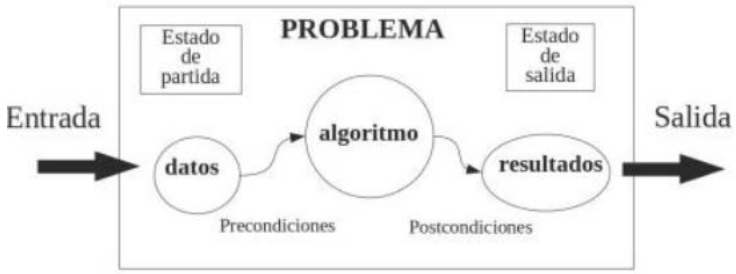
U3. Desarrollo de software

Software como herramienta de trabajo, búsqueda de información u ocio. Tipos: Instalación y uso local o funcionamiento bajo un servidor. En este aspecto, surgen conceptos de "**cliente ligero" vs "cliente pesado**" en función de donde se encuentra la mayor carga de recursos o computo en el cliente o en el servidor.

En el desarrollo de una aplicación, veremos que todo software tiene un ciclo de vida, con unos pasos. Este **ciclo de desarrollo** depende directamente de la metodología y es una **estrategia de desarrollo** que debemos seguir.

3.1. PROGRAMA INFORMÁTICO.

Programa: Conjunto de órdenes que se ejecutan en el ordenador para conseguir un objetivo. Operaciones que combina: Aritmética, Lógica y De almacenamiento.



Un programa combina:

- **Datos**: Representación simbólica, bien sea mediante números o letras, de una información.
- **Algoritmo**: Un algoritmo es un conjunto ordenado y finito de operaciones o reglas que permite hallar la solución de un problema.
  Expresiones:
  - Lenguaje natural
  - DIAGRAMA DE FLUJO.
  - PSEUDOCÓDIGO.
  - LENGUAJE DE PROGRAMACIÓN

Un programa combinará uno o más algoritmos con el fin de obtener un resultado dado unos datos. Normalmente, estos programas estarán codificados mediante un lenguaje de programación (Notación para escribir un programa): Gramática, Léxico, sintaxis y semántica propia.

3.2. Ejecución de programas

3.2.1. El hardware y el software.

HARDWARE: # HERRAMIENTA SIMULADOR DE HARDWARE.

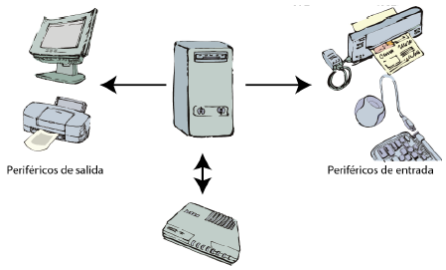
Ordenador – elementos físicos que mediante impulsos eléctricos realiza operaciones. Los elementos con que se construyen están interconectados entre sí y tienen una función determinada.

- El microprocesador es el encargado de realizar las operaciones.
- La RAM es la memoria primaria donde guardar los datos que se están procesando.
- El disco duro es el lugar donde almacenar datos de forma permanente, etc.

Estructura funcional de un ordenador.

Un ordenador realiza las operaciones con datos internos o externos, para ello integra componentes de entrada y salida (E/S):

- **PERIFÉRICOS DE ENTRADA**: Teclado, ratón, escáner…
- **PERIFÉRICOS DE SALIDA**: Mostrar la información mediante pantalla, impresora…
    - **PERIFÉRICOS MIXTOS:** Recibir/Enviar información -> tarjetas de red.



SOFTWARE:

INSTRUCCIONES Y DATOS.

Un ordenador trabaja con un sistema de numeración en **base 2 (BINARIO**–Niv. tensión).

Por eso, los **microprocesadores facilitan una serie de instrucciones** para realizar operaciones matemáticas, comparaciones o saltos de una forma más sencilla para el programador: **lenguaje ensamblador**.

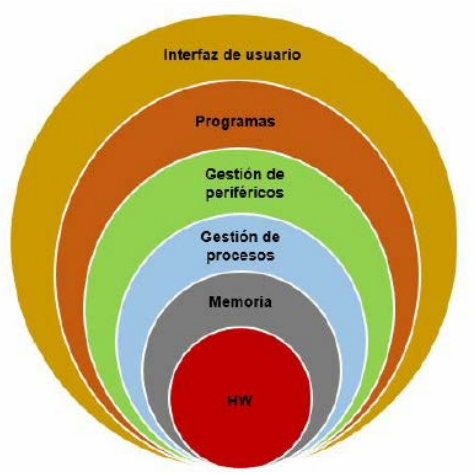
```
STACK SEGMENT STACK ; Segmento de pila
DW 64 DUP (?) ; Define espacio en la pila
STACK ENDS ; Segmento de datos
SALUDO DB "!Hola mundo!",13,10,"$" ; Cadena
DATA ENDS
CODE SEGMENT ; Segmento de Código
ASSUME CS:CODE, DS:DATA, SS:STACK
INICIO: ; Punto de entrada al programa
MOV AX,DATA ; Pone dirección en AX
MOV DS,AX ;Pone la dirección en los registros
MOV DX,OFFSET SALUDO ; Obtiene dirección del mensaje
MOV AH,09H ; Función: Visualizar cadena
INT 21H ; Servicio: Funciones alto nivel DOS
MOV AH,4CH ; Función: Terminar
    INT 21H
CODE ENDS
    END INICIO ; Marca fin y define INICIO
```

IMPLEMENTACIÓN Y CODIFICACIÓN.

Para la realización de un programa, lo primero es codificar algoritmos y estructuras de datos en **un lenguaje de programación.** Veremos lenguajes de bajo y alto nivel.

TIPOS DE SOFTWARE.

El ordenador, para realizar las tareas necesita programas que trabajen conjuntamente con objetivos comunes. Distinguimos 3 grandes grupos:

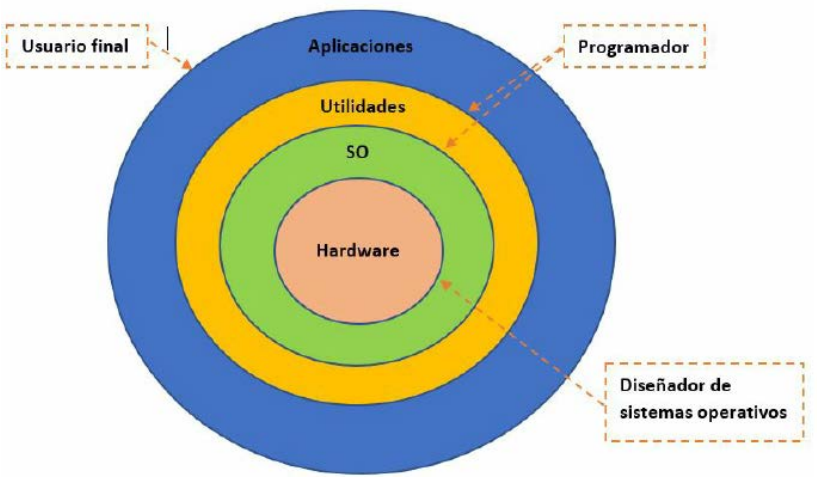


1. Sistema operativo.

Comunicación directa con el hardware. Facilita la funcionalidad de todo el hardware y periféricos. Intermediario entre aplicaciones – hardware – usuario.

2. Aplicaciones.

Programas que tienen funciones específicas que se instalan en un ordenador. Tienen que estar programados para un sistema operativo determinado, pues este será el encargado de atender las diferentes peticiones que realice (procesos).



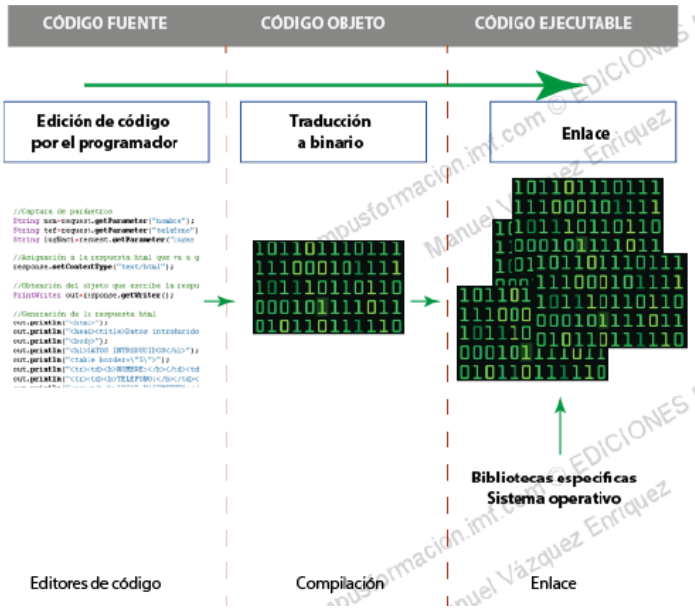
3. Drivers.

Programa cuya función es indicar al sistema operativo cómo debe operar con periféricos que desconoce.

3.2.2. CÓDIGO FUENTE, CÓDIGO OBJETO Y CÓDIGO EJECUTABLE.

3.2.2.1. CÓDIGO FUENTE:

Conjunto de instrucciones escritas en un lenguaje de programación (JAVA, C, Python).



PROS:

- Fácil de leer y entender + comentarios.
- Define procesos de forma simple.
- Se puede mantener y corregir errores/bugs – Mantenimiento.
- Funcional.

CONTRAS:

- No se ejecutan directamente sobre el ordenador. Requiere un paso intermedio.

3.2.2.2. CÓDIGO OBJETO.

Es el código resultante de compilar el código fuente. El resultado será código máquina o bytecode.

3.2.2.3. CÓDIGO EJECUTABLE.

Resultado de enlazar código con las librerías. Tras pasar por los anteriores pasos, obtenemos los archivos ejecutables para que nuestro sistema reconozca la aplicación. Reúne diferentes códigos u objetos generados por el programador junto con las "librerías de uso general" del lenguaje de programación o S.O [ES ESPECÍFICO PARA UN S.O CONCRETO]

>> FASES DE COMPILACIÓN:

- Análisis lexicográfico y sintáctico-semántico.
- Generación de código intermedio tras el análisis.
- Optimización de código. Objetivo código más fácil y rápido de interpretar.
- Generación de código.
- Enlazador de librerías.

3.3. LENGUAJES DE PROGRAMACIÓN.

CLASIFICACIÓN POR NIVEL:

Se distinguen dos tipos:

1. LENGUAJES DE BAJO NIVEL. Cuenta con una sintaxis interpretable por la máquina: LENGUAJE MÁQUINA & LENGUAJE ENSAMBLADOR.
2. LENGUAJES DE NIVEL INTERMEDIO. (nivel no aceptado por todos los manuales). Hace referencia a los lenguajes que pueden acceder a los registros del sistema, trabajar con direcciones de memoria.
3. LENGUAJES DE ALTO NIVEL. Lenguajes más modernos buscan una sintaxis que el humano pueda escribir e interpretar. (Java, C+, Python)

CLASIFICACIÓN POR GENERACIÓN:

1. Primera generación. Inicios de la informática: lenguaje máquina.
2. Segunda generación. Para facilitar la creación de programas los procesadores definen una serie de instrucciones: lenguaje ensamblador.
3. Tercera generación. Primeros lenguajes de alto nivel.
4. Cuarta generación. Mediante herramientas CASE e IDEs: autogeneración de código.
5. Quinta generación. Lenguajes de programación orientado a la IA.

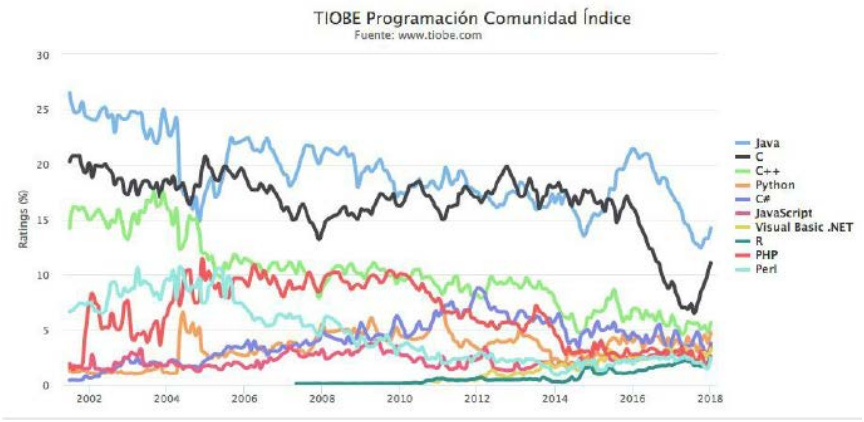
CLASIFICACIÓN POR FORMA DE EJECUCIÓN:

1. Lenguajes compilados. Un compilador convierte el código fuente en código binario y otro programa enlazador unirá este código fuente con el código de las librerías necesarias por el S.O.
2. Lenguajes interpretados. Ejecutan las instrucciones directamente en tiempo de ejecución sin que se genere código objeto. Es necesario un programa interprete en el S.O. para que interprete cada una de las líneas de código. Ejecución más lenta.
3. Lenguajes virtuales. Similar al lenguaje compilado, pero no genera un código objeto sino un “bytecode” que es interpretado por cualquier máquina que tenga la máquina virtual que se encargará de interpretar y ejecutar el código bytecode. Ejecución lenta al igual que los lenguajes interpretados, pero ofrece una mayor portabilidad del programa.

CLASIFICACIÓN POR LA TÉCNICA O PARADIGMA UTILIZADO:

1. SECUENCIAL. Programa ejecuta su código de forma secuencial.
2. ESTRUCTURADO. Uso de estructuras de control: Selectivas y/o repetitivas.
3. MODULAR. Construcción de varios módulos que interactúan entre sí.
4. PROGRAMACIÓN ORIENTADO A OBJETOS (POO). Definición de un conjunto de datos como un objeto con sus atributo y métodos: Reutilización de código.
5. PROGRAMACIÓN IMPERATIVA. Se escriben sentencias que modifican el estado del programa. Dentro de esta está la estructurada, modular y POO.
6. PROGRAMACIÓN DECLARATIVA. Solución mediante procesos internos del lenguaje sin especificar el proceso. (SQL, Lisp).

LENGUAJES DE PROGRAMACIÓN ACTUALES: https://tiobe.com/tiobe-index/



3.4. Ingeniería del software

La ingeniería del software comprende todos los aspectos de la creación de este, desde sus procesos iniciales (definición) hasta el mantenimiento de este una vez se está utilizando.

3.4.1. Conceptos previos

¿Qué es el software?

Programas de ordenador y documentación asociada. Los productos des oftware se pueden desarrollar para algún cliente en particular o para un mercado general.

¿Qué es un proceso de software?

Un conjunto de actividades cuya meta es el desarrollo o evolución del software.

¿Cuál es el coste de un desarrollo de software?

A grandes rasgos, el 60 % de los costos son de desarrollo, el 40 % restante son de pruebas. En el caso del software personalizado, los costos de evolución a menudo exceden los de desarrollo.

¿Qué son los métodos de desarrollo?

Enfoques estructurados para el desarrollo de software que incluyen modelos de sistemas, notaciones, reglas, sugerencias de diseño y guías de procesos.

¿Qué es el CASE? (Computer Aided Software Engineering)

Herramientas que intentan proporcionar ayuda automatizada a las actividades del proceso del software

Según su funcionalidad podemos encontrar:

- Herramientas de generación semiautomática de código.
- Editores UML.
- Herramientas de refactorización de código.
- Herramientas de mantenimiento como los sistemas de control de versiones.

3.4.2. CICLO DE VIDA DEL SOFTWARE. ("ISO/IEC Standard 12207:2008:software life-Cycleprocesses)

Cada etapa del desarrollo lleva asociadas una serie de actividades que debemos realizar; en cada una de ellas, dispondremos de unas herramientas para ayudarnos y tendremos que generar una serie de documentos que servirán como guía a la siguiente fase de desarrollo.

Según la ISO las actividades que debemos llevar a cabo durante el ciclo de vida de un software son:

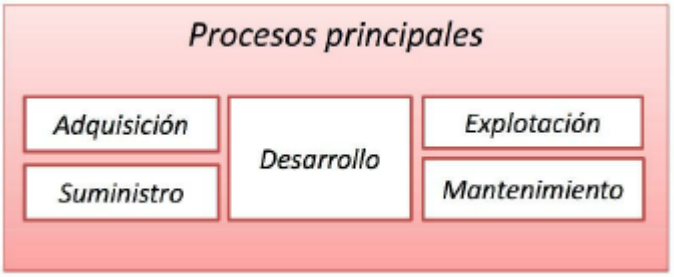
A. PROCESOS PRINCIPALES:

A.1. ADQUISICIÓN. Preparar oferta.

A.2. SUMINISTRO. Contratos e identificación recursos.

A.3. DESARROLLO. Creación del software. Se subdivide en:

- ANALISIS.
- DISEÑO
- CODIFICACIÓN.
- PRUEBAS
- INTEGRACIÓN
- IMPLANTACIÓN.



A.4. EXPLOTACIÓN. Aplicación en uso (Soporte).

A.5. MANTENIMIENTO. Corrección de bugs, aplicar mejoras, adaptación del software.

B. PROCESOS DE SOPORTE:

B.1. DOCUMENTACIÓN. Registrar toda la información generada.

B.2. GESTIÓN DE CONFIGURACIÓN. Control de versiones.

B.3. ASEGURAMIENTO DE LA CALIDAD. Protocolo para cumplir SLA.

B.4. VERIFICACIÓN. Determinar el correcto funcionamiento.

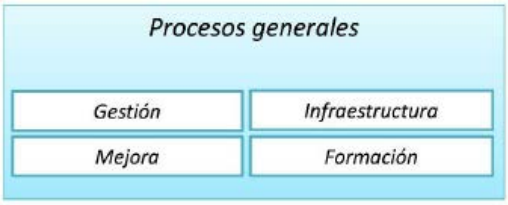
B.5. VALIDACIÓN. Comprobar si cumple con SLA.

B.6. REVISIÓN CONJUNTA. Reunión de grupos de trabajo.

B.7. AUDITORÍA. Comprobar si los procesos y objetivos se aplican, cumplen y son óptimos.

B.8. RESOLUCIÓN DE PROBLEMAS. Solventar anomalías respecto SLA pactados fruto de la ejecución del producto.

C. PROCESOS GENERALES.



C.1. GESTIÓN. Planificación, seguimiento, revisiones…

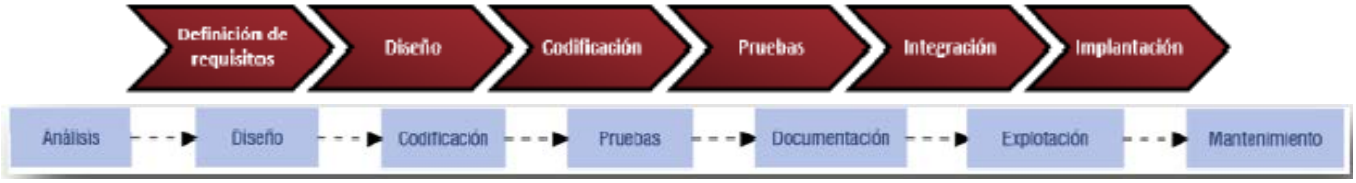
C.2. INFRAESTUCTURA. Recursos para la puesta en marcha del producto: hardware, software, consumo…

C.3. MEJORA. Evaluar y mejorar procesos.

C.4. FORMACIÓN. Formar a los agentes implicados en el cualquier proceso del ciclo de vida del producto.

3.4.3. FASES DE DESARROLLO.

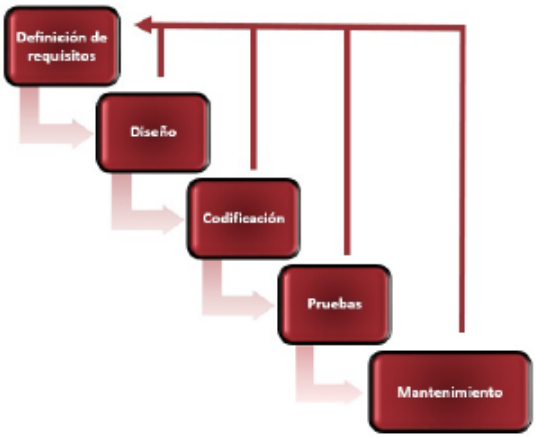
Las tareas o acciones anteriores forman parte de todo el ciclo de vida de un software. Pero un punto clave en su ejecución es "establecer el orden en que se realizan estas tareas". Disponemos de multitud de modelos o estrategias de desarrollo.



3.4.4. MODELOS DE DESARROLLO:

Un modelo de desarrollo es la representación abstracta de los diferentes procesos que tenemos que realizar durante el ciclo de vida del software. Cada modelo busca el mejor modo de alcanzar los objetivos. **No existe un modelo ideal, pues cada desarrollo y/o entorno de trabajo puede adaptarse mejor a uno u otro.**

1. MODELO CASCADA.



Este modelo está dividido en etapas que se realizan secuencialmente: Una etapa no puede empezar hasta que no haya terminado la anterior. En cada una de las fases, generaremos toda la documentación necesaria para explicar lo realizado y las bases para el siguiente paso.

Es un modelo dirigido por documentos.

PROS:

- Útil si se tiene claro los requisitos, fases desde el principio.
- Útil en equipos de trabajo pequeños o inexpertos. Estructura definida y apoyada en mucha documentación – útil para migraciones.

CONTRAS:

- Necesario definir desde inicio todos los requisitos: No realista, imposible. (Pueden surgir necesidades durante el desarrollo que no se preveían).
- El cliente no ve el avance o producto terminado hasta finalizar el proceso.
- Poca flexibilidad ante cambios o imprevistos.

2. MODELO ITERATIVO.

Es un **modelo evolutivo que se basa en prototipos** que se pueden probar. Esto ayuda a perfeccionar los requisitos del sistema.

Está compuesto por iteraciones y, en cada una de ellas, se revisan y mejoran las diferentes partes del producto hasta llegar a la solución final.

PROS:

- Se puede ir viendo los resultados durante el proceso
- Útil cuando el cliente no sabe lo que quiere y los requisitos no están bien definidos desde el principio.
- Reduce el riesgo de no cumplir las necesidades de los usuarios.
- Útil cuando los requisitos cambian durante el proceso.

CONTRAS:

- El diseño del prototipo hace que los desarrolladores utilicen herramientas que faciliten la rápida generación de código, dejando a un lado la eficiencia, fiabilidad y otros aspectos de calidad.
- Probablemente no se tendrá un código óptimo o completamente documentado.
- Posibilidad de desviarse de los objetivos o requisitos previos inicialmente y con ello es necesario una alta adaptabilidad.
- Exige disponer de las herramientas específicas.

3. MODELO INCREMENTAL.

Es un modelo evolutivo que permite a los ingenieros desarrollar versiones cada vez más completas del producto.

PROS:

- Los clientes se involucran más en todo el proceso.
- Se puede ir revisando el producto en toda la fase de desarrollo.
- Fácil introducción de cambios.

CONTRAS:

- Los incrementos deben ser pequeños.
- Se necesita un grado muy alto de planificación.
- Difícil de documentar.

>> METOLOGÍAS ÁGILES.

Podemos definir las metodologías ágiles como un conjunto tareas y procedimientos dirigidos a la gestión de proyectos. Son aquellos métodos de desarrollo en los cuales tanto las necesidades como las soluciones a estas evolucionan con el pasar del tiempo, a través del trabajo en equipo de grupos multidisciplinarios que se caracterizan por tener las siguientes cualidades:

- Desarrollo evolutivo y flexible.
- Autonomía de los equipos.
- Planificación.
- Comunicación.

Existen diferentes opciones ágiles entre las cuales podemos destacar las siguientes: Scrum, programación extrema (XP) y Kanban. Todas las metodologías ágiles cumplen con el Manifiesto ágil. Valores:

- Las personas y cómo se relacionan entre sí son más importantes que los procesos y las herramientas tecnológicas (es orientado a las personas, por definición)
- Un producto funcionando es más importante que la documentación exhaustiva (orientado al producto)
- La colaboración del cliente es más importante que la negociación del contrato (que discrimina al cliente como parte externa del producto)
- Responder al cambio es más importante que seguir el plan (adaptativo/iterativo)

>> ROLES QUE INTERACTÚAN EN EL DESARROLLO.

Analistas de sistemas.

Realiza el estudio del sistema para dirigir el proyecto en una dirección que garantice las expectativas del cliente determinando el comportamiento del software.

ETAPAS PRINCIPALES: ANÁLISIS.

Diseñador de software.

En función del análisis de un software el diseño de la solución que hay que desarrollar.

ETAPAS PRINCIPALES: DISEÑO.

Analista programador. “Desarrollador”

Aporta una visión más general del proyecto más detallada para la codificación y participa activamente en ella.

ETAPAS PRINCIPALES: DISEÑO Y CODIFICACIÓN.

Programador.

Se encarga de manera exclusiva en crear el restulado del estudio realizado por el analista y diseñador. Escribie el código fuente del software.

ETAPAS PRINCIPALES: CODIFICACIÓN.

Arquitecto de software.

Es la argamasa que cohesiona el proceso de desarrollo. Conoce e investiga los frameworks y tecnologías, revisando que todo el procedimiento se lleva a cabo de la mejor forma y con los recursos necesarios.

ETAPAS PRINCIPALES: ANALISIS, DISEÑO, INTEGRACIÓN E IMPLEMENTACIÓN.

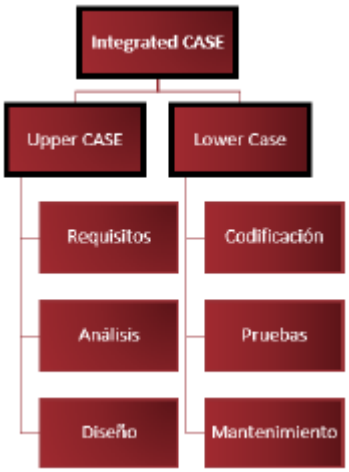
3.4.5. Herrramientas CASE.

Las herramientas **CASE (Computer Aided Software Engineering)** son aplicaciones que nos ayudan a aumentar la productividad en el desarrollo del software, reduciendo los tiempos de creación y el coste económico.

CLASIFICACIÓN. Según la fase del ciclo de vida:

A. **Herramientas de alto nivel: Upper-CASE o front-end**.
   Automatización y soporte de actividades de las primeras fases del desarrollo: definición, análisis y diseño.

B. **Herramientas de bajo nivel: Lower-CASE o back-end.**
   Dirigidas a las últimas fases del desarrollo: implementación, pruebas y mantenimiento.



Feature Support	Visio	Rational Rose	Visible Analyst	Argo UML	Meta Edit+	mart draw	Visual Case	DB-Main	Omni graffle	ADO it
UPPER CASE										
Diagramas de flujo de datos	X	X	X	X	x	X	X		X	X
Meta Edit +					X					
Diagramas de flujo de trabajo	x	X	X	X	X	X	X		X	X
Diagramas orientados a objetos	X	X	X	X	X	X	X		X	X
Diccionario de datos	X	X	X	X	X		X	X		X
Adopción de reglas de negocio	X	X	X	X	X		X			X
Herramientas de análisis		X	X				X			X
LOWER CASE										
Generación de código	X	X	X	X	X		X	X		X
Generación de formularios	X	X	X		X					X
Generación de informes	X	X	X		X		X			X
Generación de documentación	X	X	X	X	X			X		X
Utilidades de importación y exportación	X	X	X	X	X			x		x
Codificación del programa		X	X		X		X			
Testing		X	x		x		x			

- Rational Rose
- Visible Analyst
- ArgoUML
- MetaEdit+
- Smart Draw
- Visual Case
- DB-Main
- Omni graffle
- ADOit
- Visio

3.4.6. Errores en el desarrollo.

Diversos errores como una mala planificación, una expectativa muy alta, dotar de recursos insuficientes al proyecto… pueden hacer que el producto no llegue a buen término.

Motivación indeterminada: "La motivación es el mayor factor sobre la productividad y la calidad del desarrollo de software" (Boehm).
Personal poco capacitado: después de la motivación, las capacidades de los individuos o su relación como equipo son los siguientes factores en importancia en la productividad.
Falta de actuación con los elementos problemáticos.
Heroísmo: no interesan actuaciones heroicas, sino el desarrollo continuo, pausado y sin sobresaltos.
Añadir más personal a los proyectos retrasados.
Oficinas repletas y ruidosas. Los trabajadores que están en oficinas silenciosas tienden a funcionar mejor que los que ocupan cubículos en salas ruidosas y repletas.
Fricciones entre los clientes y los desarrolladores.
Expectativas poco realistas: según el *standish group*, las expectativas realistas son uno de los cinco factores necesarios para asegurar el éxito de los desarrollos de software internos.
Falta de un promotor efectivo del proyecto: ayuda a realizar una planificación realista, control de cambios y la introducción de nuevos métodos de desarrollo.
Falta de participación de los implicados: promotores, ejecutivos, responsables del equipo, miembros del equipo, personal de ventas, usuarios finales, clientes, marketing...
Falta de participación del usuario: según el *standish group*, la razón fundamental de que los proyectos de sistemas de información tengan éxito es la implicación del usuario.
Política sobre el contenido: Los grupos de desarrolladores tienen diferente personalidad igual que las personas. Podemos encontrar políticos (se concentran en las relaciones con los responsables), investigadores (su principal preocupación es investigar, buscar y recopilar más información), aislacionistas (buscan marcar su zona y aislarse del exterior)... Los equipos en los que predominan los políticos funcionan bien al principio pero luego pierden el interés de la dirección.
Buenos deseos: no es optimismo, es cerrar los ojos y esperar que algo funcione, sin tener bases razonables para pensarlo.

Referente al proceso de desarrollo

Planificaciones excesivamente optimistas: además reduce en gran medida la autoestima y la productividad de los desarrolladores.
Nula o poca gestión de riesgos.
Fallos de los subcontratistas (retrasos, baja calidad, no respuesta a los requisitos).
Planificación insuficiente.
Abandono de la planificación por un exceso de presión: se hacen planes pero se abandonan al aparecer las dificultades.
Perder el tiempo en los trámites iniciales: tiempo empleado en obtener la aprobación de la dirección y los fondos. Es más fácil no perder ese tiempo que recuperarlo al programar.
Saltar a codificar: eliminar las actividades iniciales "improductivas". Ante la pregunta: ¿Enséñame la documentación de diseño?; la respuesta: "No ha habido tiempo de diseñar. Eliminamos el diseño, el análisis y los requisitos".
Diseño inadecuado: no se tiene el tiempo ni la tranquilidad suficiente para llevarlo a cabo.
Eliminar las actividades de control de calidad: un ahorro de un día en calidad puede costar entre 3 y 10 días al final del proyecto.
Poco seguimiento del proyecto.
Forzar la convergencia de todas las actividades (módulos) del producto: si intentamos que todo encaje antes de tiempo, tendremos trabajo extra.
Omitir tareas importantes al hacer la estimación.
Mantener una planificación cuando ya se han producido retrasos: hay que actualizar la planificación cuando vamos teniendo más datos del producto (cambios en los requisitos).
Programación a destajo (*code-like-hell*): piensan que si los programadores están suficientemente motivados, salvarán cualquier obstáculo.

Referente al producto

Exceso de requisitos: se pide características complejas al sistema, que el usuario no necesita. Además, se da excesiva importancia al rendimiento.
Cambio en los requisitos: el proyecto medio experimenta un 25 % de cambio en los requisitos durante su desarrollo, lo que origina un 25 % de incremento en el tiempo de desarrollo.
Exceso de funcionalidades: los desarrolladores, fascinados por la tecnología u otros productos, añaden funcionalidades no solicitadas por el cliente.
Tiras y aflojas en la negociación: la dirección asume el retraso en el proyecto pero, a la vez, añade nuevas funcionalidades.
Desarrollo orientado a la investigación: Seymour Cray, diseñador de los ordenadores Cray, dice que nunca trata de superar los límites de la ingeniería en más de dos áreas simultáneamente, porque el riesgo de fallar es demasiado alto. La investigación no es predecible.

Referente a la tecnología

Síndrome de la panacea: se confía demasiado en las nuevas tecnologías antes de ser probadas.
Exceso en la estimación de los ahorros de los nuevos métodos y herramientas. Raramente se producen grandes mejoras instantáneas en productividad. Los beneficios de las nuevas prácticas se ven reducidas por las curvas de aprendizaje asociadas con ellas.
Cambio de herramientas a mitad del proyecto.
Falta de control automático de código fuente.

3.4.7. Reutilización de código.

En muchas ocasiones, los programas que realizamos proceden de un código que ya existe, o partimos de otro proyecto donde ya disponemos de algo de código que

tenemos que adaptar. Esto puede suponer un considerable ahorro en el tiempo de desarrollo. Es esencial entender el código para reutilizarlo.

Y al crear un software, no solo tenemos que tener en cuenta el funcionamiento del mismo, también debemos hacerlo entendible por otros desarrolladores, realizando comentarios y estructurándolo de forma lógica y, a poder ser, lo más simple posible

- DOCUMENTACIÓN.
- REUTILIZACIÓN.
- MODIFICACIÓN.
- MANTENIMIENTO.