C.F.G.S. DESARROLLO DE APLICACIONES WEB

MÓDULO: Entornos de Desarrollo

Unidad 1

Desarrollo de Software

Contenido

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.
Por tanto el software es el equipamiento lógico necesario para que el ordenador realice tareas específicas.

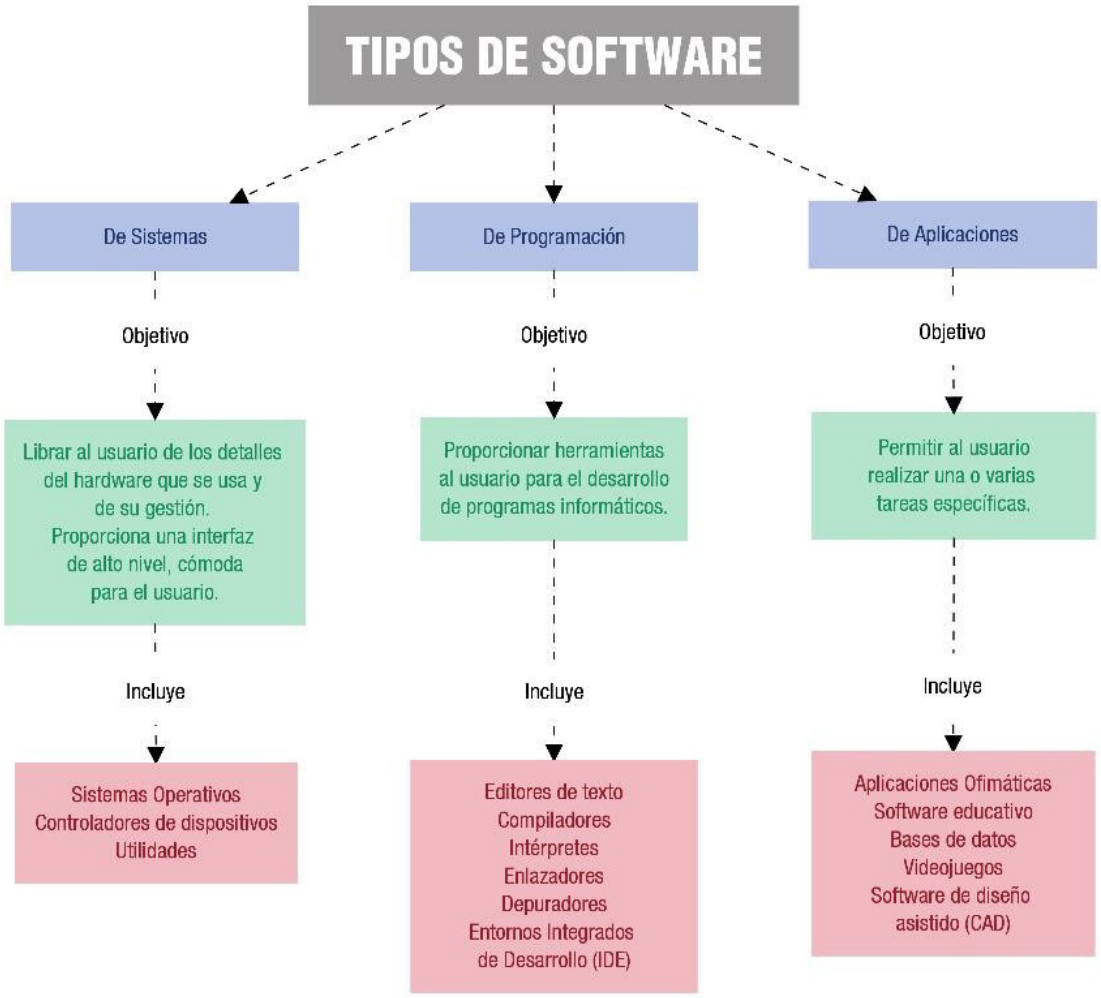
Según su función se distinguen **tres tipos de software**: sistema operativo, software de programación y aplicaciones.

- **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.
  También facilita la interacción entre los componentes físicos y el resto de las aplicaciones, y proporcionando una interfaz con el usuario
  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.
  A su vez, un programa es un conjunto de instrucciones escritas en un lenguaje de programación, que una vez ejecutadas realizan una o varias tareas en un ordenador.

- **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.

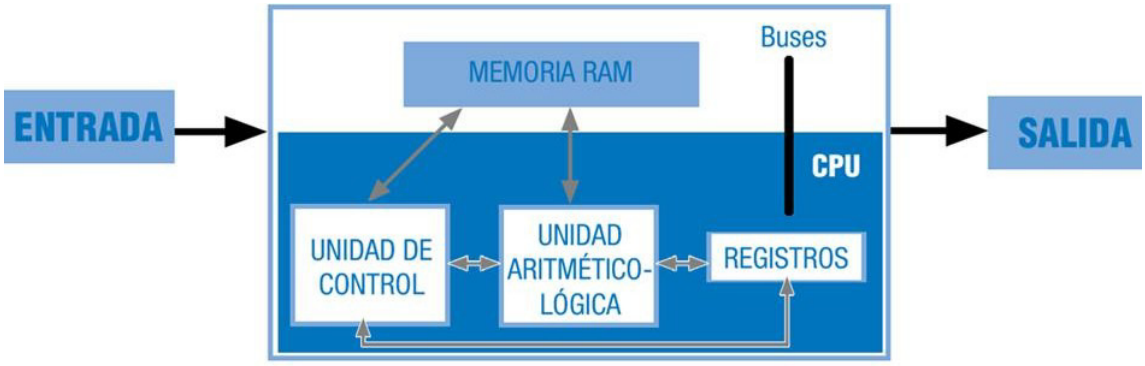
En definitiva, distinguimos los siguientes tipos de software:



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:



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

- **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.

- **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. 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 interpretar 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.

3.- Licencias de Software. Software libre y propietario

Una **licencia de software** es un contrato que se establece entre el desarrollador de un software sometido a propiedad intelectual y a derechos de autor, y el usuario. Es el desarrollador, el que elige la licencia según la cual distribuye el software.

Una **Patente** es un conjunto de derechos exclusivos garantizados por un gobierno o autoridad al inventor de un nuevo producto (material o inmaterial) susceptible de ser explotado industrialmente para el bien del solicitante por un periodo de tiempo limitado.

Derecho de autor o copyright es la forma de protección proporcionada por las leyes vigentes en la mayoría de los países para los autores de obras originales incluyendo obras literarias, dramáticas, musicales, artísticas e intelectuales, tanto publicadas como pendientes de publicar

Software de dominio público: aquél que no está protegido con copyright es decir carece de licencia o no hay forma de determinarla pues se desconoce al autor.

Software libre: es un software cuyo código fuente puede ser estudiado, modificado, y utilizado libremente con cualquier finalidad y redistribuido con cambios o mejoras sobre ellas.1 Su definición está asociada al nacimiento del movimiento de software libre, encabezado por el activista y experto informático estadounidense Richard Stallman. Un software es libre si otorga a los usuarios de manera adecuada las denominadas cuatro libertades: libertad de usar, estudiar, distribuir y mejorar, de lo contrario no se trata de software libre.

Software con Copyleft: ***software libre*** cuyos términos de distribución no permiten a los redistribuidores agregar ninguna restricción adicional cuando lo redistribuyen o modifican, o sea, la versión modificada debe ser también libre Con esto se intenta evitar que el Software Libre se vea modificado y luego redistribuido por empresas privadas como Software Privativo. Sin embargo, **Copyleft** no posee reconocimiento legal al día de hoy, entorpeciendo el crecimiento natural del Software Libre.

Software semi libre: aquél que no es libre, pero viene con autorización de usar, copiar, distribuir y modificar para particulares sin fines de lucro

Freeware: se usa comúnmente para programas que permiten la redistribución pero no la modificación (y su código fuente no está disponible) Debemos tener en cuenta que el **Software Libre** no tiene por qué ser gratuito, del mismo modo en que el **Freeware** no tiene por qué ser libre.

Shareware: software con autorización de redistribuir copias, pero debe pagarse cargo por Licencia de uso continuado.
Software propietario: aquél cuyo uso, redistribución o modificación están prohibidos o necesitan una autorización.
Software comercial: el desarrollado por una empresa que pretende ganar dinero por su uso.

Adware: Subprograma que descarga publicidad sobre otro programa principal. Esto ocurre cuando un programa tiene versiones comerciales o más avanzadas que necesitan ser compradas para poder ser utilizadas. Pagando por la versión comercial, esos anuncios desaparecen.

Trial: Versión de programa pago, distribuido gratuitamente con todos los recursos activos, pero por un tiempo determinado.

4.- Ciclo de vida del Software.

Entendemos por **Desarrollo de Software** todo el proceso que ocurre desde que se concibe una idea hasta que un programa está implementado 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 (aquellos que van a utilizar el programa).

Podemos definir el **Ciclo de Vida del Software** como 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 reemplazado por otro más adecuado.

Estas fases son las siguientes:

Análisis: Construye un modelo de los requisitos. En esta etapa se debe entender y comprender de forma detallada el problema que se va a resolver. Es muy importante producir en esta etapa una documentación entendible, completa y fácil de verificar y modificar.

Diseño: En esta etapa ya sabemos qué es lo que hay que hacer, ahora hay que definir cómo se va a resolver el problema. Se deducen las estructuras de datos, la arquitectura de software, la interfaz de usuario y los procedimientos. Por ejemplo, en esta etapa hay que seleccionar el lenguaje de programación, el Sistema Gestor de Base de Datos, etc.

Codificación: En esta etapa se traduce lo descrito en el diseño a una forma legible por la máquina. La salida de esta fase es código ejecutable.

Pruebas: Se comprueba que se cumplen los criterios de corrección y calidad. Las pruebas deben garantizar el correcto funcionamiento del sistema.

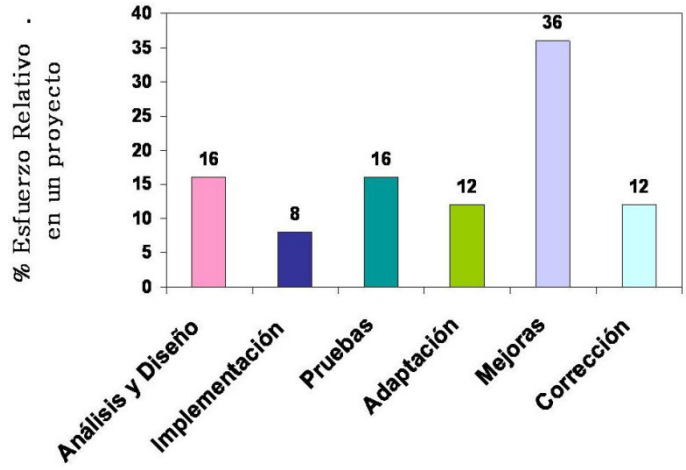
Mantenimiento: Esta fase tiene lugar después de la entrega del software al cliente. En ella hay que asegurar que el sistema pueda adaptarse a los cambios. Se producen cambios porque se han encontrado errores, es necesario adaptarse al entorno (por ejemplo se ha cambiado de sistema operativo) o porque el cliente requiera mejoras funcionales.

Cada etapa tiene como entrada uno o varios documentos proce3dentes de las etapas anteriores y produce otros documentos de salida, por ello una tarea importante a realizar en cada etapa es la **documentación**.

Análisis y diseño.	Estudio del problema y planteamiento de soluciones.
Implementación.	Confección de la solución elegida
Pruebas.	Proceso para comprobar la calidad del producto
Adaptación.	Instalación al cliente para que pueda usarlo.
Mejoras.	Retoques que permiten hacer más atractivo el uso.
Correcciones.	Solución de errores o ajustes para evitar problemas.

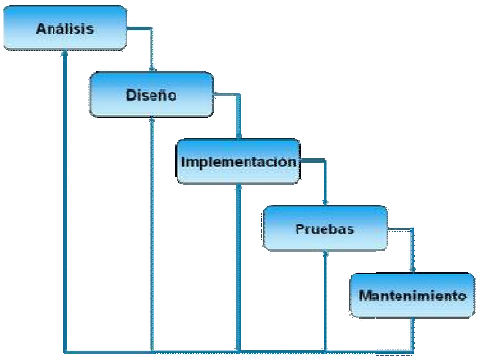
4.1.- Modelos de Ciclo de vida.

Durante el ciclo de vida del software se realiza un **reparto del esfuerzo** de desarrollo del mismo en cada una de las fases que lo componen. La tabla siguiente muestra cuáles son esas fases, y el gráfico que le sigue muestra el porcentaje de esfuerzo y por tanto de coste que supone cada fase sobre el total de un proyecto.



Diversos autores han planteado distintos modelos de ciclos de vida, pero los más conocidos y utilizados son: *Cascada, Incremental y en Espiral*.

- **Modelo en Cascada**.



Es el modelo de vida clásico del software. Requiere finalizar una etapa para realizar la siguiente. Es prácticamente imposible que se pueda utilizar, ya que requiere conocer de antemano todos los requisitos del sistema. Sólo es aplicable a pequeños desarrollos, ya que las etapas pasan de una a otra sin retorno posible. (Se presupone que no habrá errores ni variaciones del software).

- **Modelo en Cascada con Realimentación.**

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

Es el modelo perfecto si el proyecto es rígido (pocos cambios, poco evolutivo) y los requisitos están claros.

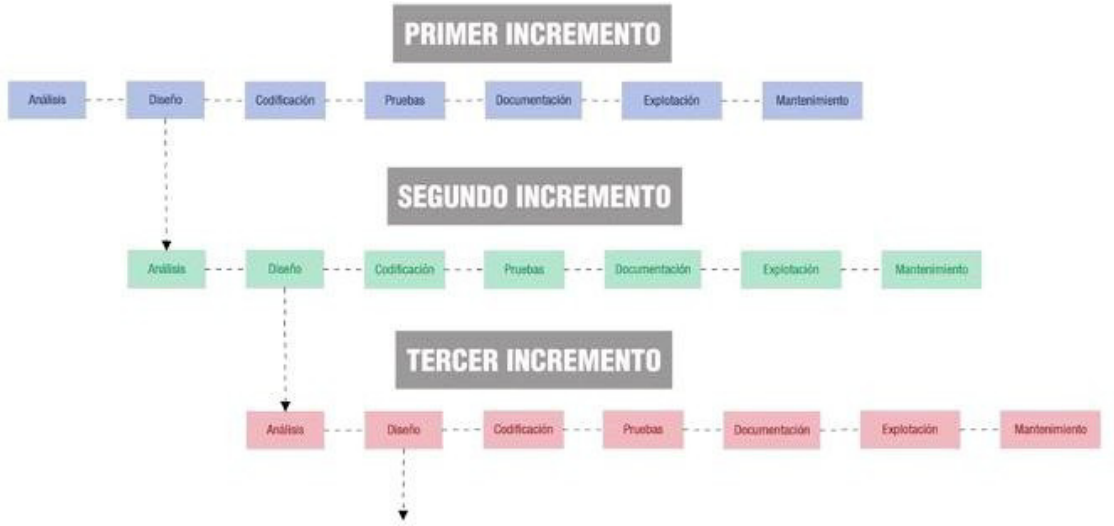
- **Modelos Evolutivos**

Son más modernos que los anteriores. Tienen en cuenta la naturaleza cambiante y evolutiva del software. Al igual que el software evoluciona también lo hacer los requisitos del usuario y el producto. Los modelos evolutivos permiten entregar versiones cada vez más completas hasta llegar al producto final deseado.

Distinguimos dos variantes:

a) Modelo Iterativo Incremental

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



No se necesitan conocer todos los requisitos al inicio y permite la entrega temprana al cliente de partes operativas del software. Sin embargo es difícil estimar el esfuerzo y conlleva el riesgo de no acabar nunca.

b) Modelo en Espiral

Es una combinación del modelo anterior con el modelo en cascada. En él, el software se va construyendo repetidamente en forma de versiones que son cada vez mejores, debido a que incrementan la funcionalidad en cada versión. Es un modelo bastante complejo.

5.- Fases en el desarrollo y ejecución del software.

Hemos visto 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.
Estas etapas son:

1. **Análisis de requisitos:** Se especifican los requisitos funcionales y no funcionales del sistema.
2. **Diseño:** Se divide el sistema en partes y se determina la función de cada una.
3. **Codificación:** Se elige un Lenguajes de Programación y se codifican los programas.
4. **Pruebas:** Se prueban los programas para detectar errores y se depuran.
5. **Documentación.**: De todas las etapas, se documenta y guarda toda la información.
6. **Explotación:** Instalamos, configuramos y probamos la aplicación en los equipos del cliente.
7. **Mantenimiento:** Se mantiene el contacto con el cliente para actualizar y modificar la aplicación el futuro.

<table>
<tr><td>ANÁLISIS
Decidir qué hacer.</td><td colspan="2">• Estudio de viabilidad.
• Deducción de requisitos.
• Análisis de requisitos.
• Modelado del sistema.
• Prototipado.
• …</td></tr>
<tr><td>DISEÑO
Decidir cómo hacerlo.</td><td>• Arquit. del sistema.
• Detallado.
• Interfaz de usuario.
• Datos.
• …</td><td rowspan="3">Durante todas estas fases:
Aceptación del producto.

VALIDACIÓN Y VERIFICACIÓN

• Inspecciones y revisiones.
• Planificación de prueba.
• Pruebas de unidad.
• Pruebas de integración.
• Pruebas de regresión.
• Pruebas del sistema.
• Pruebas de aceptación.</td></tr>
<tr><td>CONSTRUCCIÓN
• CODIFICACIÓN. Hacerlo.
• PRUEBAS. Probar el producto.
• INSTALACIÓN. Usar el producto obtenido.</td><td>• Documentación.
• Codificación.
• Debug (Depuración).
• …</td></tr>
<tr><td colspan="2">MANTENIMIENTO
Seguimiento del producto durante su funcionamiento real.</td></tr>
</table>

5.1.- Análisis.

Esta es la primera fase del proyecto es la más complicada y la que más depende de la capacidad del **analista**. 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. En esta fase es esencial una buena comunicación entre el cliente y los desarrolladores, para facilitar esta comunicación se utilizan varias **técnicas**, como las siguientes:

- **Entrevistas.** Es la técnica más tradicional que consiste en hablar con el cliente. Hay que tener sobre todo conocimientos de psicología**.**
- **Desarrollo conjunto de aplicaciones.** Se apoya en la dinámica de grupos. Participan en ella, usuarios, administradores, analistas desarrolladores, etc. Cada persona juega un rol concreto y todo está reglamentado.
- **Planificación conjunta de requisitos**. Las entrevistas están dirigidas a la alta dirección, y los productos resultantes son los requisitos de alto nivel o estratégicos.
- **Brainstorming**. Reuniones de grupos cuyo objetivo es generar ideas desde diferentes puntos de vista para la resolución de un problema. Permite explorar un problema desde múltiples puntos de vista.

En esta fase se especifican y analizan todos los requisitos del sistema:

- **Funcionales**: Servicios que el sistema debe proporcionar. 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**: Restricciones que afectaran al sistema. Tiempos de respuesta del programa, legislación aplicable, tratamiento ante la simultaneidad de peticiones, etc.

Requisitos funcionales	**Requisitos no funcionales**
El usuario puede agregar un nuevo contacto. El usuario puede eliminar uno o varios contactos de la lista. El usuario puede imprimir la lista de contactos.	La aplicación debe funcionar en sistemas operativos Linux y Windows. El tiempo de respuesta a consultas. Altas, bajas y modificaciones ha de ser inferior a 5 segundos.
Se va a operar con tarjetas de crédito	

Para representar los requisitos del sistema se utilizan diferentes técnicas:

- **Diagramas de flujo de datos DFD**. Es un diagrama que representa el flujo de los datos entre los distintos procesos, entidades externas y almacenes que forman el sistema.
    - Los procesos de identifican funciones dentro del sistema (burbujas ovaladas)
    - Las entidades externas representan componentes que no forman parte del sistema. Que proporcionan datos o los reciben. (rectángulos)
    - Los almacenes representan los datos desde el punto de vista estático, lugar donde se almacenan o desde donde se recuperan. (dos líneas horizontales paralelas)
    - El flujo de datos representa el movimiento de datos del sistema (flechas)

- **Diagramas Entidad** / **Relación DER**. Usado para representar los datos y la forma en la que se relacionan entre ellos. Está formado por entidades (rectángulo) y relaciones (rombo).
- **Diccionario de datos DD**. Es una descripción detallada de los datos utilizados por el sistema que gráficamente se encuentran representados por los flujos de datos y almacenes presentes sobre el conjunto de DFD.
- **Prototipos**. Es una versión inicial del sistema, se utiliza para clarificar algunos puntos, demostrar los conceptos y conocer mejor el problema y sus posibles soluciones
- **Casos de uso**. Es la técnica definida en UML (*Unified Modeling Language*), basada en escenarios que describen cómo se usa el software en una determinada situación.
- **Un modelo de dominio**, es diagrama que describe el modelo conceptual de todos los temas relacionados con un problema específico. En él se describen las distintas entidades, sus atributos, papeles y relaciones, además de las restricciones que rigen el dominio del problema. Se asocia más bien con modelos **orientados a objetos** y proporciona una visión estructural del sistema a desarrollar que puede ser complementado con otros puntos de vista dinámicos, como el modelo de **casos de uso**.

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.

La estructura del documento ERS propuesta por el IEEE es la última versión del estándar 830 [IEEE, 1998]

5.2.- Diseño.

Una vez identificados los requisitos es necesario componer la forma en que se solucionará el problema. En esta etapa se traducen los requisitos funcionales y no funcionales en una representación de software.

Hay dos tipos de diseño, el diseño estructurado basado en el flujo de datos a través del sistema y el diseño orientado a objetos donde el sistema se entiende como un conjunto de objetos que tienen propiedades y comportamientos además de eventos que activan operaciones que modifican el estado de los objetos.

5.2.1.- Diseño estructurado

Es el diseño clásico y representa un modelo de 4 niveles.

- **Diseño de Datos**. Se encarga de transformar los datos y las relaciones definidas en los diagramas de entidad-relación y en el diccionario de datos, en las estructuras de datos que se utilizarán para implementar el software. (Entidades y relaciones de las bases de datos).
- **Diseño arquitectónico**. Representación de la estructura de los componentes del software, sus propiedades e interacciones partiendo de los DFD. Módulos de programas, conectores entre componentes, …etc
- **Diseño de la interfaz**. El resultado de esta tarea es la creación de formatos de pantalla.
- **Diseño a nivel de componentes (procedimental)**. Diseño de cada componente de software con el suficiente nivel de detalle. Para ello se utilizarán representaciones gráficas como diagramas de flujo, diagramas de cajas, tablas de decisión, pseudocódigo, etc,

Es también responsabilidad de esta fase del diseño

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

5.2.2.- Diseño orientado a objetos.

Para llevar a cabo un diseño de software orientado a objetos (DOO) es necesario partir de un análisis orientado a objetos (AOO). En este análisis se definen todas las clases que son importantes para el problema que se trata de resolver, las operaciones y los atributos asociados, las relaciones y comportamientos (Diagramas de casos de uso, Modelo de Dominio,..). Este diseño define 4 capas de diseño:

- **Subsistema**. Se centra en el diseño de de los subsistemas que implementan las funciones principales del sistema.
- **Clases y objetos**. Especifica la arquitectura de objetos y la jerarquía de clases.
- **Mensajes**. Indica cómo se realiza la colaboración entre los objetos.
- **Responsabilidades**. Identifica las operaciones y atributos que caracterizan cada clase.

Para el análisis y diseño orientado a objetos se utiliza UML, Lenguaje de modelado basado en diagramas que sirve para expresar modelos y que se ha convertido en un estándar de las metodologías de desarrollo orientado a objetos.

5.3.- Codificación.

Durante la fase de codificación se realiza el proceso de programación. El programador recibe las especificaciones del diseño y las transforma en instrucciones escritas en un lenguaje de programación y almacenadas dentro de un programa.

Las características deseables de todo código son:

- Modularidad: que está dividido en trozos más pequeños.
- Corrección: que haga lo que se pide realmente.
- Fácil de leer: para facilitar su desarrollo y mantenimiento futuro.
- Eficiencia: que haga un buen uso de los recursos.
- Portabilidad: que se pueda implementar en cualquier equipo.

5.3.1- Fases en la obtención de 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 utilizando un editor de texto.

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 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, funciones, bibliotecas y procedimientos necesarios para codificar la aplicación.

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.

Objeto.

El código objeto es un código intermedio. Es el resultado de traducir código fuente a un código equivalente formado 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) 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 proceso de traducción de código fuente a código objeto puede realizarse de dos formas:

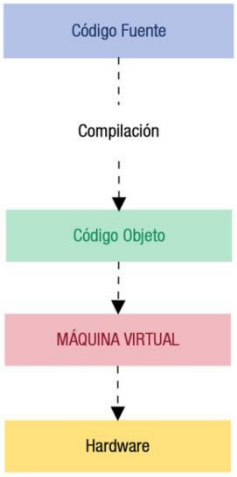
- **Compilación**: El proceso de traducción se realiza sobre todo el código fuente, en un solo paso. Se crea código objeto que habrá que enlazar. El software responsable se llama compilador.
- **Interpretación**: El proceso de traducción del código fuente se realiza línea a línea y se ejecuta simultáneamente. No existe código objeto intermedio. El software responsable se llama intérprete. 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.

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

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) y obtener así un único archivo que ya sí es ejecutable directamente por la computadora.

5.3.2.- 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 compilados como interpretados.



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 de las aplicaciones.

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.

- **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 y actúa de puente entre la aplicación y el hardware concreto del equipo donde se instale.

5.3.4.- Entornos de Ejecución.

Un entorno de ejecución es un servicio de máquina virtual que sirve como base software para la ejecución de programas. En ocasiones pertenece al propio sistema operativo, pero también se puede instalar como software independiente que funcionará por debajo de la aplicación.
Es decir, es un conjunto de utilidades que permiten la ejecución de programas.

Se denomina runtime al tiempo que tarda un programa en ejecutarse en la computadora.

Durante la ejecución, los entornos se encargarán de:

- Configurar la memoria principal disponible en el sistema.
- Enlazar los archivos del programa con las bibliotecas existentes y con los subprogramas creados. Considerando que las bibliotecas son el conjunto de subprogramas que sirven para desarrollar o comunicar componentes software pero que ya existen previamente y los subprogramas serán aquellos que hemos creado a propósito para el programa.
- Depurar los programas: comprobar la existencia (o no existencia) de errores semánticos del lenguaje (los sintácticos ya se detectaron en la compilación).

Funcionamiento del entorno de ejecución:
El Entorno de Ejecución está formado por la máquina virtual y los API's (bibliotecas de clases estándar, necesarias para que la aplicación, escrita en algún Lenguaje de Programación pueda ser

ejecutada). Estos dos componentes se suelen distribuir conjuntamente, porque necesitan ser compatibles entre sí.

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

5.3.5.- Java runtime environment.

En esta sección de va a explicar el funcionamiento, instalación, configuración y primeros pasos del Runtime Environment del lenguaje Java (se hace extensible a los demás lenguajes de programación).

- **Concepto.**

Se denomina JRE al Java Runtime Environment (entorno en tiempo de ejecución Java). El JRE se compone de un conjunto de utilidades que permitirá la ejecución de programas java sobre cualquier tipo de plataforma.

- **Componentes.**

JRE está formado por:

- Una Máquina virtual Java (JMV o JVM si consideramos las siglas en inglés), que es el programa que interpreta el código de la aplicación escrito en Java.
- Bibliotecas de clase estándar que implementan el API de Java.
- Las dos: JMV y API de Java son consistentes entre sí, por ello son distribuidas conjuntamente.

5.4.- Pruebas.

Una vez obtenido el software, la siguiente fase del ciclo de vida es la realización de pruebas. Normalmente, éstas se realizan sobre un conjunto de datos de prueba, que consisten en un conjunto seleccionado y predefinido de datos límite a los que la aplicación es sometida.

La realización de pruebas es imprescindible para asegurar la **validación** y **verificación** del software construido.

- **Pruebas de verificación.**
  Conjunto de actividades que tratan de comprobar si se está construyendo el producto correctamente, es decir, si el software implementa correctamente la función para la que está diseñado.

- **Pruebas de validación.**
  Conjunto de actividades que tratan de comprobar si el producto se ajusta a los requisitos del cliente.

El objetivo en esta etapa es planificar y diseñar pruebas que sistemáticamente detecten diferentes clases de error. Una prueba tiene éxito si descubre un error no detectado hasta entonces. Para ello se crean diferentes casos de prueba, que son documentos que especifican los valores de entrada, salida esperada y las condiciones previas para la ejecución de la prueba.
Para llevar a cabo el diseño de casos de prueba se utilizan dos técnicas:

- **Pruebas de caja blanca**. Se centran en validar la estructura interna del programa.
- **Pruebas de caja negra**. Se centran en validar los requisitos funcionales sin fijarse en el funcionamiento interno del programa.

5.5.- 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:

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.	• Descripción de la funcionalidad de la	Toda la información necesaria para:

Documentos a elaborar en el proceso de desarrollo de software			
	GUÍA TÉCNICA	**GUÍA DE USO**	**GUÍA DE INSTALACIÓN**
	• La codificación de los programas. • Las pruebas realizadas.	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.	• 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.

5.6.- 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. Aunque diversos autores consideran la explotación y el mantenimiento como la misma etapa, nosotros vamos a diferenciarlas en base al momento en que se realizan.

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. 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. En este punto se deberá proporcionar soporte al usuario cuando la solicite.

Es muy importante tenerlo todo preparado antes de presentarle el producto al cliente: será el momento crítico del proyecto.

5.7.- 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 es la 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).

El mantenimiento se define como el proceso de control, mejora y optimización del software.

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).

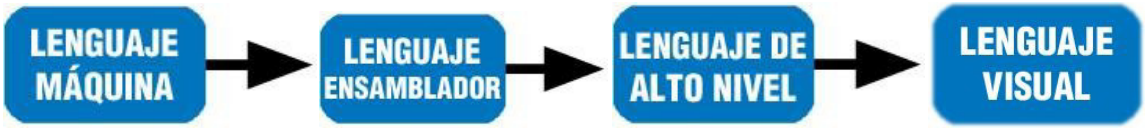
6.- Lenguajes de Programación.

Ya dijimos que los programas informáticos están escritos usando algún lenguaje de programación.

Por tanto, 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.

En otras palabras, es muy importante tener muy clara la función de los lenguajes de programación. Son los instrumentos que tenemos para que el ordenador realice las tareas que necesitamos.
Hay multitud de lenguajes de programación, cada uno con unos símbolos y unas estructuras diferentes. Además, cada lenguaje está enfocado a la programación de tareas o áreas determinadas. Por ello, la elección del lenguaje a utilizar en un proyecto es una cuestión de extrema importancia.
Los lenguajes de programación han sufrido su propia evolución, como se puede apreciar en la figura siguiente:



- **Lenguaje máquina**:
  - Sus instrucciones son combinaciones de unos y ceros.

- Es el único lenguaje que entiende directamente el ordenador. (No necesita traducción).
- Fue el primer lenguaje utilizado.
- Es único para cada procesador (no es portable de un equipo a otro).
- Hoy día nadie programa en este lenguaje.

- **Lenguaje ensamblador**:
  - Sustituyó al lenguaje máquina para facilitar la labor de programación.
  - En lugar de unos y ceros se programa usando mnemotécnicos (instrucciones complejas).
  - Necesita traducción al lenguaje máquina para poder ejecutarse.
  - Sus instrucciones son sentencias que hacen referencia a la ubicación física de los archivos en el equipo.
  - Es difícil de utilizar.

- **Lenguaje de alto nivel basados en código**:
  - Sustituyeron al lenguaje ensamblador para facilitar más la labor de programación.
  - En lugar de mnemotécnicos, se utilizan sentencias y órdenes derivadas del idioma inglés. (Necesita traducción al lenguaje máquina).
  - Son más cercanos al razonamiento humano.
  - Son utilizados hoy día, aunque la tendencia es que cada vez menos.

- **Lenguajes visuales**:
  - Están sustituyendo a los lenguajes de alto nivel basados en código.
  - En lugar de sentencias escritas, se programa gráficamente usando el ratón y diseñando directamente la apariencia del software.
  - Su correspondiente código se genera automáticamente.
  - Necesitan traducción al lenguaje máquina.
  - Son completamente portables de un equipo a otro.

6.1.- Concepto y características.

Los lenguajes de programación han evolucionado, y siguen haciéndolo, siempre hacia la mayor usabilidad de los mismos (que el mayor número posible de usuarios lo utilicen y exploten).

La elección del lenguaje de programación para codificar un programa dependerá de las características del problema a resolver.

- **Concepto.**

  Un lenguaje de programación es el conjunto de:

  - **Alfabeto**: conjunto de símbolos permitidos.

- **Sintaxis**: normas de construcción permitidas de los símbolos del lenguaje.
- **Semántica**: significado de las construcciones para hacer acciones válidas.

- **Características**

  Podemos clasificar los distintos tipos de Lenguajes de Programación en base a distintas características:

  - **Según el nivel de abstracción, o sea, según lo cerca que esté del lenguaje humano**:
    - Lenguajes de Programación de **Alto nivel**: por su esencia, están más próximos al razonamiento humano.
    - Lenguajes de Programación de **Bajo nivel**: están más próximos al funcionamiento interno de la computadora:
      - Lenguaje Ensamblador.
      - Lenguaje Máquina.
  - **Según el propósito, es decir, el tipo de problemas a tratar con ellos:**
    - Lenguajes de propósito general: Aptos para todo tipo de tareas, por ejemplo el C.
    - Lenguajes de propósito específico: Hechos para un objetivo muy concreto, por ejemplo Csound(para crear ficheros de audio).
    - Lenguajes de programación de sistemas: Diseñados para realizar sistemas operativos o drivers, por ejemplo el C.
    - Lenguajes de script: para realizar tareas de control y auxiliares. Antiguamente eran los llamados lenguajes de procesamiento por lotes (batch) o JCL("Job Control Languages").
  - **Según la manera de ejecutarse:**
    - Lenguajes compilados: Un programa traductor traduce el código del programa (código fuente) en código máquina (código objeto). Otro programa, el enlazador, unirá los ficheros de código objeto del programa principal con los de las librerías para producir el programa ejecutable. Ejemplo el C.
    - Lenguajes interpretados: Un programa (interprete), ejecuta las instrucciones del programa de manera directa. Ejemplo el Lisp.
    - También los hay mixtos, como Java, que primero pasan por una fase de compilación y luego es interpretado.

- **Según el paradigma de programación:**
  - Lenguajes imperativos: Indican cómo hay que hacer la tarea, es decir, expresan los pasos a realizar. Ejemplo el C.
  - Lenguajes declarativos: Indican que hay que hacer. Ejemplos: Lisp, Prolog. Otros ejemplos de lenguajes declarativos, pero que no son lenguajes de programación, son HTML (para describir páginas Web) o SQL (para consultar Bases de datos).
  - Lenguajes de Programación Estructurados: Usan la técnica de programación estructurada. Ejemplos: Pascal, C, etc.
  - Lenguajes de Programación Orientados a Objetos: Usan la técnica de programación orientada a objetos. Ejemplos: C++, Java, Ada, Delphi, etc.

A pesar de la inmensa cantidad de lenguajes de programación existentes, Java, C, C++, PHP y Visual Basic concentran alrededor del 60% del interés de la comunidad informática mundial.

6.2.- Lenguajes de programación estructurados.

Aunque los requerimientos actuales de software son bastante más complejos de lo que la técnica de programación estructurada es capaz, es necesario por lo menos conocer las bases de los Lenguajes de Programación estructurados, ya que a partir de ellos se evolucionó hasta otros lenguajes y técnicas más completas (orientada a eventos u objetos) que son las que se usan actualmente.
La programación estructurada se define como una técnica para escribir lenguajes de programación que permite sólo el uso de tres tipos de sentencias o estructuras de control:

- Sentencias secuenciales.
- Sentencias selectivas (condicionales).
- Sentencias repetitivas (iteraciones o bucles).

Los lenguajes de programación que se basan en la programación estructurada reciben el nombre de lenguajes de programación estructurados.

La programación estructurada fue de gran éxito por su sencillez a la hora de construir y leer programas. Fue sustituida por la programación modular, que permitía dividir los programas grandes en trozos más pequeños (siguiendo la conocida técnica "divide y vencerás"). A su vez, luego triunfaron los lenguajes orientados a objetos y de ahí a la programación visual (siempre es más sencillo programar gráficamente que en código).

- **Ventajas de la programación estructurada**

- Los programas son fáciles de leer, sencillos y rápidos.
- El mantenimiento de los programas es sencillo.
- La estructura del programa es sencilla y clara.

- **Inconvenientes**
  - Todo el programa se concentra en un único bloque (si se hace demasiado grande es difícil manejarlo).
  - No permite reutilización eficaz de código, ya que todo va "en uno". Es por esto que a la programación estructurada le sustituyó la programación modular, donde los programas se codifican por módulos y bloques, permitiendo mayor funcionalidad.

Ejemplos de lenguajes estructurados: Pascal, C, Fortran.

La Programación estructurada evolucionó hacia la Programación modular, que divide el programa en trozos de código llamados módulos con una funcionalidad concreta, que podrán ser reutilizables

6.3.- Lenguajes de programación orientados a objetos.

Después de comprender que la programación estructurada no es útil cuando los programas se hacen muy largos, es necesaria otra técnica de programación que solucione este inconveniente. Nace así la Programación Orientada a Objetos (en adelante, P.O.O.).
Los lenguajes de programación orientados a objetos tratan a los programas no como un conjunto ordenado de instrucciones (tal como sucedía en la programación estructurada) sino como un conjunto de objetos que colaboran entre ellos para realizar acciones.

En la P.O.O. los programas se componen de objetos independientes entre sí que colaboran para realizar acciones
Los objetos son reutilizables para proyectos futuros.

Su primera desventaja es clara: no es una programación tan intuitiva como la estructurada.

A pesar de eso, alrededor del 55% del software que producen las empresas se hace usando esta técnica.
Razones:

- El código es reutilizable.
- Si hay algún error, es más fácil de localizar y depurar en un objeto que en un programa entero.

- **Características:**

- Los objetos del programa tendrán una serie de atributos que los diferencian unos de otros.
- Se define clase como una colección de objetos con características similares.
- Mediante los llamados métodos, los objetos se comunican con otros produciéndose un cambio de estado de los mismos.
- Los objetos son, pues, como unidades individuales e indivisibles que forman la base de este tipo de programación.

Principales lenguajes orientados a objetos: Ada, C++, VB.NET, Delphi, Java, PowerBuilder, etc.

7.- Herramientas de apoyo al desarrollo del software.

7.1.- IDE

Para llevar a cabo la codificación y prueba de los programas se suelen utilizar entornos de programación. Estos entornos nos permiten realizar diferentes tareas:

- Crear, editar y modificar el código fuente del programa.
- Compilar, montar y ejecutar el programa.
- Examinar el código fuente.
- Ejecutar el programa en modo depuración.
- Generar documentación.
- Realizar control de versiones.
- Etc.

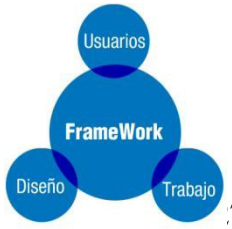
A estos entornos se les denomina entornos de desarrollo integrado o IDE (Integrated Development Environment).

La mayoría de los IDEs actuales propocionan un entorno de trabajo visual formado por ventanas, barras de menús, barras de herramientas, paneles laterales para presentar la estructura en abrbol de los proyectos o del codigo del programa que estamos editando,etc. Los editores suelen ofrecer facilidades como el resaltado de la sintaxis uitlizando diferentes colores y tipos de letra, etc.

Un mismo IDE puede funcionar con varios lenguajes de programación, este es e l caso de Eclipse o Netbeans, que mediante la instalación de plugins proporcinan soporte a lenguajes adicionales.

7.2.- Frameworks.

Un framework es una estructura de ayuda al programador, en base a la cual podemos desarrollar proyectos sin partir desde cero. Se trata de una plataforma software donde están definidos programas soporte, bibliotecas,



lenguaje interpretado, etc., que ayuda a desarrollar y unir los diferentes módulos o partes de un proyecto.

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 cualquier 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 framework en nuestro equipo consume bastantes recursos del sistema.

Ejemplos de Frameworks:

- **.NET** es un framework para desarrollar aplicaciones sobre Windows. Ofrece el "Visual Studio .net" que nos da facilidades para construir aplicaciones y su motor es el ".Net framework" que permite ejecutar dichas aplicaciones. Es un componente que se instala sobre el sistema operativo.
- Spring de Java. Son conjuntos de bibliotecas (API's) para el desarrollo y ejecución de aplicaciones.