1. DESARROLLO DE SOFTWARE.

1 INTRODUCCIÓN

El ordenador se compone de dos partes:

- **Hardware:** Componentes físicos que se pueden ver y tocar: la placa base, el monitor, el teclado, el ratón, la memoria RAM,…
- **Software:** Parte lógica del ordenador. Conjuntos de programas y aplicaciones que actúan sobre el hardware y facilitan al usuario la realización de tareas.

2 EL SOFTWARE DEL ORDENADOR

Es posible dar diferentes definiciones de la palabra software, por ejemplo:

El diccionario de la Real Academia Española (RAE) define software como “el conjunto de programas, instrucciones y reglas informáticas para ejecutar ciertas tareas en una computadora”.

El estándar 729 del IEEE (Institute of Electrical and Electronics Engineers), define 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”.

En Wikipedia se conoce como software al “soporte lógico de un sistema informático, que comprende el conjunto de los componentes lógicos necesarios que hacen posible la realización de tareas específicas, en contraposición a los componentes físicos que son llamados hardware. La interacción entre el software y el hardware hace operativo un ordenador (u otro dispositivo), es decir, el Software envía instrucciones que el Hardware ejecuta, haciendo posible su funcionamiento”.

Clasificación del software:

Software basado en el tipo de tarea que realiza:

- Software de sistema
- Software de aplicación
- Software de programación o desarrollo

Software basado en el método de distribución:

- Shareware
- Freeware
- Adware
- Software multimedia
- Software de uso específico

Teniendo en cuenta la licencia:

- Software libre
- Software propietario
- Software de dominio público

2.1 Software basado en el tipo de tarea que realiza:

- **Software de sistema**; es aquel que permite que el hardware funcione. Lo forman los programas que permiten la administración de la parte física o los recursos del ordenador, y es el que interactúa entre el usuario y los componentes hardware del ordenador.

- **Software de aplicación**: lo forman los programas que nos ayudan a realizar tareas específicas en cualquier campo susceptible de ser automatizado o asistido. Este software hace que el ordenador sea una herramienta útil para el usuario.

- **Software de programación o desarrollo**; es el que proporciona al programador herramientas para ayudarle a escribir programas informáticos y a usar diferentes lenguajes de programación de forma práctica. Entre ellos se encuentran los entornos de desarrollo integrados (IDE), que agrupan las anteriores herramientas, normalmente en un entorno visual, de forma que el programador no necesite introducir múltiples comandos para compilar, interpretar, depurar, etc. Habitualmente, cuentan con una avanzada interfaz gráfica de usuario (GUI).

2.2 Software basado en el método de distribución:

- **Shareware**. Es una modalidad de distribución de software, para que el usuario pueda evaluar de forma gratuita el producto por un tiempo especificado. Para adquirir una licencia de software que permita el uso del software de manera completa se requiere de un pago.

- **Freeware**. Es un software que se distribuye sin cargo. A veces se incluye el código fuente, pero no es lo usual. El freeware suele incluir una licencia de uso, que permite su redistribución pero con algunas restricciones, como no modificar la aplicación en sí, ni venderla y dar cuenta de su autor.

- **Adware**. Suelen ser programas Shareware que de forma automática descargan publicidad en nuestro ordenador cuando lo ejecutamos o instalamos. Hemos de estar atentos a la hora de instalarlos porque a veces se puede evitar su descarga. Al comprar la licencia del programa se elimina la publicidad.

- **Software multimedia**. Se refiere a los programas utilizados para presentar de una forma integrada textos, gráficos, sonidos y animaciones. Este tipo de software es considerado como una nueva tecnología. Sobre todos se utiliza en el ámbito educativo.

- **Software de uso específico**. Este tipo de software es el que se desarrolla especialmente para resolver un problema determinado de alguna organización o persona; utilizar este software requiere de expertos en informática para su creación o adaptación.

2.3 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, en el cual se definen con precisión los derechos y deberes de ambas partes. Es el desarrollador, o aquel a quien éste haya cedido los derechos de explotación, quien elige la licencia según la cual distribuye el software.

Software libre es aquel en el cual el autor cede una serie de libertades básicas al usuario, en el marco de una licencia, que establece las siguientes libertades:

1. Libertad de utilizar el programa con cualquier fin en cuantos ordenadores desee.
2. Libertar de estudiar cómo funciona el programa y de adaptar su código a necesidades específicas; para ello, como condición previa, es necesario poder acceder al código fuente.
3. Libertad de distribuir copias a otros usuarios (con o sin modificaciones).
4. Libertad de mejorar el programa (ampliarlo, añadir funciones) y de hacer públicas y distribuir al público las modificaciones; para ello, como condición previa, es necesario poder acceder al código fuente.

A diferencia del software libre y de fuentes abiertas, el **software propietario** es aquel que, habitualmente, se distribuye en formato binario, sin posibilidad de acceso al código fuente según una licencia en la cual el propietario, por regla general, prohíbe alguna o todas las posibilidades siguientes: la redistribución, modificación, copia, uso en varias máquinas simultáneamente, transferencia de titularidad, difusión de fallos y errores que se pudiesen descubrir en el programa, entre otras.

Un **software de dominio público** es aquel que carece de licencia o no hay forma de determinarla pues se desconoce el autor. Esta situación se produce bien cuando su propietario abandona los derechos que le acreditan como titular o bien cuando se produce la extinción de la propiedad por expiración del plazo de la misma, es decir, el fin del plazo de protección de los derechos de autor. El software de dominio público no pertenece a una persona concreta, sino que todo el mundo lo puede utilizar; e incluso cabe desarrollar una oferta propietaria sobre la base de un código que se encuentra en el dominio público.

La licencia más utilizada en los productos y desarrollo de software libre y de fuentes abiertas es la **licencia GPL** que da derecho al usuario a usar y modificar el programa con la obligación de hacer públicas las versiones modificadas de éste.

3 Ciclo de vida del software

El proceso de desarrollo del software implica un conjunto de actividades que se tienen que planificar y gestionar de tal manera que aseguren un producto final que dé solución a las necesidades de todas las personas que lo van a utilizar.

3.1 Definición

El estándar ISO/IEC 12207-1 define ciclo de vida del software como: *Un marco de referencia que contiene los procesos, las actividades y las tareas involucradas en el desarrollo, explotación y el mantenimiento de un producto de software, abarcando la vida del sistema desde la definición de requisitos hasta la finalización de su uso.*

El ciclo de vida de un producto software comprende el periodo que transcurre desde que el producto es concebido hasta que deja de estar disponible o es retirado. Normalmente, se divide en etapas y en cada etapa se realizarán una serie de tareas. Usualmente se consideran las siguientes etapas: especificación y análisis de requisitos, diseño del sistema, implementación del software, aplicación y pruebas, entrega y mantenimiento:

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

2. **Diseño**. En esta etapa ya sabemos 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.

3. **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.

4. **Pruebas**. Se comprueba que se cumplen criterios de corrección y calidad. Las pruebas debe garantizar el correcto funcionamiento del sistema.

5. **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.

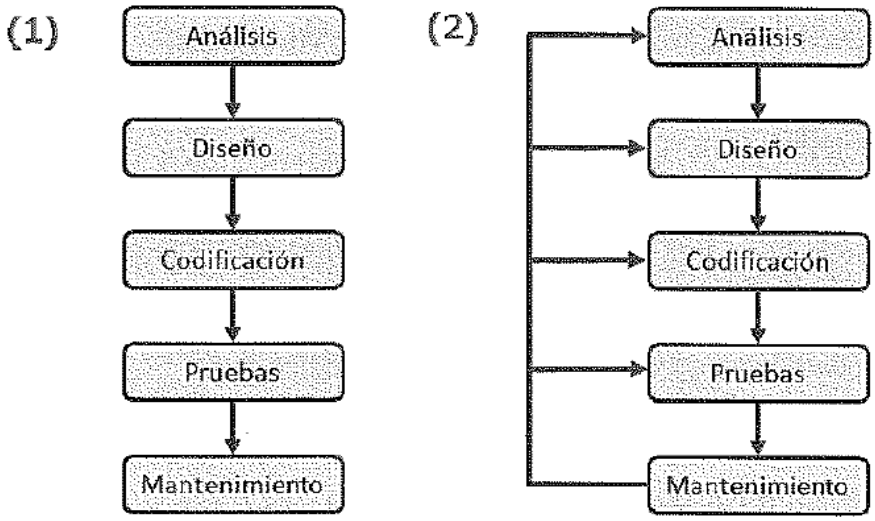
Cada etapa tiene como entrada uno o varios documentos procedentes de las etapas anteriores y produce otros documentos de salida, por ello una tarea importante a realizar en cada etapa es **la documentación**.

3.2 Modelos de ciclo de vida

Existen varios modelos de ciclo de vida y es importante tener en cuenta las características del proyecto software para elegir un modelo u otro. Algunos modelos son:

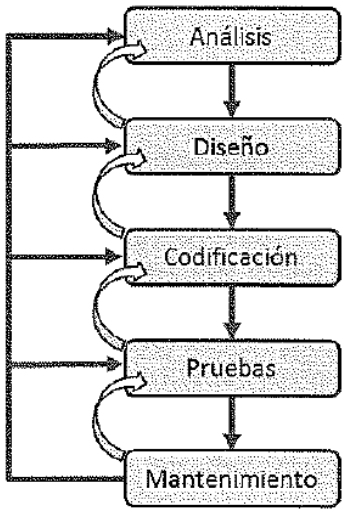
3.2.1 Ciclo de vida en cascada

En este modelo las etapas para el desarrollo del software tienen un orden, de tal forma que para empezar una etapa es necesario finalizar la anterior. Después de cada etapa se realiza una revisión para comprobar si se puede pasar a la siguiente. Este modelo permite hacer iteraciones, por ejemplo, durante la etapa de mantenimiento del producto el cliente requiere una mejora, esto implica que hay que modificar algo del diseño, lo cual significa que habrá que hacer cambios en la codificación y se tendrán que realizar de nuevo las pruebas, es decir, si se tiene que volver a una de las etapas anteriores, hay que recorrer de nuevo el resto de etapas.



Modelo en Cascada.

Tiene varias variantes, una de las más utilizadas es la que produce una realimentación entre etapas, se la conoce como Modelo en Cascada con Realimentación. Por ejemplo, supongamos que la etapa de Análisis ha finalizado y se puede pasar a la de Diseño. Durante el desarrollo de esta etapa se detectan fallos: los requisitos han cambiado, han evolucionado, ambigüedades en la definición de los mismos, etc., entonces será necesario retornar a la etapa anterior, realizar los ajustes pertinentes y continuar con el Diseño. A esto se le conoce como realimentación, pudiendo volver de una etapa a la anterior o de varias etapas a la anterior.



Modelo en Cascada con Realimentación.

Ventajas:

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

Inconvenientes:

- La necesidad de tener todos los requisitos desde el principio.
- Es difícil volver atrás si se cometen errores en una etapa.
- El producto no está disponible para su uso hasta que no está completamente terminado.

Se recomienda cuando:

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

3.2.2 Modelos evolutivos

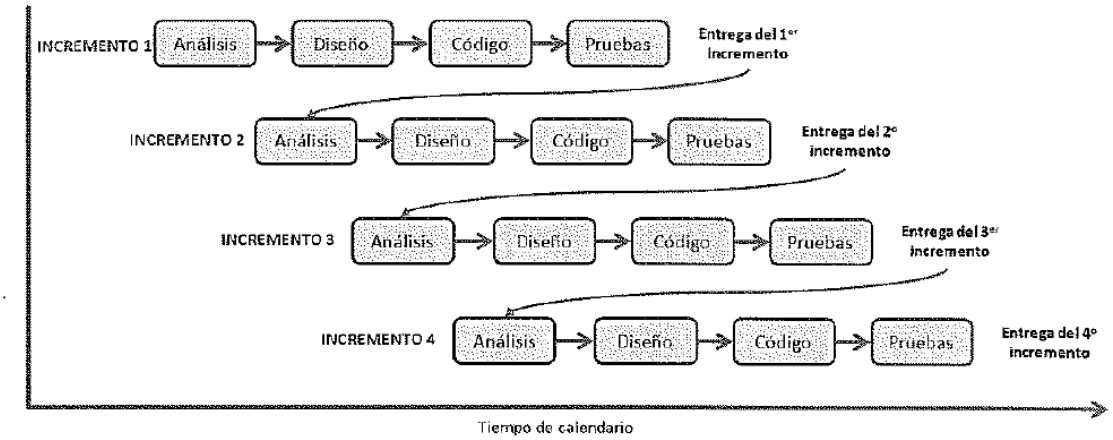
El software evoluciona con el tiempo y, es normal que los requisitos del usuario y del producto cambien conforme se desarrolla el mismo. La competencia en el mercado del software es tan grande que las empresas no pueden esperar a tener un producto totalmente completo para lanzarlo al mercado, en su lugar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

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

Los modelos evolutivos más conocidos son: el Iterativo incremental y el Espiral.

Modelo iterativo incremental

Está basado en varios ciclos cascada realimentados aplicados repetidamente. El modelo incremental entrega el software en partes pequeñas pero utilizables, llamadas "incrementos". En general, cada incremento se construye sobre aquél que ya ha sido entregado.



Modelo Iterativo Incremental.

Como ejemplo de software desarrollado bajo este modelo se puede considerar un procesador de textos: en el primer incremento se desarrollan funciones básicas de gestión de archivos y de producción de documentos; en el segundo incremento se desarrollan funciones gramaticales y de corrección ortográfica; en el tercer incremento se desarrollan funciones avanzadas de paginación y así sucesivamente.

Ventajas:

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

Inconvenientes:

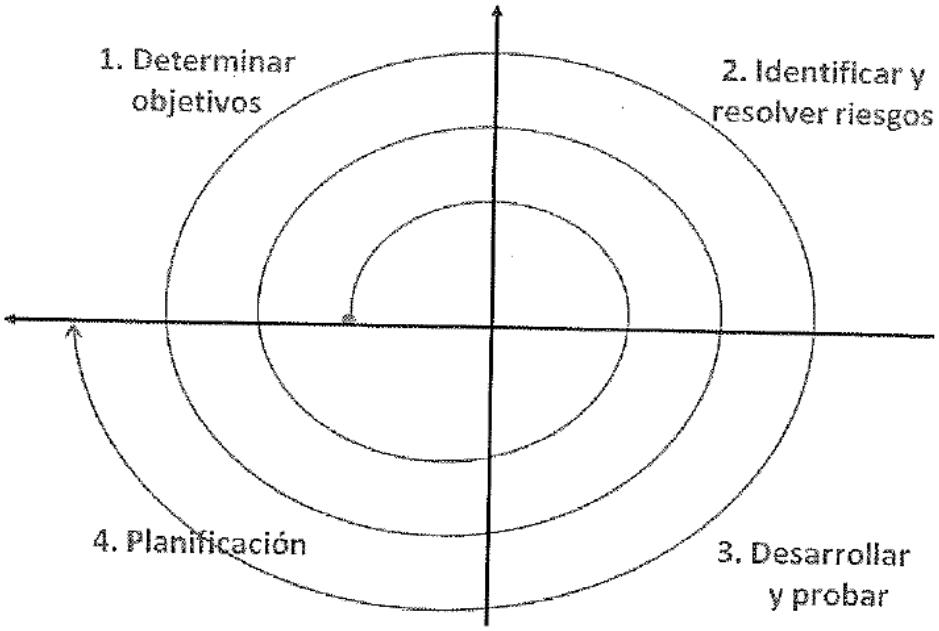
- Es difícil estimar el esfuerzo y el coste final necesario.
- Se tiene el riesgo de no acabar nunca.
- No recomendable para desarrollo de sistemas en tiempo real, de alto nivel de seguridad, de procesamiento distribuido y/o de alto índice de riesgos.

Se recomienda cuando:

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

Modelo en espiral

Este modelo combina el Modelo en Cascada con el modelo Iterativo de construcción de prototipos. El proceso de desarrollo del software se representa como una espiral, donde en cada ciclo se desarrolla una parte del mismo. Cada ciclo está formado por cuatro fases y cuando termina produce una versión incremental del software con respecto al ciclo anterior. En este aspecto se parece al Modelo Iterativo Incremental con la diferencia de que en cada ciclo se tiene en cuenta el análisis de riesgos.



Modelo en espiral.

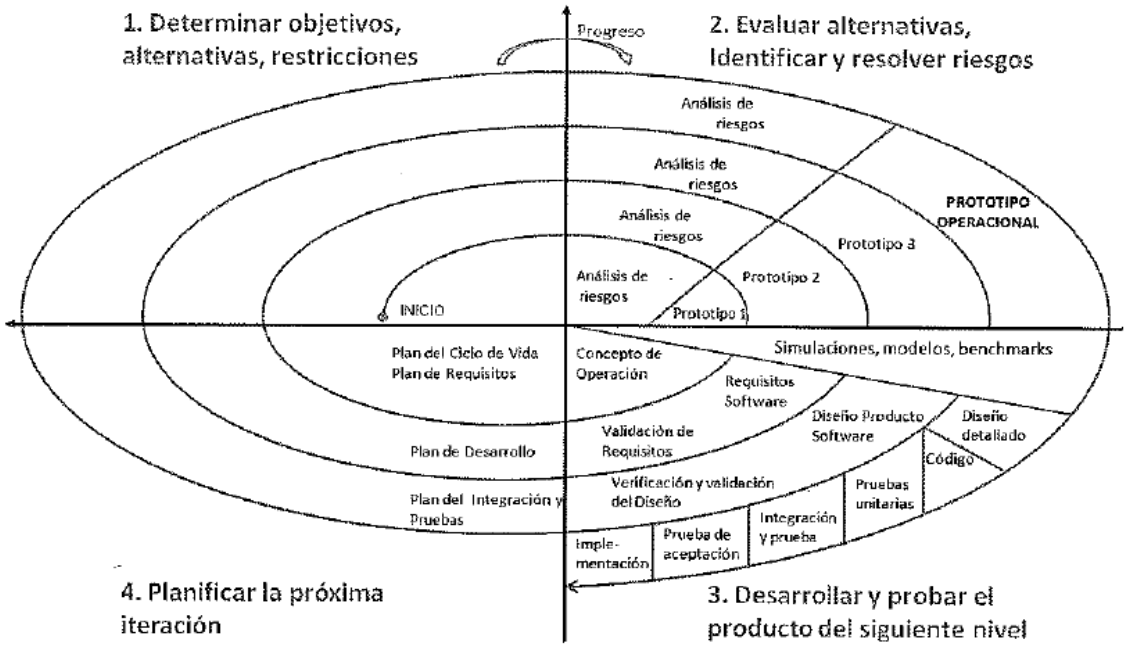
Durante los primeros ciclos la versión incremental podría estar formada por maquetas en papel o modelos de pantalla; en el último ciclo se tendrá un prototipo operacional que implementa algunas funciones del sistema. Para cada ciclo, los desarrolladores siguen estas fases:

1. **Determinar objetivos**. Cada ciclo de la espiral comienza con la identificación de los objetivos, las alternativas para alcanzar los objetivos y las restricciones impuestas a la aplicación de las alternativas.

2. **Análisis de riesgo**. A continuación hay que evaluar las alternativas en relación con los objetivos y limitaciones. Con frecuencia, en este proceso se identifican los riesgos involucrados y, si es posible, la manera de resolverlos Un riesgo puede ser cualquier cosa: requisitos no comprendidos, mal diseño, errores en la implementación, etc. Se utiliza la construcción de prototipos como mecanismo de reducción de riesgos.

3. **Desarrollar y probar**. Desarrollar la solución al problema en este ciclo, y verificar que es aceptable.

4. **Planificación.** Revisar y evaluar todo lo que se ha hecho, y con ello decidir si se continúa, entonces hay que planificar las fases del ciclo siguiente.



Ejemplo de Modelo en espiral.

En el ciclo más interno se comienza con los requisitos y un plan inicial de desarrollo; se evalúan los riesgos y se construyen prototipos de las alternativas. Para terminar, se construye un documento con el "concepto de las operaciones" que describe la funcionalidad del sistema. Al final del ciclo se genera el plan de requisitos del sistema y el plan para todo el ciclo de vida útil.

A partir de aquí comienza el segundo ciclo, el resultado será la especificación y validación de los requisitos del software y la elaboración del plan de desarrollo.

En el tercer ciclo se hace el diseño del producto software, la verificación y validación del diseño y se produce el plan de integración y pruebas.

En el cuarto ciclo los desarrolladores producen un diseño detallado; implementan los módulos, realizan pruebas unitarias, de integración y aceptación. En este punto del desarrollo se puede haber alcanzado el éxito o no, en cuyo caso será necesario otro ciclo.

En todos los ciclo se hace un análisis de riesgo, donde se evalúan las alternativas según los requisitos y restricciones, y se construyen prototipos para analizarlas y seleccionar una. Estos prototipos pueden ser modelos en papel, modelos de pantallas, simulaciones del sistema, prototipos operacionales, y dependerán del riesgo a evaluar, del ciclo en el que se esté y del tipo de aplicación.

Ventajas:

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

Inconvenientes:

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

Se recomienda para:

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

Este sistema es muy utilizado para el desarrollo de sistemas orientados a objetos.

Prototipos

El modelo de construcción de prototipos comienza con la recogida de requisitos, se definen los objetivos globales para el software y se identifican los requisitos y las áreas donde es obligatoria más definición. Entonces aparece un diseño rápido, que se centra en una representación de esos aspectos software que serán visibles al usuario o cliente. Esto lleva a la construcción del prototipo que puede servir como primer sistema construido.

El objetivo es crear un producto intermedio antes de realizar el producto final y así, con el prototipo, se puede ver cómo responde la funcionalidad prevista para el producto final.

4 FASES DE DESARROLLO DE UNA APLICACIÓN

Antes de desarrollar un proyecto hay que elegir un modelo de ciclo de vida. Será necesario examinar las características del proyecto para elegir un modelo u otro. Independientemente del modelo elegido, hay una serie de etapas que se deben seguir para construir un proyecto de calidad.

4.1 Análisis

Lo más importante del éxito de un proyecto es, entender y comprender el problema que se necesita resolver, y una vez comprendido, darle una solución. En esta fase se analizan y especifican los requisitos o capacidades que debe tener el sistema porque el cliente así lo ha solicitado.

La obtención de requisitos no es tarea fácil, porque

- El cliente puede no tener claro los requisitos,
- Pueden surgir nuevos requisitos,
- Se puede cambiar lo especificado,
- Pueden surgir malos entendidos por falta de conocimiento del equipo de desarrollo,
- Pueden surgir malos entendidos por falta de conocimiento informáticos por parte del cliente, etc.

Para realizar un proyecto satisfactorio es necesario obtener buenos requisitos, y para ello es esencial una buena comunicación entre los clientes y los desarrolladores. Algunas de las **técnicas** utilizadas en esta fase son:

- **Entrevistas**: Es la técnica más tradicional. Consiste en hablar con el cliente.

- **Desarrollo conjunto de aplicaciones (JAD):** Se apoya en la dinámica de grupos; es un tipo de entrevista muy estructurada donde cada persona juega un rol concreto.

- **Planificación conjunta de requisitos (JRP):** Es un subconjunto del JAD, que está dirigido a la alta dirección para obtener los requisitos de alto nivel o estratégicos.

- **Brainstorming:** Es un tipo de reuniones cuyo objetivo es generar ideas desde diferentes puntos de vista para la resolución de un problema.

- **Prototipos.** Es una versión inicial del sistema, se utiliza para clarificar ideas y conceptos.

- **Casos de usos:** Es una técnica definida en UML, se basa en la representación de escenarios que describen el comportamiento deseado del sistema, es decir, lo que queremos que haga el sistema. Representan requisitos funcionales del sistema. Describe lo QUÉ hace el sistema, NO CÓMO lo hace.

Se especifican 2 tipos de requisitos:

- **Requisitos funcionales:** Describen con detalle la función que realiza el sistema, cómo reacciona ante determinadas entradas, cómo se comporta ante situaciones determinadas, etc.

- **Requisitos NO funcionales:** Tratan sobre las características del sistema, como pueden ser la fiabilidad, mantenibilidad, sistema operativo, plataforma hardware, restricciones, etc.

Para representar los requisitos se utilizan diferentes técnicas:

- Diagramas de Flujo de Datos, **DFD**. Es un diagrama que representa el flujo de datos entre los distintos procesos, entidades externas y almacenes que forman el sistema.

- Diagramas de Flujo de Control, **DFC**. Similar a los DFD, pero en éstos se muestra el flujo de control.

- Diagramas de Transición de Estados, **DTE**. Representa cómo se comporta el sistema como consecuencia de sucesos externos.

- Diagrama Entidad/Relación, **DER**. Usado para representar los datos y la forma en la que se relacionan entre ellos.

- Diccionario de Datos, **DD**. Es una descripción detallada de los datos utilizados por el sistema.

Todo lo realizado en esta fase debe quedar reflejado en el documento Especificación de Requisitos del Software **(ERS)**. Este documento no debe tener ambigüedades, debe ser completo, consistente, fácil de verificar y modificar, fácil de utilizar y fácil de identificar el origen y las consecuencias de los requisitos.

4.2 Diseño

Una vez identificados los requisitos es necesario componer la forma en que se solucionará el problema.

Principalmente hay dos tipos de diseño:

1. El diseño estructurado que está basado en el flujo de los datos a través del sistema,

2. El diseño orientado a objetos, donde el sistema se entiende como un conjunto de objetos que tienen propiedades y comportamiento, además de eventos que activan operaciones que modifican el estado de los objetos.

4.2.1 El diseño estructurado

Produce un modelo de diseño con 4 elementos:

- **Diseño de datos**: Está basado en los datos y las relaciones definidas en el DER y en el DD.
- **Diseño arquitectónico**: Se centra en la representación de la estructura de los componentes del software, sus propiedades e interacciones. Un componente puede ser un módulo de un programa, una base de datos o los conectores que se utilizan entre los propios conectores. Se parte del DFD.
- **Diseño de la interfaz**: Describe cómo se comunica el software consigo mismo, con los sistemas que operan con él, y con las personas que lo utilizan. El resultado de esta tarea es la creación de formatos de pantalla
- **Diseño a nivel de componentes o procedimental**: El resultado de esta tarea el diseño de cada componente software con el suficiente nivel de detalle para que pueda servir de guía en la generación de código fuente en un lenguaje de programación. Para llevar a cabo este diseño se utilizan representaciones gráficas mediante diagramas de flujo, diagramas de cajas, tablas de decisión, pseudocódigo, etc.

4.2.2 Diseño Orientado a Objetos

Para llevar a cabo un **Diseño Orientado a Objetos** (DOO) hay que partir de un Análisis Orientado a Objetos (AOO). En el análisis se definen todas las clases que son importante para el problema que se trata de resolver, las operaciones y los atributos asociados, las relaciones y comportamientos y las comunicaciones entre clases.

El Diseño Orientado a Objetos define 4 capas de diseño:

- **Subsistema**: Se centra en el diseño de subsistemas que implementan las funciones principales del sistema.
- **Clases y objetos**: Especifica la arquitectura de objetos global y la jerarquía de clases requerida para implementar un sistema.
- **Mensajes**: Indica cómo se realiza la colaboración entre 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).

4.3 Codificación

Una vez realizado el diseño se realiza el proceso de codificación. En esta etapa, el programador recibe las especificaciones del diseño y las transforma en un conjunto de instrucciones escritas en un lenguaje de programación, almacenadas en un programa, el código fuente. El programador debe conocer la sintaxis del **lenguaje de programación** utilizado.

En cualquier proyecto en el que trabaja un grupo de personas, debe haber unas normas de codificación y estilo, claras y homogéneas. Estas normas facilitan las tareas de corrección y mantenimiento de los programas, sobre todo cuando se realizan por personas que no los han desarrollado.

Una vez generado el **código fuente**, deberá ser compilado o interpretado, conectado a las librerías necesarias, y se obtendrá el código ejecutable. En este momento, se deberá probar el programa con el fin de comprobar y verificar que cumple con las especificaciones del diseño.

A la vez que se va desarrollando código, se deben ir escribiendo los manuales técnicos y de referencia necesarios, así como la parte inicial correspondiente del Manual de Usuario. Esta documentación es esencial para la etapa de pruebas y mantenimiento, así como para la entrega final del producto.

4.4 Pruebas

Durante la prueba del software se realizarán tareas de verificación y validación del software:

- La **verificación**: Se refiere al conjunto de actividades que tratan de comprobar si se está construyendo el producto correctamente, es decir, si el software implementa correctamente una función específica

- La **validación**: Se refiere al conjunto de actividades que tratan de comprobar si el producto es correcto, es decir, si el software construido se ajusta a los requisitos del cliente.

El objetivo de esta etapa es planificar y diseñar pruebas que sistemáticamente saquen a la luz diferentes clases de errores, haciéndolo en la menor cantidad de tiempo y esfuerzo. Una prueba tiene éxito si descubre un error no detectado hasta entonces. Un caso de prueba es un documento que especifica los valores de entrada, salida esperada y las condiciones previas para la ejecución de la prueba.

4.5 Documentación

Todas las etapas del desarrollo deben quedar perfectamente documentadas. En esta etapa será necesario reunir todos los documentos generados y clasificarlos según el nivel técnico de sus descripciones.

Los documentos relacionados con un proyecto de software:

- Deben actuar como un medio de comunicación entre los miembros del equipo de desarrollo.
- Deben ser un repositorio de información del sistema para ser utilizado por el personal de mantenimiento.
- Deben proporcionar información para ayudar a planificar la gestión del presupuesto y programar el proceso del desarrollo del software.
- Algunos de los documentos deben indicar a los usuarios cómo utilizar y administrar el sistema.

Por lo general, se puede decir que la documentación presentada se divide en dos clases:

- La documentación del proceso. Estos documentos registran el proceso de desarrollo y mantenimiento.

- La documentación del producto. Describe el producto que está siendo desarrollado. Estará formada por la documentación del sistema y documentación del usuario.

4.6 Explotación

Una vez que se han realizado todas las pruebas y documentado todas las etapas, se pasa a la explotación del sistema. En esta etapa se lleva a cabo la instalación y puesta en marcha del producto software en el entorno de trabajo del cliente.

En esta etapa se llevan a cabo las siguientes tareas:

1. Se define la estrategia para la implementación del proceso. Se desarrolla un plan donde se establecen las normas para la realización de las actividades y tareas de este proceso. Se definen los procedimientos para recibir, registrar, solucionar, hacer el seguimiento de los problemas y para probar el producto software en el entorno de trabajo

2. Pruebas de operación. Para cada release del producto software, se llevarán a cabo pruebas de funcionamiento y tras satisfacerse los criterios especificados, se libera el software para uso operativo.

3. Uso operacional del sistema. El sistema entrará en acción en el entorno previsto de acuerdo con la documentación de usuario.

4. Soporte al usuario. Se deberá proporcionar asistencia y consultoría a los usuarios que lo soliciten. Estas peticiones y las acciones consecuentes se deberán registrar y supervisar.

4.7 Mantenimiento

El mantenimiento del software se define como la modificación de un producto software después de la entrega para corregir los fallos, para mejorar el rendimiento u otros atributos o para adaptar el producto a un entorno modificado.

Existen cuatro tipos de mantenimiento:

1. Mantenimiento **adaptativo**: Tiene como objetivo la modificación del producto por los cambios que se produzcan, tanto en el hardware como en el software del entorno en el que se ejecuta. Es el más usual.

2. Mantenimiento **correctivo**: Tiene como objetivo corregir los fallos descubiertos.

3. Mantenimiento **perfectivo**: Está orientado a incorporar nuevas funcionalidades y nuevas mejoras en el rendimiento o la mantenibilidad del producto.

4. Mantenimiento **preventivo**: Consiste en la modificación del producto del software sin alterar las especificaciones del mismo, con el fin de mejorar y facilitar las tareas de mantenimiento. A este mantenimiento se le llama reingeniería del software.

Las **tareas** que se llevan a cabo en esta etapa son:

- Implementación del proceso
- Análisis de problemas y modificaciones
- Implementación de las modificaciones
- Revisión/aceptación del mantenimiento
- Migración
- Retirada del software

5 CONCEPTO DE PROGRAMA

Un programa informático es un conjunto de instrucciones escritas en un lenguaje de programación que aplicadas sobre un conjunto de datos resuelven un problema o parte del mismo. Para que el programa pueda ser ejecutado es necesario traducirlo a un lenguaje entendible por el ordenador, el lenguaje máquina; esta tarea se lleva a cabo por otro programa llamado compilador o intérprete. En el caso del compilador, una vez que tenemos el programa en código entendible por la máquina hay que cargarlo en la memoria principal para que el procesador ejecute una a una todas las instrucciones.

Para ejecutar un programa se necesitan los recursos hardware del ordenador: el procesador, la memoria RAM, los dispositivos de E/S, etc. Las instrucciones de un programa se cargan en la memoria principal y la CPU será la encargada de ejecutarlas.

En la arquitectura de Von Neumann, son varios los componentes que forman la CPU:

- La Unidad de Control (UC)
- La Unidad Aritmético-Lógica (UAL)
- Los registros de trabajo

A la hora de ejecutar una instrucción se distinguen 2 fases:

- Fase de **búsqueda**. Consiste en localizar la instrucción a ejecutar dentro de la memoria principal y llevarla a la UC para procesarla.

- Fase de **ejecución**. Es la realización de las acciones que llevan asociadas las instrucciones. Por ejemplo una suma, una resta o una carga de datos.

6 LENGUAJES DE PROGRAMACION

Podemos definir un lenguaje de programación como un conjunto de caracteres, las reglas para la combinación de esos caracteres y las reglas que definen sus efectos cuando los ejecuta un ordenador. Un lenguaje de programación consta de los siguientes elementos:

- Un alfabeto o vocabulario (léxico): formado por un conjunto de símbolos permitidos.
- Una sintaxis: son las reglas que indican cómo realizar las construcciones con los símbolos del lenguaje.
- Una semántica: son las reglas que determinan el significado de cualquier construcción del lenguaje.

Los lenguajes de programación se pueden clasificar atendiendo a varios criterios.

6.1 Clasificación según su nivel de abstracción:

6.1.1 Lenguajes de bajo nivel

Son lenguajes de programación que se acercan al funcionamiento del ordenador. El lenguaje de más bajo nivel por excelencia es el lenguaje máquina, que es entendible directamente por la máquina. Las instrucciones están formadas por cadenas de ceros y unos. Los programas en este lenguaje son específicos para cada procesador.

A continuación está el lenguaje ensamblador que es difícil de aprender y es específico para cada procesador. Un programa escrito en este lenguaje necesita ser traducido a lenguaje máquina para poder ejecutarse. Se programa utilizando nombre nemotécnicos y las instrucciones trabajan directamente con registros de memoria física de la máquina.

6.1.2 Lenguajes de nivel medio

Este tipo de lenguajes tienen ciertas características que los acercan a los lenguajes de bajo nivel, pero a la vez tienen características de los lenguajes de alto nivel. Un lenguaje de programación de este tipo es el lenguaje C. Se suelen utilizar para aplicaciones como la creación de sistemas operativos.

6.1.3 Lenguajes de alto nivel

Son normalmente más fáciles de aprender porque están formados por palabras del lenguaje natural, como el inglés. Para poder ejecutarlos en el ordenador se necesita un

programa intérprete o compilador que traduzca las instrucciones escritas en este lenguaje, en instrucciones en lenguaje máquina que el ordenador pueda entender.

Los lenguajes de programación de alto nivel son independientes de la máquina, es decir, no dependen del hardware del ordenador y no requieren ningún conocimiento de código máquina por parte del usuario que lo utiliza. El alfabeto utilizado se acerca más a la del problema que se trata de resolver que al código máquina.

6.2 Clasificación según la forma de ejecución

6.2.1 Lenguajes compilados

Un programa que se escribe en un lenguaje de alto nivel debe traducirse a un código que pueda utilizar la máquina. Los programas traductores que pueden realizar esta operación se llaman compiladores o intérpretes.

Un compilador es un programa que puede leer un programa escrito en un determinado lenguaje (lenguaje fuente) y traducirlo en un programa equivalente en otro lenguaje (lenguaje destino). Hay que tener en cuenta que el compilador devolverá errores si el programa en el lenguaje fuente no está bien escrito. El programa destino se podrá ejecutar si el lenguaje destino es directamente ejecutable por la máquina.

6.2.2 Lenguajes interpretados

Los intérpretes son otra alternativa para traducir los programas escritos en lenguaje de alto nivel. En este caso, en vez de producir un programa destino como resultado del proceso de traducción, el intérprete nos da la apariencia de ejecutar directamente las operaciones especificadas en el programa fuente con las entradas proporcionadas por el usuario. Cada vez que se ejecuta una instrucción, se debe interpretar y traducir a lenguaje máquina.

El programa destino en el lenguaje máquina que produce un compilador es por lo general más rápido que un intérprete al momento de asignar las entradas a las salidas. El intérprete elimina la necesidad de realizar la compilación después de cada codificación. Algunos ejemplos de lenguajes interpretados son: PHP, Java Script, Pitón, Perl, Logo, ASP, etc.

Los procesadores de lenguaje Java combinan la compilación y la interpretación. Un programa fuente en Java puede compilarse en un formato intermedio, llamado bytecodes, y después una máquina virtual los interpreta.

6.3 Clasificación según el paradigma de programación

Un paradigma de programación es un enfoque particular para la construcción del software. Define un conjunto de reglas, patrones y estilos de programación que usan los lenguajes de programación. Un lenguaje de programación puede usar más de un paradigma. Dependiendo del problema a resolver un paradigma resultará más apropiado que otro.

Constituyen varias categorías de los lenguajes de programación: imperativos, funcionales o aplicativos y lógicos o declarativos.

6.3.1 Lenguajes imperativos

Los lenguajes imperativos fueron los lenguajes máquina de los ordenadores, en los que las instrucciones eran muy simples, después se utilizaron los lenguajes ensambladores. En los lenguajes imperativos un cálculo consiste en una serie de sentencias que establecen explícitamente cómo se debe manipular la información digital presente en memoria, y/o cómo se debe recoger o enviar información desde/hacia los dispositivos.

La sentencia principal es la **asignación**. Las estructuras de control permiten establecer el orden de ejecución y cambiar el fuljo flujo del programa dependiendo de los resultados de las acciones primitivas.

La mayoría de los lenguajes usados para desarrollo de software comercial son imperativos. Ejemplos de lenguajes imperativos son: Basic, Fortran, Algol, C, Ada, Java, C++.

Dentro de esta categoría se engloba la programación estructurada, la programación modular y la programación orientada a objetos.

6.3.2 Lenguajes funcionales

El paradigma está basado en el concepto matemático de función. Los programas escritos en lenguajes funcionales estarán constituidos por un conjunto de definiciones de funciones junto con los argumentos sobre los que se aplican.

En los lenguajes funcionales:

- No existe la operación asignación.
- Las variables almacenan definiciones o referencias a expresiones.
- La operación fundamental es la aplicación de una función a una serie de argumentos.
- La computación se realiza mediante la evaluación de las expresiones

Ejemplos de los lenguajes funcionales son: Lisp, Scheme, ML, Miranda o Haskell. Se utilizan poco en el desarrollo de software comercial.

6.3.3 Lenguajes lógicos

En este tipo de lenguajes un cálculo es el proceso de encontrar qué elementos de un dominio cumplen determinada relación definida sobre dicho dominio o el proceso de determinar si un determinado elemento cumple o no dicha relación.

Los programas escritos en estos lenguajes se pueden ver como una base de datos formada por listas de declaraciones lógicas (reglas) que se pueden consultar. La ejecución consistirá en realizar preguntas de forma interactiva.

El lenguaje lógico por excelencia es Prolog, que está especialmente indicado para aplicaciones específicas como: sistemas expertos, demostración de teoremas, consulta de bases de datos relacionales, procesamiento del lenguaje natural.

6.3.4 Lenguajes de programación estructurada

Un programa estructurado, que utiliza las tres construcciones lógicas: la estructura secuencial, la condicional y la repetitiva, resulta fácil de leer, es decir, puede ser leído secuencialmente desde el comienzo hasta el final sin perder la continuidad de lo que hace. El problema de estos programas es que todo el código se concentra en un único bloque, y si el programa es demasiado grande o el problema a resolver es complejo, resulta difícil su lectura y manejo.

Actualmente, cuando se habla de programación estructurada, se suele referir a la división de un programa en partes más manejables conocidas como módulos. Un programa estructurado puede estar compuesto por un conjunto de módulos, donde cada uno tendrá una entrada y una salida. La comunicación entre ellos debe estar perfectamente controlada y se debe poder trabajar de forma independiente con cada uno de ellos. El módulo raíz controlará el resto de los módulos.

A esta evolución de la programación estructura se le llama programación modular. Aporta una serie de ventajas:

- Al dividir el programa en varios módulos, varios programadores pueden trabajar simultáneamente.
- Los módulos se pueden reutilizar en otras aplicaciones.
- Es menos costoso resolver pequeños problemas de forma aislada que abordar el problema a resolver de forma global.

6.3.5 Lenguajes de programación orientados a objetos

En la programación orientada a objetos un programa está compuesto por un conjunto de objetos no por un conjunto de instrucciones o un conjunto de módulos.

Un objeto consta de una estructura de datos y de una colección de módulos u operaciones que manipulan esos datos. Los datos definidos dentro de un objeto son sus atributos. Las operaciones definen el comportamiento del objeto y cambian el valor de uno o más atributos. Los objetos se comunican entre ellos a través del paso de mensaje.

Una clase no es más que una plantilla para la creación de objetos. Cuando se crea un objeto se ha de especificar de qué clase es el objeto instanciado, para que el compilador comprenda las características del objeto.

Desde el punto de vista de la programación estructurada, una clase se asemejaría a un módulo, los atributos a las variables globales de dicho módulo, y los métodos a las funciones del módulo. Entre las ventajas de la programación orientada a objetos destacamos que facilita la reutilización de código, el trabajo en equipo o el mantenimiento del software. Sin embargo, la principal desventaja es la complejidad para adaptarse a esta programación ya que es menos intuitiva que la estructurada.

Ejemplos de lenguajes orientados a objetos son: C++, Java, Ada, Smalltalk, etc.

7 OBTENCIÓN DE CÓDIGO EJECUTABLE

Durante la etapa de diseño se construyen los componentes software con el suficiente nivel de detalle de tal forma que sirvan como guía en la generación de código fuente en un lenguaje de programación; algunas herramientas utilizadas para ello son los diagramas de flujo o el pseudocódigo.

La generación de código fuente se lleva a cabo en la etapa de codificación.

7.1 Tipos de código

El código de un programa pasa por diferentes estados desde que se escribe hasta que se ejecuta en el ordenador:

- **Código fuente.** Es el código escrito por los programadores utilizando algún editor de texto o alguna herramienta de programación. Se utiliza un lenguaje de programación de alto nivel apropiado para el problema que se trata de resolver. Para escribir el código se parte de los diagramas de flujo o pseudocódigos diseñados en la etapa de diseño. Este código no es directamente ejecutable por el ordenador.

- **Código objeto.** Es el código resultante de compilar el código fuente. No es directamente ejecutable por el ordenador ni entendido por el ser humano. Es un código o representación intermedia de bajo nivel.

- **Código ejecutable.** Es el resultado de enlazar el código objeto con una serie de rutinas y librerías, obteniendo así el código que es directamente ejecutable por la máquina.

7.2 Compilación

El proceso de compilación de un programa se lleva a cabo mediante dos programas, el compilador y el enlazador. Si el compilador en el proceso de traducción devuelve algún error, no se generará el programa objeto. Y será necesario modificar el programa fuente y pasarlo de nuevo por el compilador.

El compilador se compone internamente de varias etapas o fases que realizan distintas operaciones:

- Análisis léxico: se lee secuencialmente todo el código fuente obteniendo unidades significativas de caracteres denominadas tokens.

- Análisis sintáctico: recibe el código fuente en forma de tokens y realiza el análisis sintáctico que determina la estructura del programa; es decir, se comprueba si las construcciones de tokens cumplen las reglas de sintaxis definidas en el lenguaje de programación correspondiente. El proceso es semejante al análisis gramatical sobre alguna frase en lenguaje natural.

- Análisis semántico: se comprueba que las declaraciones sean correctas, se verifican los tipos de todas las expresiones, si las operaciones se pueden realizar sobre esos tipos, si los arrays tienen el tamaño y tipo adecuados, etc.

- Generación de código intermedio: después del análisis se genera una representación intermedia similar al código máquina con el fin de facilitar la tarea de traducir al código objeto.

- Optimización de código. Trata de mejorar el código intermedio generado en la fase anterior, de tal forma que el código resultante sea más fácil y rápido de interpretar por la máquina.

- Generación de código: genera el código objeto de nuestro programa.

El programa enlazador inserta en el código objeto las funciones de librería necesarias para producir el programa ejecutable. Por ejemplo, en un programa escrito en C, si el fichero fuente hace referencia a funciones de una biblioteca o a funciones que están definidas en otros ficheros fuente, entonces el enlazador combina estas funciones con el programa principal para crear un fichero ejecutable.

8 HERRAMIENTAS UTILIZADAS EN PROGRAMACIÓN

Para llevar a cabo la codificación y prueba de los programas se suelen utilizar entornos de programación. Estos entornos nos permiten realizar las siguientes 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.
- Realizar pruebas del programa de forma automática.
- Generar documentación.
- Gestionar los cambios que se van haciendo (control de versiones), etc.

A estos entornos de programación se les suele llamar entornos de desarrollo integrado o IDE. Los IDEs están diseñados para maximizar la productividad del programador. Un IDE es un programa informático formado por un conjunto de herramientas de programación que facilitan las tareas de creación, modificación, compilación, implementación y depuración de software.

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, pestañas adicionales con el resultado de aplicar determinadas herramientas o bien que nos permiten tener abiertos varios ficheros del proyecto, asistentes que ayudan al programador a desarrollar parte del proyecto o del código, etc.

Los editores pueden ofrecer facilidades como el resaltado de la sintaxis utilizando diferentes colores y tipos de letra, el emparejamiento de llaves o paréntesis poniendo el cursor sobre uno de ellos, el plegado y desplegado de código, por ejemplo una función se puede reducir a una sola línea.

Un mismo IDE puede funcionar con varios lenguajes de programación, mediante la instalación de plugins adicionales.