análisis el Lenguaje Unificado de Modelado (UML), que contempla tanto el diseño estático (datos) como el diseño funcional.

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

- *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.*

DISEÑO

En esta fase se alcanza con mayor precisión una solución óptima de la aplicación, teniendo en cuenta los recursos físicos del sistema (tipo de ordenador, periféricos, comunicaciones, etc.) y los recursos lógicos (sistema operativo, programas de utilidad, bases de datos, etc.).

En el diseño estructurado se pueden definir estas etapas:

- *Diseño externo:* Se especifican los formatos de información de entrada y salida. (pantalla y listados)

- *Diseño de datos:* Establece las estructuras de datos de acuerdo con su soporte físico y lógico. (estructuras en memoria, ficheros y hojas de datos)

- *Diseño modular:* Es una técnica de representación en la que se refleja de forma descendente la división de la aplicación en módulos. Está basado en diagramas de flujo de datos obtenidos en el análisis.

- *Diseño procedimental:* Establece las especificaciones para cada módulo, escribiendo el algoritmo necesario que permita posteriormente una rápida codificación. Se emplean técnicas de programación estructurada, normalmente ordinogramas y pseudocódigo.

Al final de esta etapa se obtiene el denominado cuaderno de carga.

CODIFICACIÓN Y PRUEBAS

Consiste en traducir los resultados obtenidos a un determinado lenguaje de programación, teniendo en cuenta las especificaciones obtenidas en el cuaderno de carga. Se deben de realizar las pruebas necesarias para comprobar la calidad y estabilidad del programa.

Las pruebas se pueden clasificar en:

- *Pruebas unitarias:* Sirven para comprobar que cada módulo realice bien su tarea.

- *Pruebas de interconexión:* Sirven para comprobar en el programa el buen funcionamiento en conjunto de todos sus módulos.

- *Pruebas de integración:* Sirven para comprobar el funcionamiento correcto del conjunto de programas que forman la aplicación. (el funcionamiento de todo el sistema).

EXPLOTACIÓN

En esta fase se realiza la implantación de la aplicación en el sistema o sistemas físicos donde van a funcionar habitualmente y su puesta en marcha para comprobar el buen funcionamiento.

Las actividades a realizar durante la explotación serían:

- Instalación del/los programa/s.
- Pruebas de aceptación al nuevo sistema.
- Conversión de la información del antiguo sistema al nuevo (si hay una aplicación antigua)
- Eliminación del sistema anterior.

Al final de esta fase se debe de completar la información al usuario respecto al nuevo sistema y su uso. Así como facilitarle toda la documentación necesaria para una correcta explotación del sistema (manual de ayuda, manual de uso, guía de la aplicación, etc.)

MANTENIMIENTO

Esta es la fase que completa el ciclo de vida y en ella nos encargaremos de solventar los posibles errores o deficiencias de la aplicación. Existe la posibilidad de que ciertas aplicaciones necesiten reiniciar el ciclo de vida.

Tipos de mantenimiento:

- *Mantenimiento correctivo:* Consiste en corregir errores no detectados en pruebas anteriores y que aparezcan con el uso normal de la aplicación. Este mantenimiento puede estar incluido en la garantía o mantenimiento de la aplicación.
- *Mantenimiento adaptativo:* Consiste en modificar el programa a causa de cambio de entorno gráfico y lógico en el que estén implantados. (nuevas generaciones de ordenadores, nuevas versiones del sistema operativo, etc.)

- *Mantenimiento perfectivo:* Consiste en una mejora sustancial de la aplicación al recibir por parte de los usuarios propuestas sobre nuevas posibilidades y modificaciones de las existentes.

Los tipos de mantenimiento adaptativo y perfectivo reinician el ciclo de vida, debiendo proceder de nuevo al desarrollo de cada una de sus fases para obtener un nuevo producto.

5 METODOLOGÍAS DE DESARROLLO

Una metodología de desarrollo de software es una vista de las actividades que ocurren durante el desarrollo de software, intenta determinar el orden de las etapas involucradas y los criterios de transición asociados entre estas etapas.

Una metodología de desarrollo del software describe las fases principales de desarrollo de software y ayuda a administrar el progreso del desarrollo.

METODOLOGÍA EN CASCADA

Es un enfoque metodológico que ordena rigurosamente las etapas del ciclo de vida del software, de forma que el inicio de cada etapa debe esperar a la finalización de la inmediatamente anterior.

El modelo en cascada es un proceso de desarrollo secuencial, en el que el desarrollo se ve fluyendo hacia abajo (como una cascada) sobre las fases que componen el ciclo de vida.

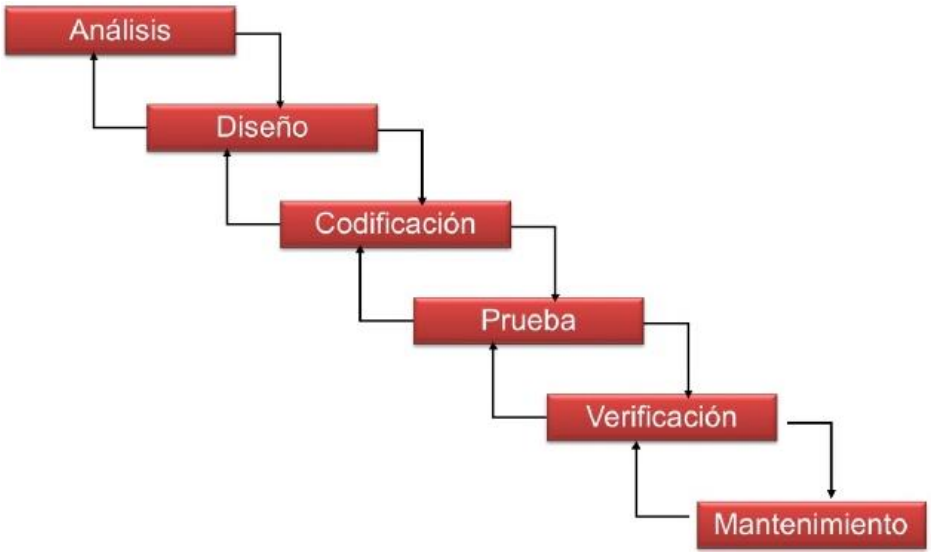
En el modelo original, existían las siguientes fases:

1. Especificación de requisitos (Análisis)
2. Diseño
3. Construcción (Implementación o codificación)
4. Integración
5. Pruebas
6. Instalación
7. Mantenimiento

Para seguir el modelo en cascada, se avanza de una fase a la siguiente en una forma puramente secuencial.

Este modelo ha sido ampliamente criticado desde el ámbito académico y la industria, dado

que, en el modelo teórico, una vez se ha avanzado de fase, no es posible volver a la anterior. Por ello, se debe utilizar para proyectos con los requisitos claramente definidos y sin posibilidad de cambio. Debido a esta falta de flexibilidad, existen múltiples variantes del mismo, como el modelo en cascada con retroalimentación (figura inferior), en el que es posible volver a una fase previa si detectamos un error en cualquiera de las fases.

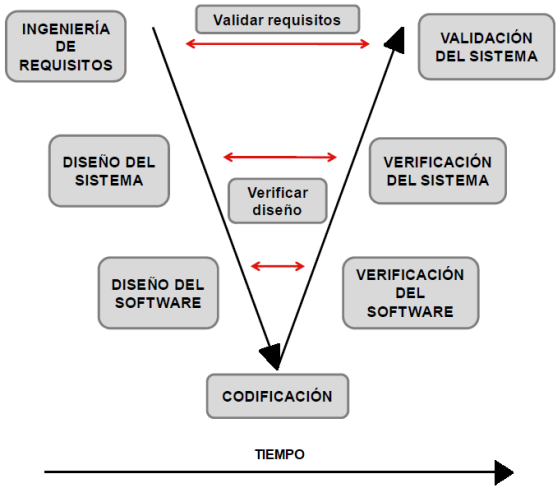


METODOLOGÍA EN V

El modelo en V se desarrolló para terminar con algunos de los problemas que se vieron utilizando el enfoque de cascada tradicional. Los defectos estaban siendo encontrados demasiado tarde en el ciclo de vida, ya que las pruebas no se introducían hasta el final del proyecto.

El modelo en V dice que las pruebas necesitan empezarse lo más pronto posible en el ciclo de vida. También muestra que las pruebas no son sólo una actividad basada en la ejecución. Hay una variedad de actividades que se han de realizar antes del fin de la fase de codificación. Estas actividades deberían ser llevadas a cabo en paralelo con las actividades de desarrollo, y los técnicos de pruebas necesitan trabajar con los desarrolladores y analistas de negocio de tal forma que puedan realizar estas actividades y tareas y producir una serie de entregables de pruebas.

El modelo en V es un proceso que representa la secuencia de pasos en el desarrollo del ciclo de vida de un proyecto. Describe las actividades y resultados que han de ser producidos durante el desarrollo del producto. La parte izquierda de la v representa la descomposición de los requisitos y la creación de las especificaciones del sistema. El lado derecho de la v representa la integración de partes y su verificación. V significa "Validación y Verificación".

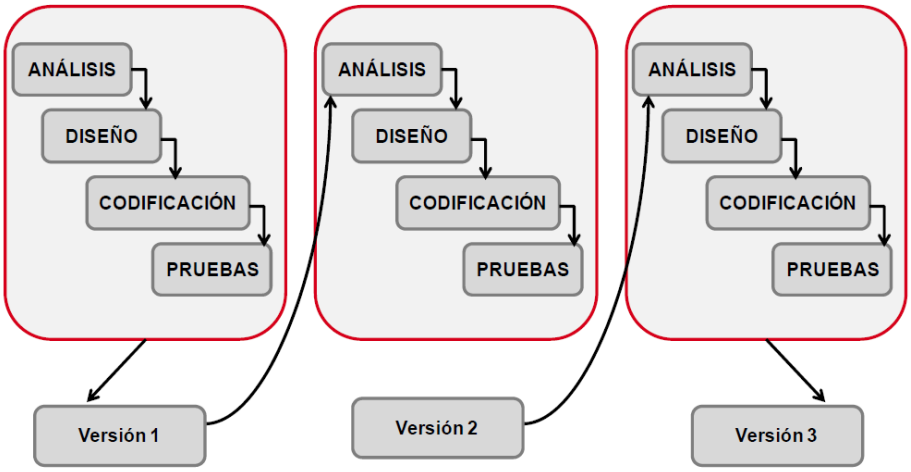


Realmente las etapas individuales del proceso pueden ser casi las mismas que las del modelo en cascada. Sin embargo, hay una gran diferencia. En vez de ir para abajo de una forma lineal las fases del proceso vuelven hacia arriba tras la fase de codificación, formando una v. La razón de esto es que para cada una de las fases de diseño se ha encontrado que hay un homólogo en las fases de pruebas que se correlacionan.

METODOLOGÍA ITERATIVA

Es un modelo derivado del ciclo de vida en cascada. Este modelo busca reducir el riesgo que surge entre las necesidades del usuario y el producto final por malentendidos durante la etapa de recogida de requisitos.

Consiste en la iteración de varios ciclos de vida en cascada. Al final de cada iteración se le entrega al cliente una versión mejorada o con mayores funcionalidades del producto. El cliente es quien, después de cada iteración, evalúa el producto y lo corrige o propone mejoras. Estas iteraciones se repetirán hasta obtener un producto que satisfaga las necesidades del cliente.

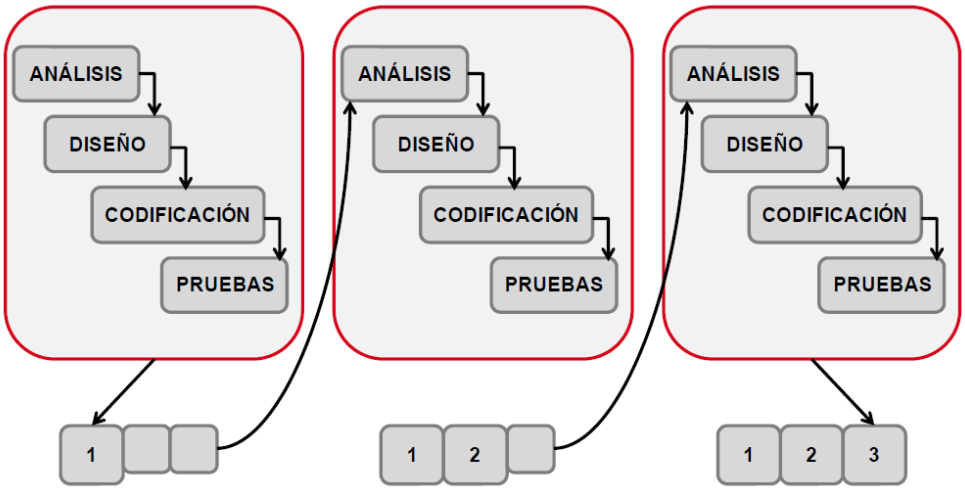


Este modelo se suele utilizar en proyectos en los que los requisitos no están claros por parte del usuario, por lo que se hace necesaria la creación de distintos prototipos para presentarlos y conseguir la conformidad del cliente

METODOLOGÍA INCREMENTAL

El modelo incremental combina elementos del modelo en cascada con la filosofía iterativa de construcción de prototipos. Se basa en la filosofía de construir incrementando las funcionalidades del programa. Este modelo aplica secuencias lineales de forma escalonada mientras progresa el tiempo en el calendario. Cada secuencia lineal produce un incremento del software.

Cuando se utiliza un modelo incremental, el primer incremento es a menudo un producto esencial, sólo con los requisitos básicos. Este modelo se centra en la entrega de un producto operativo con cada incremento. Los primeros incrementos son versiones incompletas del producto final, pero proporcionan al usuario la funcionalidad que precisa y también una plataforma para la evaluación.



METODOLOGÍA EN ESPIRAL

El desarrollo en espiral es un modelo de ciclo de vida desarrollado por Barry Boehm en 1985, utilizado de forma generalizada en la ingeniería del software. Las actividades de este modelo se conforman en una espiral, cada bucle representa un conjunto de actividades. Las actividades no están fijadas a priori, sino que las siguientes se eligen en función del análisis de riesgos, comenzando por el bucle anterior.

Al ser un modelo de ciclo de vida orientado a la gestión de riesgos se dice que uno de los aspectos fundamentales de su éxito radica en que el equipo que lo aplique tenga la necesaria experiencia y habilidad para detectar y catalogar correctamente riesgos.

Tareas:

Para cada ciclo habrá cuatro actividades:

1. Determinar o fijar objetivos:
    - Fijar también los productos definidos a obtener: requerimientos, especificación, manual de usuario.
    - Fijar las restricciones.
    - Identificar riesgos del proyecto y estrategias alternativas para evitarlos.
    - Hay una cosa que solo se hace una vez: planificación inicial o previa
2. Análisis del riesgo:
    - Estudiar todos los riesgos potenciales y se seleccionan una o varias alternativas propuestas para reducir o eliminar los riesgos
3. Desarrollar, verificar y validar (probar):
    - Tareas de la actividad propia y de prueba.
    - Análisis de alternativas e identificación de resolución de riesgos.
    - Dependiendo del resultado de la evaluación de riesgos, se elige un modelo para el desarrollo, que puede ser cualquiera de los otros existentes. Así, por ejemplo, si los riesgos de la interfaz de usuario son dominantes, un modelo de desarrollo apropiado podría ser la construcción de prototipos evolutivos.
4. Planificar:
    - Revisar todo lo que se ha llevado a cabo, evaluándolo y decidiendo si se continua con las fases siguientes y planificando la próxima actividad.

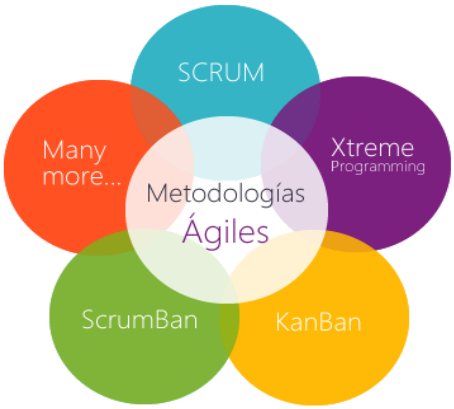
El proceso empieza en la posición central. Desde allí se mueve en el sentido de las agujas del reloj.

PROTOTIPADO

El paradigma de construcción de prototipos comienza con la recolección de requisitos. El desarrollador y el cliente encuentran y definen los objetivos globales para el software, identifican los requisitos conocidos y las áreas del esquema en donde es obligatoria más definición. Entonces aparece un diseño rápido. El diseño rápido se centra en una representación de esos aspectos del software que serán visibles para el usuario/cliente. El diseño rápido lleva a la construcción de un prototipo. El prototipo lo evalúa el cliente/usuario y se utiliza para refinar los requisitos del software a desarrollar. La iteración ocurre cuando el prototipo se pone a punto para satisfacer las necesidades del cliente, permitiendo al mismo tiempo que el desarrollador comprenda mejor lo que se necesita.

METODOLOGÍAS ÁGILES

El desarrollo ágil de software refiere a métodos de ingeniería del software basados en el desarrollo iterativo e incremental, estas metodologías son imprescindibles en un mundo en el que nos exponemos a cambios recurrentemente. Siempre hay que tener en cuenta como programadores que lo que es la última tendencia hoy puede que no exista mañana y por esto existe la metodología ágil donde los requisitos y soluciones evolucionan mediante la colaboración de grupos auto organizados y multidisciplinarios.



Cada metodología tiene características propias y hace hincapié en algunos aspectos más

específicos. A continuación, se resumen algunas de las más conocidas.

SCRUM: Desarrollada por Ken Schwaber, Jeff Sutherland y Mike Beedle. Define un marco para la gestión de proyectos, que se ha utilizado con éxito durante los últimos 10 años. Está especialmente indicada para proyectos con un rápido cambio de requisitos. Sus principales características se pueden resumir en dos:

- El desarrollo de software se realiza mediante iteraciones, denominadas sprints, con una duración de 14-30 días. El resultado de cada sprint es un incremento ejecutable que se muestra al cliente.
- La segunda característica importante son las reuniones a lo largo del proyecto. Éstas son las verdaderas protagonistas, especialmente la reunión diaria de 15 minutos del equipo de desarrollo para coordinación e integración.

Existen una serie de conceptos importantes que se manejarán en esta metodología:

- **Product Backlog**: Todos los elementos o tareas que componen el desarrollo. Es un documento donde se detalla la información de cada tarea a realizar en el desarrollo del software, así como su prioridad.
- **Sprint Backlog**: Todos los elementos y tareas del desarrollo que se comprenden en un sprint concreto, con sus respectivos detalles y prioridades.
- **Sprint Planning**: Reunión en la que se decide la duración del próximo sprint y las tareas que se realizarán en éste en base al feedback del anterior sprint y del cliente o Product Owner. Los sprints no deben durar más de 1 mes.
- **Product Owner**: es un rol: puede ser nuestro cliente (normalmente representante con conocimientos sobre SCRUM, inclusive puede ser el propio dueño del negocio). Es el capitán del barco, el que dirige hacia dónde va el producto en todo momento. Debe tener suficiente poder para tomar cualquier decisión referente al producto. En cada sprint, el Product Owner debe hacer una inversión en desarrollo que tiene que producir un valor. Además, es el encargado de definir las prioridades y organizar el Product Backlog.

- **Scrum Master**: es un rol. Corresponde al jefe de equipo. Puede ser cualquier persona con conocimientos sobre la metodología SCRUM y cómo aplicarla.
- **Development team**: o equipo de desarrollo. Es un rol. Son los desarrolladores que, junto al Scrum master, trabajarán de forma activa en el desarrollo del producto. Como su propio nombre indica, se encargan de desarrollar el producto, autoorganizándose para conseguir entregar un incremento del software al final del ciclo de desarrollo o sprint.

XP o Extreme Programming: La "programación extrema" es un proceso de la Metodología Ágil que se aplica en equipos con muy pocos programadores quienes llevan muy pocos procesos en paralelo. Consiste entonces en diseñar, implementar y programar lo más rápido posible, hasta en casos se recomienda saltar la documentación y los procedimientos tradicionales. Se fundamenta en la capacidad del equipo para comunicarse entre sí y las ganas de aprender de los errores propios inherentes en un programador.

La gran ventaja de XP es su increíble capacidad de respuesta ante imprevistos, aunque por diseño es una metodología que no construye para el largo plazo y para la cual es difícil documentar.

XP es un método estupendo para equipos extremadamente pequeños que se centran en un solo cliente.

Por medio de un cuadro comparativo podremos darnos cuenta de que la metodología ágil es mucho más efectiva y conveniente en el momento de desarrollar software.

Metodología Ágil	Metodología Tradicional
Pocos Artefactos. El modelado es prescindible, modelos desechables.	Más Artefactos. El modelado es esencial, mantenimiento de modelos
Pocos Roles, más genéricos y flexibles	Más Roles, más específicos
No existe un contrato tradicional, debe ser bastante flexible	Existe un contrato prefijado
Cliente es parte del equipo de desarrollo (además in-situ)	El cliente interactúa con el equipo de desarrollo mediante reuniones
Orientada a proyectos pequeños. Corta duración (o entregas frecuentes), equipos pequeños (< 10 integrantes) y trabajando en el mismo sitio	Aplicables a proyectos de cualquier tamaño, pero suelen ser especialmente efectivas/usadas en proyectos grandes y con equipos posiblemente dispersos
La arquitectura se va definiendo y mejorando a lo largo del proyecto	Se promueve que la arquitectura se defina tempranamente en el proyecto

Énfasis en los aspectos humanos: el individuo y el trabajo en equipo	Énfasis en la definición del proceso: roles, actividades y artefactos
Basadas en heurísticas provenientes de prácticas de producción de código	Basadas en normas provenientes de estándares seguidos por el entorno de desarrollo
Se esperan cambios durante el proyecto	Se espera que no ocurran cambios de gran impacto durante el proyecto