TEMA 1:

Desarrollo del Software.

Ciclo de Desarrollo de Aplicaciones Web

Módulo: Entornos de Desarrollo

1- Introducción

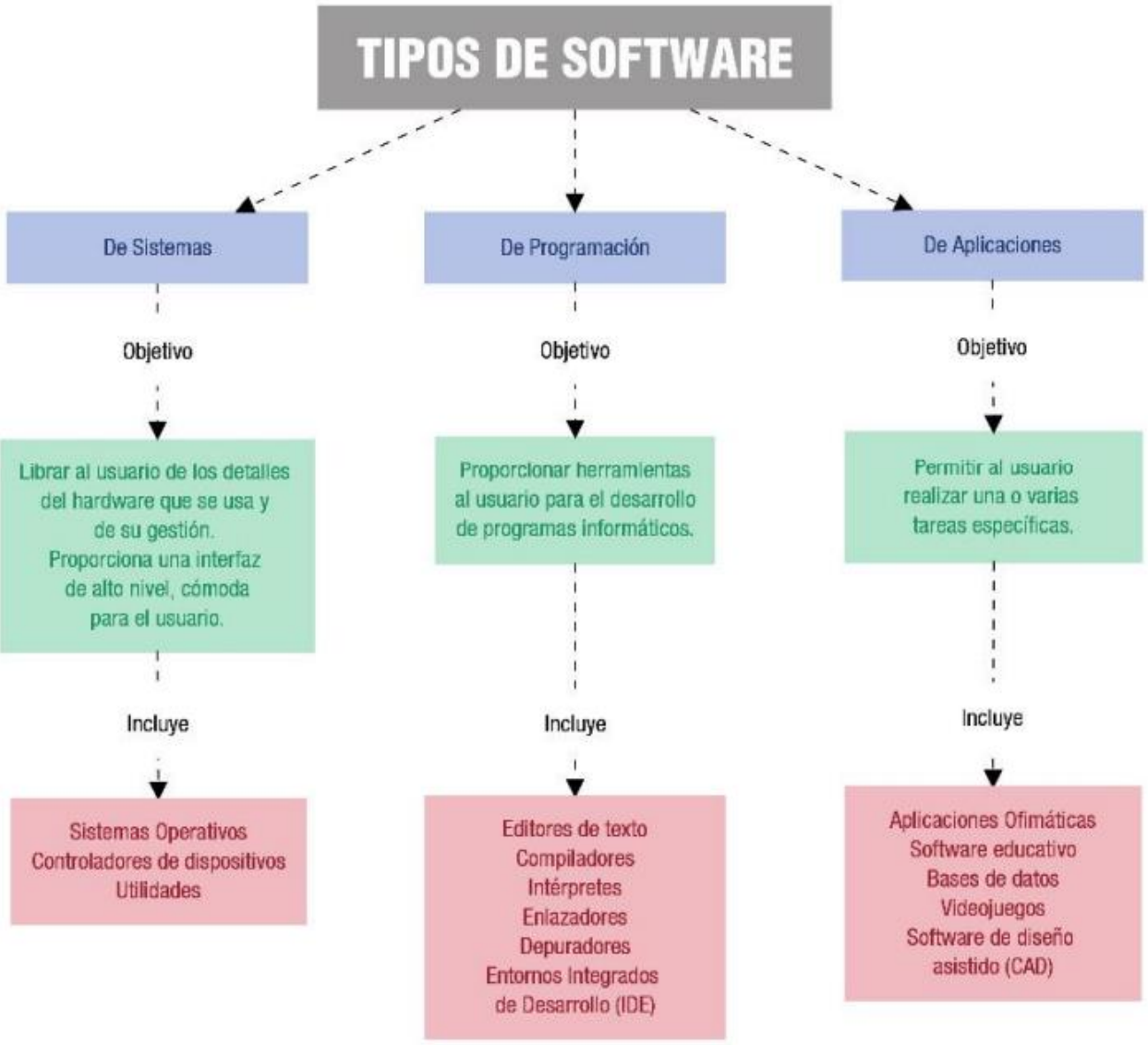
El software es la parte intangible de un sistema informático, el equivalente al equipamiento o soporte lógico: Lo constituyen los componentes lógicos (no físicos) y, por tanto, no tangibles.

Todo software está diseñado para realizar una tarea determinada en nuestro sistema.

El software es el encargado de comunicarse con el hardware, es decir, se traduce todas las órdenes que el usuario codifica en un lenguaje de programación (software) en órdenes comprensibles por el hardware.

Puede definirse software como "el conjunto de los programas de cómputo, procedimientos, reglas, documentación y datos asociados que forman parte de las operaciones de un sistema de computación" (definición extraída del estándar 729 de IEEE).

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



Un poco de historia…

El concepto de software fue utilizado por Charles Babbage (1791-1871) hace mucho tiempo. Aunque no utilizó la palabra "software" como tal, en sus trabajos ya distinguía claramente entre la *máquina* (lo que llamaríamos hardware) y las *instrucciones* que debía ejecutar (lo que hoy llamamos software).

Si visitas Londres, en el Science Museum, podrás ver algunos mecanismos *inconclusos* que Charles Babbage desarrolló. Y si quieres conocer más sobre este genio, parte de su cerebro se encuentra conservado en formol en el Royal College of Surgeons de la misma ciudad.

Junto a Babbage destaca **Ada Lovelace (1815-1852)**, hija del poeta Lord Byron. Ada fue la primera persona en escribir un programa para ser ejecutado en una máquina, lo que la convierte en la **primera**

programadora de la historia. Ella ya predijo que las máquinas no solo servirían para calcular números, sino que podrían manipular símbolos, música o texto. El lenguaje de programación ADA lleva su nombre en su honor.

Posteriormente, Alan Turing (1912-1954) profundizó en el concepto de software. Este desarrolló su carrera como matemático, pero destacó en su momento como *informático criptógrafo*. Sin embargo, sus logros se vieron truncados, ya que, tras ser acusado y procesado por ser homosexual, se suicidó.

Turing contribuyó a descifrar la máquina *Enigma* de los alemanes durante la Segunda Guerra Mundial. Elaboró la *máquina de Turing* y desarrolló la teoría de la computación, la cual se tiene como referente hoy en día y forma parte del origen del software moderno. Se considera a Turing uno de los padres de la ciencia de la computación, antecedente de la informática moderna.

En 2009 el gobierno británico pidió disculpas oficialmente por su persecución, y hoy su figura es un símbolo de la informática y de los derechos humanos. Incluso aparece en el billete de £50 del Reino Unido.

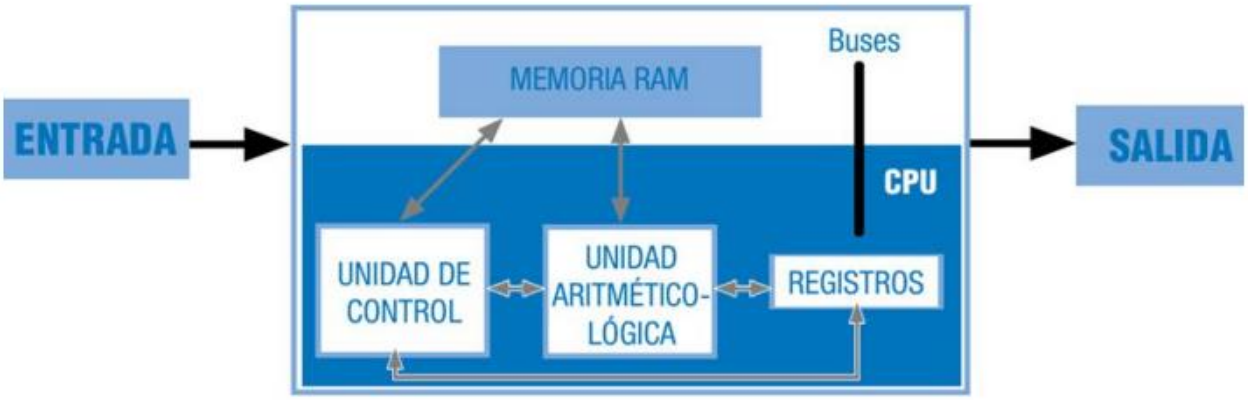
La palabra software, entendida como tal, la empleó por primera vez John Wilder Tukey (1915-2000) en un artículo de la revista American Mathematical Monthly en 1958. En este artículo, donde se empleó el término *computer software* por primera vez, se hablaba de aprovechar las capacidades de cálculo de los ordenadores de tal manera que los programadores pudiesen escribir conjuntos de instrucciones (programas), los cuales podrían llegar a ser complejos, y que luego se traducirían en otras más comprensibles para las máquinas en las que fueran a ser ejecutadas.

A partir de los años 60 y 70, con el auge de los primeros sistemas operativos y lenguajes de programación de alto nivel, el software empezó a crecer como disciplina propia, hasta llegar al desarrollo actual: aplicaciones móviles, videojuegos, inteligencia artificial o sistemas de control que usamos cada día.

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.

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 (lenguajes de Alto Nivel).

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.

¿Sabrías decirme que significa la palabra *bit*? (Tiempo 5 min)

¿Qué es el *Firmware*?¿Forma parte del Hardware o del Software? (Tiempo 5 min)

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 se refiere a software que otorga a los usuarios libertad para usarlo, estudiarlo, modificarlo, mejorarlo y redistribuirlo. Suelen ser distribuidos bajo la Licencia Pública General (GPL).

Software semi libre: aquél que no es libre, pero viene con autorización de usar, copiar, distribuir y modificar para particulares **sin fines de lucro** y con algunas limitaciones por parte del propietario.

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.

Freeware: se usa comúnmente para programas que permiten el uso gratuito y 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 uso, pero con limitaciones (opciones avanzadas deshabilitadas) o que requieren de un pago si su uso es 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

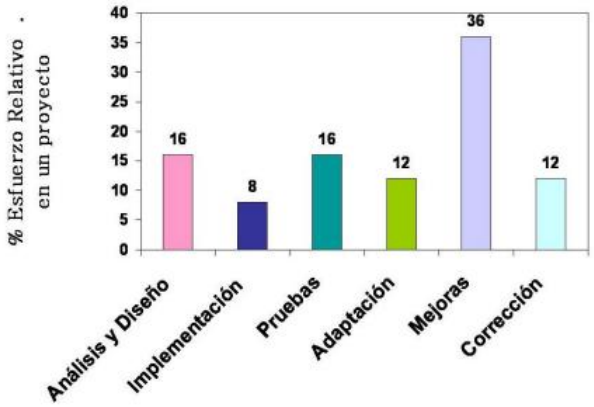
Entendemos por Desarrollo de Software todo el proceso que ocurre desde que se concibe una idea hasta que un programa está implementado en el ordenador y funcionando.

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 **Análisis, Diseño, Codificación, Pruebas, Documentación, Explotación y Mantenimiento**. Este es un enfoque bastante común, pero tampoco es raro encontrarnos proyectos en los que la fase de Documentación vaya incluida en cada una de las fases anteriores: es decir, en cada fase se va documentando. Esto sería lo ideal, pero como muchas veces el tiempo en cada fase es muy justo, es preferible planificar una fase específica de Documentación.

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 (de una manera orientativa, ya que vemos que, por ejemplo, no está la Fase de Documentación; o la fases de Análisis y Diseño van juntas) cuáles son **los costes** que supone cada fase sobre el total de un proyecto.



¿Sabes que es la *oferta* que la empresa creadora de software presenta a un cliente? (Tiempo 5 min)

¿Sabes que es un cronograma o Diagrama de Gantt? Investiga sobre ello. (Tiempo 10 min)

Veamos cada una de las fases con un poco mas de detalle.

4.1. Fase de Análisis

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:

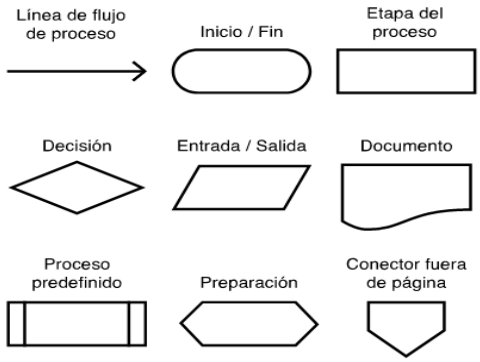
- **Funcionales:** Funciones que tendrá que realizar el futuro software, qué respuesta dará la aplicación ante todas las entradas y cómo se comportará la aplicación en situaciones inesperadas.
- **No funcionales:** Restricciones que afectaran al sistema. Tiempos de respuesta del programa, legislación aplicable, tratamiento ante la simultaneidad de peticiones, etc.

REQUISITOS FUNCIONALES	REQUISITOS NO FUNCIONALES
El software debe buscar las habitaciones disponibles y mostrarlas en pantalla.	Los datos de la aplicación solo podrán ser modificados por aquellas personas autorizadas para ello (Administrador, Recepcionista)
El software debe almacenar los datos básicos del cliente (nombre, teléfono, cedula, dirección)	La licencia de uso de software donde se aloje y con el que se realice la aplicación debe ser lo menos restrictiva posible, preferentemente software de código abierto.
El software debe almacenar los diferentes consumos hechos por el usuario durante su estadía en el hotel dentro de una factura para ser impresa.	La base de datos deberá disponer de un pool de conexiones configurables en número para que la aplicación sea escalable de función de los recursos hardware y software disponibles.
El software debe almacenar que habitaciones están fuera de servicio, cuales están sucias y cuales están limpias.	El sitio web deberá tener una estructura clara, ordenando el contenido y las funciones de la aplicación en pestañas o apartados que abarquen todas las funcionalidades disponibles, según el perfil de seguridad del usuario conectado.

Para representar los requisitos del sistema se utilizan diferentes técnicas:

- **Diagramas de flujo de datos DFD:** Consiste en representar el flujo y los pasos a realizar para la resolución del problema de una manera gráfica, utilizando para ello flechas (que representan el flujo) y una serie de símbolos predefinidos que representan las acciones a realizar.

  Algunos de los símbolos más utilizados son:



- **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**. 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 de Análisis 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].

4.2. Fase de Diseño

Ya tengo perfectamente analizado el problema e identificados todos los requisitos → ahora tenemos que diseñar la solución para ellos. En esta etapa hay que seleccionar, por ejemplo, el lenguaje de programación, el Sistema Gestor de Base de Datos, etc.

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. La elección de uno u otro dependerá en gran medida del paradigma de programación que utilizaremos.

- **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.
  - **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 y su maquetación. Es una etapa bastante artística.
  - **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.

- **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 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 Unificado, basado en diagramas que sirven para expresar modelos y que se ha convertido en un estándar de las metodologías de desarrollo orientado a objetos.

4.3. Fase de Codificación

Se realiza la traducción de nuestro diseño a un lenguaje de programación. Es decir, el programador recibe el diseño y lo codifica.

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.

El proceso de traducción/codificación.

❖ **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 un Lenguaje 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.

La labor de traducción del código fuente a código máquina puede ser realizada de dos formas que dependerá del lenguaje de programación: por Compilación o por Interpretación.

- **Interpretación:** Llevada a cabo por el *interprete*, consiste en ir traduciendo el código fuente línea a línea. Se traduce una línea y se ejecuta, y así sucesivamente.

- **Compilación:** Llevada a cabo por el *compilador*, consiste en traducir el código fuente a **código objeto**. El código objeto tiene en cuenta el hardware de la máquina en la que es compilado, de tal forma que si se cambia el hardware, habría que compilar el código fuente de nuevo. Por lo tanto, el código objeto es el resultado de compilar el código fuente, y puede ser de dos tipos:
  - **Código máquina:** será necesario enlazarlo (combinar todos los ficheros de código objeto y las librerías) para crear el programa ejecutable. El proceso de enlazado es realizado por el *enlazador* o *linker*.
  - **Código intermedio:** o bytecode que será interpretado por una Virtual Machine VM.

❖ **Código Ejecutable.** Es el resultado de compilar (por el compilador) y enlazar (por el linker) el código objeto con las librerías.

Entornos de Ejecución.

Un entorno de ejecución (runtime environment) es un conjunto de componentes software que **soportan la ejecución** de un programa (bibliotecas, configuraciones, VM) durante la **fase de tiempo de ejecución**. **Es un conjunto de utilidades que permiten la ejecución de programas a partir de código intermedio (bytecode)**.

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).

Un Entorno de Ejecución está formado por la máquina virtual y los API's (bibliotecas de clases estándar, necesarias para que la aplicación escrita en algún Lenguaje de Programación pueda ser ejecutada). Estos dos componentes **se suelen distribuir conjuntamente**, porque necesitan ser compatibles entre sí.

Vemos que el entorno funciona como intermediario (junto con el Compilador) entre el lenguaje fuente y el sistema operativo, lo que **consigue aislar el código intermedio de las características físicas de la computadora en la cual queremos ejecutarlo**.

Veamos el caso concreto de JAVA.

El lenguaje está diseñado para ser independiente de la plataforma, lo que significa que el código Java puede ejecutarse en cualquier dispositivo que tenga instalada una máquina virtual Java (JVM). Es decir, es Multiplataforma.

El Java Development Kit (JDK) es un kit de desarrollo de software que incluye todas las herramientas necesarias para crear aplicaciones Java. Componentes del JDK:

- **Compilador Java:** El compilador Java es una herramienta que convierte el código fuente Java en bytecode, que puede ser ejecutado por la JVM. El compilador comprueba la sintaxis del código y genera un mensaje de error si el código contiene algún error de sintaxis.
- **Depurador de Java:** El depurador de Java es una herramienta que permite a los desarrolladores depurar su código Java. Permite a los desarrolladores recorrer su código línea por línea, establecer puntos de interrupción e inspeccionar variables para encontrar errores en su código.
- **Entorno de ejecución de Java (JRE):** El JRE es un subconjunto del JDK que incluye todo lo necesario para ejecutar aplicaciones Java. El JRE incluye la **Java Virtual Machine (JVM)**, responsable de traducir el bytecode a código máquina y la **biblioteca de clases Java (API)**, pero no incluye el compilador Java ni otras herramientas de desarrollo.

4.4. Fase de 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.

4.5. Fase de 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 (mantenimientos de calidad).

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

4.6. Fase de Explotación (o arranque)

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. Es decir, el *arranque*.

En el proceso de instalación o arranque, 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.

4.7. Fase de 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 en la mayoría de las ocasiones.

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.

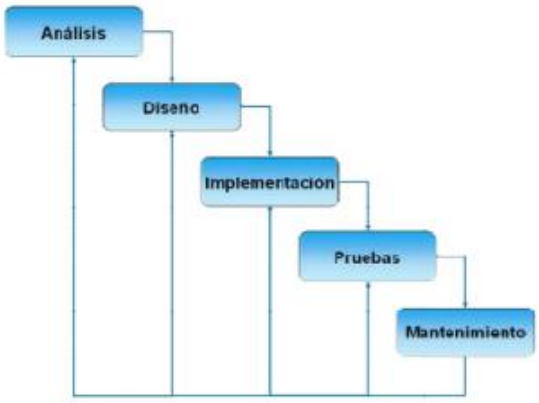
Los tipos de cambios que hacen necesario el mantenimiento del software son los siguientes:

- **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).

5- Modelos de Ciclo de vida

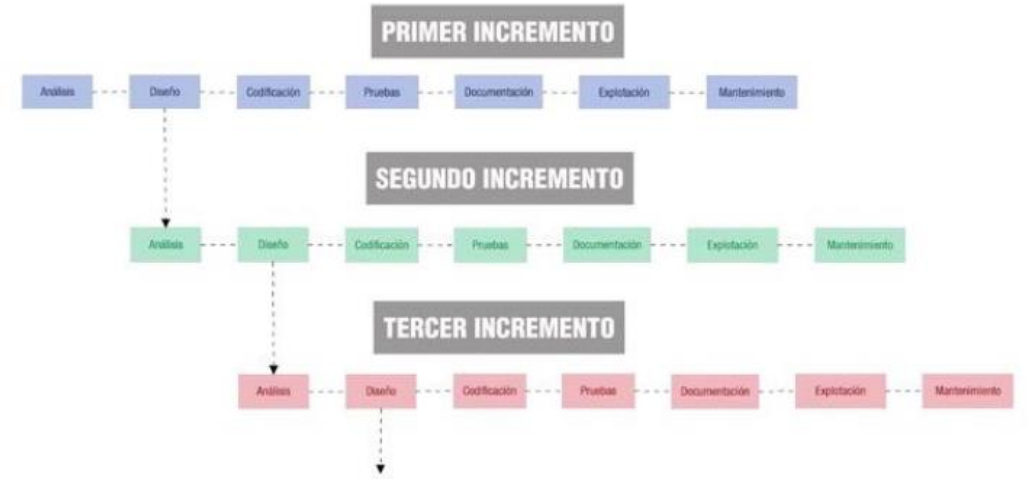
Diversos autores han planteado distintos modelos de ciclos de vida, pero los populares y utilizados son: **Cascada, Incremental y Espiral**.

- **Modelo en Cascada.** Es el modelo de vida clásico del software. **Requiere finalizar una etapa para realizar la siguiente y no se puede retornar de una etapa a otra anterior.** Por este motivo es muy complicado de 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 que nos hagan retroceder en las etapas).



- **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 pueden 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:
  - **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.

- **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**.

6- Lenguajes de Programación

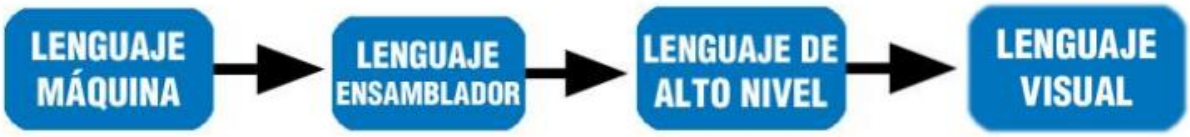
Un **lenguaje de programación** es la herramienta que me va a permitir crear programas en una computadora.

Recordemos que un programa es el conjunto de instrucciones, en un lenguaje de programación determinado, que representa un algoritmo y que interpreta o compila el ordenador para resolver un problema. Es decir, un programa es la codificación de un algoritmo en un lenguaje de programación. Los lenguajes de programación son solo un medio para expresar el algoritmo y el ordenador un procesador para ejecutarlo.

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

En otras palabras, 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 nemotécnicos (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**. En términos generales, un lenguaje de programación está definido a tres niveles:
  - Nivel SINTACTICO. Es donde se define como se combinan los elementos del lenguaje para formar instrucciones correctas.
  - Nivel SEMÁNTICO. Es donde se define el significado e interpretación de las instrucciones.
  - Nivel LÉXICO. Es donde se definen los elementos que forman parte del lenguaje.
- **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 JAVA o C++.
  - Lenguajes de propósito específico: Hechos para un objetivo muy concreto, por ejemplo Csound (para crear ficheros de audio), Julia o SQL.
  - 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"). Por ejemplo, Javascript, Python o Lua.
- 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 "clásicos", 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.

Según índices recientes (p.ej., TIOBE), el **top-5 de lenguajes** concentra alrededor del **55–60%** del indicador, aunque el conjunto varía con el tiempo (actualmente: **Python, C++, C, Java y C#**).

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", entre otras). 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 nos 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, más del 60% 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.
- La POO se apoya en cuatro pilares fundamentales:
  - **Encapsulación**: ocultar el estado tras métodos públicos.
  - **Abstracción**: exponer solo lo esencial.
  - **Herencia**: especializar comportamientos.
  - **Polimorfismo**: un mismo mensaje, distintas respuestas según el tipo.

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

7- Herramientas de apoyo al desarrollo del software.

7.1. Entorno de Desarrollo Integrado 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 proporcionan un entorno de trabajo visual formado por ventanas, barras de menús, barras de herramientas, paneles laterales para presentar la estructura en árbol de los proyectos o del código del programa que estamos editando, etc. Los editores suelen ofrecer facilidades como el resaltado de la sintaxis utilizando diferentes colores y tipos de letra, etc.

Un mismo IDE puede funcionar con varios lenguajes de programación, este es el caso de Eclipse o Netbeans, que mediante la instalación de plugins proporcionan 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 Boot** de Java. Son conjuntos de bibliotecas (API's) para el desarrollo y ejecución de aplicaciones.
- **Django** para Python.

También es común desarrollar Frameworks a medida para grandes corporaciones, como por ejemplo el framework **openFWPA del Principado de Asturias**:

El openFWPA es esencialmente un framework de desarrollo para sistemas de administración electrónica y gobierno electrónico basado en **la tecnología J2EE** y que permiten facilitar el diseño, implementación y mantenimiento de las aplicaciones. Consiste en más de 100.000 líneas de código desarrolladas por el Principado de Asturias.

El desarrollo de aplicaciones a partir de un framework como este plantea una serie de ventajas:

- **Reutilización:** Numerosos componentes sólo han de ser configurados, y no es necesario desarrollarlos de nuevo cada vez.
- **Homogeneización:** Las aplicaciones tienen la misma estructura y los mismos elementos, lo que simplifica su desarrollo, mantenimiento y gestión.
- **Mayor calidad:** Pueden establecerse criterios y objetivos de calidad, basados en métricas.
- **Menor coste:** Tanto de mantenimiento como de formación.

Los dos grandes objetivos perseguidos en el diseño y construcción de openFWPA son: Simplificación y homogeneización del proceso de desarrollo de aplicaciones y definición de estándares de desarrollo, calidad y aceptación.