CURSO 2022-2023
CICLO SUPERIOR DE DESARROLLO DE APLICACIONES WEB
IES AGUADULCE

Entornos de Desarrollo

Francisco Javier Sueza Rodríguez

27 de octubre de 2022

Índice general

Índice de figuras

Tema 1

Desarrollo de Software

En esta unidad vamos a realizar una introducción al concepto de software, así como a los diferentes tipos de software que podemos encontrar y al proceso de desarrollo de éste. También hablaremos de los lenguajes de programación, en que consiste y que características tiene. Por último, veremos que son los entornos de desarrollo, cuales su función y su evolución histórica.

1.1 Software y Tipos de Software

Un ordenador se compone de dos partes bien diferenciadas, el **hardware** y el **software**.

El **hardware**, **equipo** o **soporte físico** en informática se refiere a las partes físicas, tangibles de un sistema informático, sus componentes eléctricos, electrónicos y electromecánicos. Los cables, así como los muebles o cajas de todo tipo también se incluyen dentro de esta categoría. [1]

Por otro lado, el **software** es el conjunto de componentes lógicos que hace posible la realización de tareas específicas [2]. Dicho de otra forma, es el conjunto de programas informáticos que actúa sobre el hardware para ejecutar lo que el usuario desee.

Según su funcionalidad, podemos diferencias tres tipos principales de software:

- **Sistema Operativo**: conjunto de programas de un sistema informático que gestiona los recursos hardware y provee servicios a las aplicaciones informáticas para su funcionamiento. Ejemplos de sistemas operativos son: Microsoft Windows, Linux, macOS...
- **Software de Programación**: conjunto de herramientas y utilidades que permiten a los programadores desarrollar programas informáticos.
- **Aplicaciones Informáticas**: programas o conjunto de programas que tienen una aplicación concreta. Algunos ejemplos de aplicaciones informáticas son los procesadores de texto, reproductores multimedia, juegos...

Aunque estos son los tipos principales de software, podemos ampliar esta clasificación incluyendo los siguientes tipos, los cuales son, en mayor medida, subdivisiones de las aplicaciones informáticas [3]:

- **Software de Tiempo Real**: son aquellos programas que se usan en sistemas de tiempo real y que reaccionan con el entorno físico respondiendo a los estímulos del éste en un tiempo limitado. Ejemplos de este software son los sistemas de control de procesos, aplicaciones de robótica,..

- **Software de Gestión**: son programas que utilizan grandes cantidades de información almacenadas en bases de datos para ayudar a la administración y toma de decisiones. Un ejemplo de estos sistemas son los **ERP**.
- **Software Científico o de Ingeniería**: software que se encarga de realizar complejos cálculos numéricos de todo tipo, siendo la corrección y exactitud en estos cálculos uno de los requisitos básicos. También incluye los sistemas de diseño, ingeniería y fabricación asistida por ordenador (CAD, CAE y CAM), simuladores gráficos,..
- **Software Empotrado**: software que por norma general va instalado en memorias ROM y sirve para controlar productos y sistemas de los mercados industriales. Se aplica a todo tipo de productos como neveras, reproductores de vídeo, misiles, sistemas de control de automóviles,...
- **Software de Inteligencia Artificial**: software que basándose en el uso de lenguajes declarativos, sistemas expertos y redes neuronales, para simular comportamientos humanos como el aprendizaje, razonamiento y la resolución de problemas para la realización de forma fiable y rápida operaciones que para el ser humano son tediosas o incluso inabordables. Ejemplos de estas aplicaciones son las aplicaciones de *machine learning*, chatbots,...

En este tema, nos centraremos en las **aplicaciones informáticas**, como se desarrollan y cuales son las fases de este desarrollo. También veremos el **ciclo de vida de una aplicación informática** y los diferentes tipos de **lenguajes de programación** y sus características.

1.2 Relación Hardware-Software

Como hemos comentado en el punto anterior, un sistemas informático se compone de hardware y software. Existe una relación indisoluble entre estos dos componentes ya que se necesita que estén instalados y configurados correctamente para que el ordenador funcione.

El primer modelo de arquitectura de hardware con programa almacenado fue propuesto por **John Von Neumman** en 1946. A continuación se muestra una imagen con las diferentes partes de esta arquitectura:

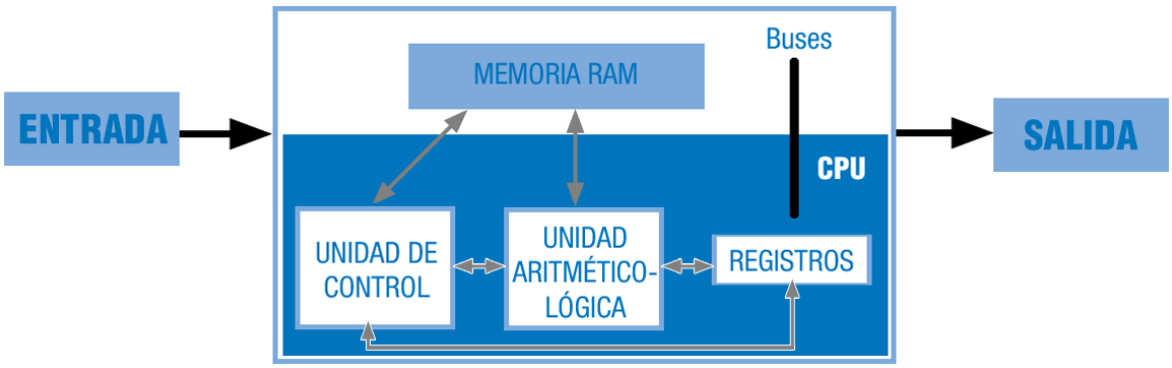

Figura 1.2.1.: Arquitectura John Von Neuman

Esta relación software-hardware la podemos analizar desde dos puntos de vista diferentes:

a **Desde el punto de vista del sistema operativo**: el sistema operativo es el encargado de coordinar el funcionamiento del ordenador, actuando entre este y las aplicaciones que están corriendo en ese momento, gestionando los diferentes recursos hardware que requieren estas aplicaciones (CPU, RAM, interrupciones, dispositivos de E/S..).

b **Desde el punto de vista de las aplicaciones**: una aplicación solo es un conjunto de programas escritos en algún lenguaje de programación. Hay multitud de lenguajes de programación, con la característica de que todos están escritos en un idioma que el ser humano puede entender. El hardware por otro lado solo es capaz de interpretar señales eléctricas que se traducen en secuencias de 0 y 1 (código binario). Para que estás aplicaciones puedan ejecutarse en el hardware debe darse un proceso de "traducción"que veremos mas adelante.

1.3 Desarrollo de Software

Entendemos por **Desarrollo de Software** todo el proceso desde que se concibe la idea hasta que el programa está implementado y funcionando en el ordenador. Aunque en principio pueda parecer una tarea simple, consta de una series de pasos de obligado cumplimiento, pues solo así podemos garantizar que las aplicaciones creadas son eficientes, seguras, fiables y responden a las necesidades de los usuarios finales.

Como veremos más adelante en la unidad, esta serie de pasos se le suele denominar **Etapas** en el desarrollo de software. Según el orden y la forma en la que se llevan a cabo estas etapas hablaremos de diferentes ciclos de vida del software.

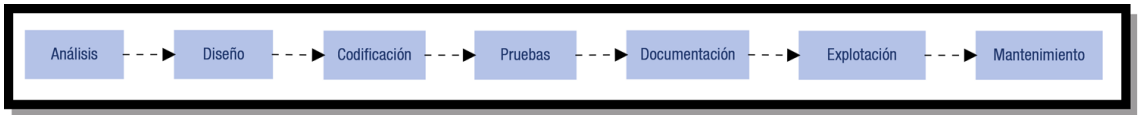

Figura 1.3.1.: Etapas del Desarrollo de Software

Cabe destacar que la construcción de software es un proceso muy complejo y que requiere de una gran coordinación y disciplina del grupo de trabajo que lo desarrolle.

1.3.1 Ciclos de Vida del Software

Ya hemos visto que para crear software debemos seguir un número de pasos conocidos como ciclo de vida del software. Más adelante en esta unidad veremos en que consiste cada paso mas detalladamente. Por ahora, vamos a centrarnos en ver los diferentes ciclos de vida del software que existen atendiendo a como se desarrollan dichos pasos.

Aunque podemos encontrar diferentes clasificaciones sobre los distintos tipos de ciclos de vida del software los mas conocidos y utilizados son los siguientes:

1. **Modelo en Cascada**: es el modelo de vida clásico del software. Actualmente es prácticamente imposible de utilizar, salvo para desarrollos muy pequeños, ya que es necesario conocer todos los requisitos con antelación y las etapas pasan de una a otra sin retorno, presuponiendo que no ha habido errores en la etapa anterior.
2. **Modelo en Cascada con Realimentación**: es uno de los modelos más utilizados. Proviene del modelo anterior, pero se introduce una realimentación entre etapas, permitiendo que podamos volver hacia atrás en cualquier momento para corregir, depurar o modificar algún aspecto. Es el modelo perfecto si el proyecto es es rígido (pocos cambios y poco evolutivo) y los requisitos están claros. No obstante, si se prevén muchos cambios durante el desarrollo no es el modelo más idóneo.
3. **Modelos Evolutivos**: son los modelos más modernos y tienen en cuenta el aspecto cambiante y evolutivo del software. Dentro de este modulo podemos encontrar dos variantes:

3.1. **Modelo Iterativo Incremental**: ésta basado en el modelo en cascada con realimentación, donde las fases se repiten y refinan, propagando su mejora a las fases siguientes. El proyecto se realiza en pequeñas porciones (**incrementa**) en sucesivas iteraciones (**sprints**) al final de las cuales se ve lo que se ha desarrollado, pudiendo hacer correcciones o modificaciones antes de la siguiente iteración o incluso añadir nuevos requerimientos (**adaptativo**). Cada sprint debe proporcional un resultado completo preparado para entregárselo al clientes.

3.2. **Modelo en Espiral**: es una combinación del modelo anterior con el modelo en cascada. En éste, el software se va construyendo repetidamente en forma de versiones que son cada vez mejores, debido a que se va incrementando la funcionalidad. Es un modelo muy complejo.

En la actualidad, la metodologías de desarrollo ágil, enmarcadas dentro de los modelos evolutivos, están en auge, y metodologías como **Scrum**, **TDD** o **BDD**, así como metodologías híbridas basadas en estas, se están convirtiendo en el estándar de la industria. [4].

1.3.2 Herramientas de Apoyo al Desarrollo De Software

En la práctica, vamos a contar con un conjunto de herramientas que nos van a facilitar llevar a cabo las diferentes etapas del ciclo de vida de desarrollo, automatizando las tareas, ganando con ello fiabilidad y tiempo, y permitiéndonos centrarnos en los requerimientos del sistema y el análisis del mismo.

Las herramientas **CASE** son un conjunto de aplicaciones que se usan en el desarrollo de software con el propósito de reducir costes y tiempo de proceso, aumentando la productividad. Estas herramientas pueden ayudarnos prácticamente en cualquier etapa del proceso de desarrollo.

El desarrollo rápido de aplicaciones o **RAD**, es un proceso de desarrollo de software que comprende el desarrollo iterativo, la construcción de prototipos y el uso de herramientas CASE. Hoy en día se usa para referirnos al desarrollo rápido de interfaces de usuario o entornos de desarrollo integrados (**IDE**) completos.

La tecnología CASE trata de automatizar el proceso de desarrollo para que mejore las calidad del proceso y el resultado final. En concreto, estas herramientas permiten:

- Mejorar la planificación del proyecto.
- Darle agilidad al proceso.
- Generar software mas reutilizable.
- Creación de aplicaciones más estandarizadas.
- Mejorar las tareas de mantenimiento de las aplicaciones.
- Permiten visualizar todo proceso de desarrollo de forma gráfica.

Las herramientas CASE se pueden clasificar según su funcionalidad o la función que desempeñan dentro del proceso de desarrollo.

Atendiendo a la función que desempeñan en cada fase del proceso, las herramientas CASE pueden ser:

- **U-CASE**: ofrecen ayuda en las fases de planificación y análisis de requisitos.
- **M-CASE**: ofrecen ayuda en el análisis y diseño.
- **L-CASE**: ofrecen ayuda en la programación del software, detección de errores en código, depuración de programas y pruebas, y en la generación de la documentación.

Si tenemos en cuenta su funcionalidad, se pueden diferenciar algunas como:

- Herramientas de generación semiautomáticas de código.
- Editores **UML**.
- Herramientas de refactorización de código.
- Herramientas de mantenimiento, como los sistemas de control de versiones.

Algunos ejemplos de herramientas **CASE libres** son: ArgoUML, Use Case Maker, ObjectBuilder,...

1.4 Lenguajes de Programación

Podemos definir un **lenguaje de programación** como un idioma creado artificialmente, que se compone de un conjunto de símbolos y normas que se aplican sobre un alfabeto para obtener un código que el hardware de la computadora pueda entender y procesar. Es el instrumento que tenemos para que el ordenador realice las tareas que necesitamos, dicho de otra forma.

Hay multitud de lenguajes cada uno con su estructura y símbolos. Además cada lenguaje está enfocado en la realización de tareas o áreas determinadas. Por ello, es muy importante la elección del lenguajes o lenguajes de programación en un proyecto.

Los lenguajes de programación han ido evolucionando a través de la historia, podemos ver esta evolución de forma simplificada en la siguientes figura:

Figura 1.4.1.: Evolución de los lenguajes de programación

Las principales características de estos lenguajes de son las siguientes:

- **Lenguajes Máquina**:
  - Sus instrucciones están compuestas por unos y ceros.
  - Es el único lenguaje que entiende el ordenador, no necesita traducción.
  - Fue el primer lenguaje utilizado.
  - Es único para cada procesador, es decir, no es portable de un equipo a otro.
  - Hoy en día nadie lo usa.
- **Lenguaje Ensamblador**:
  - Sustituyó el lenguaje máquina para facilitar la programación.
  - Se programa usando mnemotécnicos (instrucciones complejas).
  - Necesita traducción al lenguaje máquina para poder ejecutarse.
  - Sus instrucciones hacen referencia a la ubicación física de los archivos y registros.
  - Es difícil de utilizar.
- **Lenguajes de alto nivel**:

- Sustituyeron al ensamblador para facilitar la programación.
- Se utilizan sentencias y órdenes derivadas el inglés. Necesita traducción al lenguaje máquina.
- Son más cercanos al razonamiento humano.
- Son los más utilizados hoy en día.

- **Lenguajes Visuales**:
  - Están sustituyendo a los lenguajes de alto nivel basados en código.
  - Se programa gráficamente usando 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.

1.4.1 Concepto y Características

Como ya hemos dicho, un **lenguaje de programación** es un lenguaje formal que le proporciona a una persona, en este caso el programador, la capacidad de escribir una serie de **instrucciones** o secuencia de órdenes en forma de algoritmo con el fin de controlar el comportamiento lógico o físico de un sistema informático, de manera que se pueden obtener diferentes formas de dato o realizar tareas. [5]

Un lenguaje de programación esta compuesto por:

- **Alfabeto**: conjunto de símbolos permitidos.
- **Sintaxis**: normas de construcción para los símbolos del lenguaje.
- **Semántica**: significado de las construcciones para realizar acciones válidas.

Los lenguajes de programación se pueden clasificar de varias formas en base de distintas características:

- **Según lo cercano que este al lenguaje humano**:
  - **Lenguajes de Alto Nivel**: son lenguajes que por su esencia, son mas cercanos al razonamiento humano.
  - **Lenguajes de Bajo Nivel**: estos lenguajes están mas próximos al lenguaje que entiende el computador. Los principales son: **Ensamblador** y **Lenguaje Máquina**.
- **Según la Técnica de Programación**:
  - **Lenguajes de Programación Estructurados**: usan la técnica de programación estructurada. Ejemplos son Pascal, C, etc...
  - **Lenguajes de Programación Orientada a Objetos**: usan las técnica de programación orientada a objetos. Ejemplos son Java, Ruby, Ada, C++, etc...
  - **Lenguajes de Programación Funcional**: basan su técnica en el uso de verdaderas funciones matemáticas. Ejemplos son Haskell, Elixir, etc...
  - **Lenguajes de Programación Visual**: basados en las técnicas anteriores, permiten programar gráficamente, generando automáticamente el código correspondiente. Ejemplos son Visual Basic.Net, Borland, etc..

A pesar de la cantidad de lenguajes de programación que existen, Python, C, Java, C++, C#, Visual Basic y Javascript, concentran alrededor del 60 % del interés de la comunidad mundial de programadores [6], aunque en última instancia, la **elección del lenguaje de programación** para un proyecto dependerá de las características del problema a resolver.

1.4.2 Lenguajes de Programación Estructurados

Los lenguajes estructurados fueron los primeros lenguajes de alto nivel que surgieron, y a partir de ellos, se evolucionó a los diferentes tipos de lenguajes que tenemos hoy en día.

La **programación estructurada** se define como la técnica para escribir lenguajes de programación que solo permite el uso de tres estructuras de control:

- Sentencias secuencias.
- Sentencias selectivas (condicionales).
- Sentencias iterativas (bucles).

Los lenguajes de programación que se basan en este estilo de programación se conocen como **lenguajes de programación estructurados**.

La programación estructurada fue un gran éxito debido a su sencillez a la hora de construir y leer programas. Aunque como todo, tiene sus ventajas e inconvenientes, que son los siguientes:

- **Ventajas**:
  - Los programas son fáciles de leer, sencillos y rápidos.
  - El mantenimiento de los programas se simplifica.
  - La estructura del programa es sencilla y clara
- **Inconvenientes**:
  - Todo el programa se concentra en un único bloque, si el código crece mucho se hace complejo de manejar.
  - No permite la reutilización de código, ya que todo el programa va en el mismo bloque.

Debido a estos inconvenientes, la **programación estructurada** evolucionó hacia la **programación modular**, que divide el programa en trozos de código llamados módulos, con una funcionalidad concreta, y que permiten su reutilización en otras aplicaciones.

Algunos de los lenguajes estructurados mas usados son C, FORTRAN, Pascal, etc...

1.4.3 Lenguajes de Programación Orientados a Objetos

Los **lenguajes de programación orientados a objetos** tratan los programas, no como un conjunto de instrucciones secuenciales, sino como un **conjunto de objetos** independientes, que interactúan entre sí, y que son altamente reutilizables en otras aplicaciones.

Su **principal desventaja** es que no es un paradigma de programación tan intuitivo como la programación estructurada. A pesar de ello, alrededor del 55 % del software que se produce emplea esta técnica. Esto es debido principalmente a que el código es **muy reutilizable** y a la facilidad de **depuración**.

Las principales **características** de estos lenguajes son:

- Los objetos del programa tendrán una serie de atributos que los diferencias unos de otros.

- Se definen clases como una colección de objetos con características similares.
- Mediante los llamados métodos, los objetos se comunican con otros produciendo un cambio de estado.
- Los objetos son unidades independientes e indivisibles que forma la base de este paradigma de programación.

Algunos de los lenguajes orientados a objetos más utilizados son C++, Java, Ruby, Delphi, etc..

1.5 Fases del Desarrollo de Software

Como hemos visto en puntos anteriores, podemos elegir entre diferentes modelos de desarrollo de software, pero independientemente de cual elijamos, siempre hay un número de etapas que debemos seguir. Estas etapas son las siguientes:

1. **Análisis de Requisitos**: en esta etapa se especifican los requisitos funcionales y no funcionales del sistema.
2. **Diseño**: se divide el sistema en partes y se determina la función de cada una.
3. **Codificación**: se elige un lenguaje de programación y se codifican los programas.
4. **Pruebas**: se prueban los programas para detectar errores y depurarlos.
5. **Documentación**: se documentan y guarda información de todas las etapas.
6. **Explotación**: instalamos, configuramos y ejecutamos la aplicación en los equipos del clientes.
7. **Mantenimiento**: se mantiene el contacto con el cliente para actualizar o modificar la aplicación.

En los siguientes puntos vamos a ver estas etapas con más detalle.

1.5.1 Análisis

Es la primera fase del proyecto y una de las de mayor importancia, ya que todo lo demás dependerá de lo bien detallada que este esta fase. También es una de las más complicadas, ya que no esta automatizada y dependerá en gran medida del análisis que hagamos del problema a resolver.

En esta fase se especifican los requisitos funcionales y no funcionales de la aplicación. Estos **requisitos** consisten en:

1. **Requisitos funcionales**: estos requisitos especifican que tendrá que realizar la aplicación, que respuesta dará ante todas las entradas, como se comportará en situaciones inesperadas, etc...
2. **Requisitos no funcionales**: estos especifican requisitos como el tiempo de respuesta, legislación aplicable, etc...

Es de vital importancia una **buena comunicación** entre el **analista** y el **cliente** para que los requisitos se puedan especificar con el máximo detalle y adecuados a los deseos del cliente.

La culminación de esta fase es un **documento ERS** (Especificación de Requisitos del Sistema) donde deben quedar especificado lo siguiente:

- La planificación de las reuniones que van a tener lugar.
- Relación de los objetivos del cliente y el sistema.
- Relación de los objetivos funcionales y no funcionales del sistema.

- Relación de objetivos prioritarios y temporalización.
- Reconocimiento de requisitos mal planteados, etc...

Una vez realizado este documento y con los requisitos especificados y detallados, pasaremos a la siguiente fase, la fase de diseño.

1.5.2 Diseño

En esta segunda fase, el sistema debe dividirse en diferentes partes y establecer que relación habrá entre ellas. Decidir exactamente que hará cada parte. En definitiva, debemos crear un modelo funcional-estructural de los requerimientos del sistema global, para poder dividirlo y afrontarlo por separado.

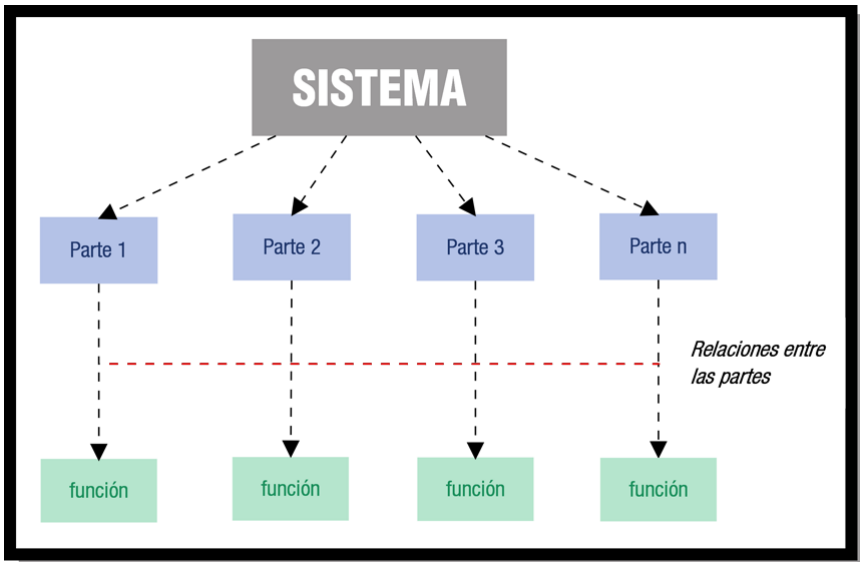

Figura 1.5.1.: División de la aplicación en partes

Además, en esta etapa se deben tomar decisiones que van a afectar el desarrollo del software, como:

- Entidades y relaciones de la base de datos.
- Selección del lenguaje de programación que se va a utilizar.
- Selección del SGBD.
- Definición de diagrama de clases.
- Definición de diagrama de colaboración.
- Definición del diagrama de paso de mensajes.

1.5.3 Codificación

En esta fase, con el lenguajes de programación elegido, codificar toda la información anterior y llevarla a código fuente. Esta tarea la realiza el **programador** y tiene que cumplir con todos los requisitos y restricciones impuestos en la fase de análisis y diseño de la aplicación.

Ademas, el **código** deberá cumplir con las siguientes **características**:

- **Modularidad**: que este divido en trozos pequeños funcionales.

- **Corrección**: que haga lo que se pide realmente.
- **Fácil de leer**: que sea fácil de leer y comprender para facilitar su desarrollo y mantenimiento.
- **Eficiencia**: que haga un buen uso de los recursos.
- **Portabilidad**: que se pueda ejecutar en cualquier equipo.

El código pasará por un número de fases desde su implementación hasta que se pueda ejecutar. Estas fases son las siguientes:

1. **Código Fuente**: es el código escrito por los programadores usando algún edito o IDE. Se escribe usando algún lenguaje de programación de alto nivel y contiene el conjunto de instrucciones necesaria.
2. **Código Objeto**: es el código binario resultado de compilar el código fuente. La **compilación** es la traducción de una sola vez del programa y se realiza utilizando un compilador. La **traducción** es la interpretación y ejecución simultánea de un programa línea a línea.
3. **Código Ejecutable**: es el resultado de enlazar el código objeto con ciertas **rutina** y **biblioteca** necesarias. El sistema operativo se encarga de cargar el programa en la memoria RAM y proceder a ejecutarlo.

En el **anexo A.2** también podemos consultar información sobre las máquinas virtuales, que son empleadas en esta etapa del desarrollo con diferentes propósitos.

1.5.3.1 Código Fuente

El código fuente es el conjunto de instrucciones que la computadora debe realizar, escritas por un programador en algún lenguaje de programación. Este conjunto de instrucciones no es directamente interpretable por la máquina, sino que deberá de pasar por un proceso de traducción para que pueda ser ejecutado.

Una parte importante de esta fase es la **elaboración de un algoritmo**, que definimos como un conjunto de pasos a seguir para obtener la solución a un problema. El algoritmo los diseñamos en pseudocódigo, y con el la codificación a un lenguaje de programación determinado será más rápida y directa.

Para obtener el código fuente de una aplicación se deben seguir estos pasos:

1. Se debe partir de la etapa anterior de análisis y diseño.
2. Se diseñara un algoritmo que simbolice los pasos a seguir para solucionar el problema.
3. Se elegirá un lenguaje de algo nivel apropiado para las características del software que se quiere codificar.
4. Se procederá a codificación del algoritmo diseñado.

La culminación de este proceso es la obtención de un documento con todos los **módulos**, **funciones**, **bibliotecas** y **procedimientos** necesarios para codificar la aplicación. Puesto que el código es inteligible por la máquina habrá que traducirlo, obteniendo así un código equivalente pero ya entendible por el ordenador.

También hay que tener en cuenta, en este punto, bajo que **licencia** vamos a crear el código, siendo las dos más comunes:

- **Licencias de Código Fuente Abierto**: son licencias que permiten que cualquier usuario o programador pueda estudiar el código, modificarlo y reutilizarlo.

- **Licencias de Código Fuente Cerrado**: estas licencias no dan permisos para editar ni modificar el código fuente.

1.5.3.2 Código Objeto

El código objeto es un código intermedio, traducido a unos y ceros, pero que aún no puede ser ejecutado directamente. Consiste en **bytecode** que esta distribuido en varios archivos, según la cantidad de módulos de software que compongan la aplicación. Este código se **debe generar** cuando el **código fuente** este **libre de errores** sintácticos y semánticos.

El proceso de transformación del código fuente a código puede realizarse de **dos formas**:

- **Compilación**: el proceso de traducción se realiza sobre todo el programa de una vez. El software que realiza esta operación se llama **compilador**.
- **Traducción**: el proceso de traducción del código fuente se realiza línea y línea y se ejecuta simultáneamente. No existe código objeto intermedio. El software utilizado se denomina **intérprete**. El proceso de traducción es más lento que el de compilación.

1.5.3.3 Ejecutable

El código ejecutable es el que obtenemos al enlazar el código objeto con las librerías y rutinas necesarias, resultando un único archivo que puede ser directamente ejecutado por la computadora. Este archivo es ejecutado y controlado por el sistema operativo.

Para obtener un solo archivo ejecutable habrá que enlazar todos los archivos de código objeto mediante una aplicación denominada **linker** (enlazador) y obtener así un único archivo que ya si es ejecutable directamente por la computadora.

En la siguiente figura vemos el proceso completo para generar archivos ejecutables.

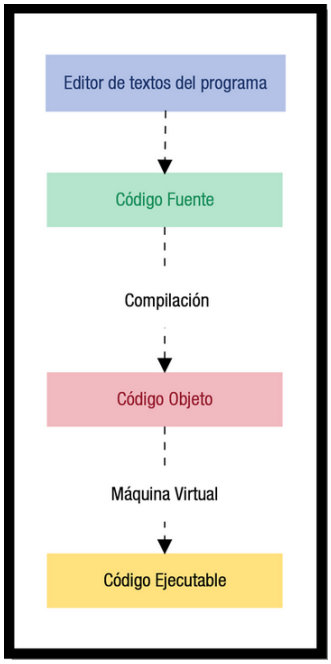

Figura 1.5.2.: Generación de código ejecutable

1.5.4 Pruebas

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

La **realización de pruebas** es una fase **imprescindible** para asegurar la **validación** y **verificación** del software desarrollado y podemos clasificarlas en dos tipos:

- **Pruebas Unitarias**: consiste en probar una a una las diferentes partes del software y comprobar su funcionamiento de forma individual. Un ejemplo de librerías para realizar este tipo de pruebas es **JUnit** para el lenguaje Java o **Jest** para Javascript.
- **Pruebas de Integración**: se realiza una vez que se han realizado las pruebas unitarias y consistirán en probar el funcionamiento del sistema completo, con todas sus partes interrelacionadas.

La prueba final, también conocida como **Beta Test** se realizará sobre el entorno de producción del cliente.

1.5.5 Documentación

Para proporcionar toda la información posible sobre el uso del software, tanto a los usuarios como a otros desarrolladores, es necesario crear la **documentación** de la aplicación.

Debemos ir documentando cada fase del proyecto, ademas, para ir pasando de una otra de forma clara y definida. Una correcta documentación facilitará la reutilización de parte del software para futuros proyectos así como el mantenimiento y su utilización por otros desarrolladores. En la siguiente figura se muestra una tabla con los diferentes tipos de documentación que se pueden generar.

Documentos a elaborar en el proceso de desarrollo de software

	GUÍA TECNICA	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.

Figura 1.5.3.: Tipos de documentación en el desarrollo de software

1.5.6 Explotación

La explotación es la fase en la que los usuarios finales conocen la aplicación y comienzan a utilizarla. Consiste en la instalación, puesta a punto y funcionamiento de la aplicación en el sistema del cliente. Los pasos que se siguen son los siguientes:

- En el proceso de **instalación**, los programas son transferidos al ordenador del cliente, o a un servidor, y posteriormente verificados y configurados. Es importante que el cliente este presente en esta fase para ir comentándole como se realiza la instalación. En este momento, es cuando se suelen realizar las Beta Test.
- Ahora pasamos a la fase de **configuración**, en la que se asignan los parámetros de funcionamiento de la empresa a la aplicación y comprobamos que es operativa. También es probable que la configuración la realicen los usuarios finales, si le hemos proporcionado una guía y de instalación. Si la aplicación es lo suficientemente simple, se puede programar la configuración para que se realice de forma automática.
- Una vez configurado, se pasa a la fase de **producción**, donde la aplicación pasa a manos del cliente y se da comienzo a la explotación del software.

Es muy importante tener todo preparado antes de presentar el producto final al cliente, ya que esta es una de las fases mas críticas del proyecto.

1.5.7 Mantenimiento

El software es cambiante y deberá actualizarse y evolucionar con el tiempo. Deberá adaptarse a las mejoras de hardware del mercado y afrontar situaciones que no estaban prevista cuando se creó. Además, siempre pueden surgir errores que deberán irse corrigiendo y nuevas versiones del producto mejores que la anterior. Por todo ello se pacta un servicio de mantenimiento del sistema con el cliente, que tendrá un coste temporal y económico.

El **mantenimiento** se define como el proceso de control, mejora y optimización del software. Su duración es la mayor de todo el ciclo de vida del software, ya que también comprende las actualizaciones y evoluciones futuras del mismo.

Los tipos de cambios que se pueden realizar en una aplicación son los siguientes:

- **Perfectivos**: se realizan para mejorar la funcionalidad del software.
- **Evolutivos**: en el futuro el cliente tendrá nuevas necesidades, por lo que se requerirán nuevas modificaciones, expansiones o eliminación de código.
- **Adaptativos**: modificación y actualizaciones para adaptarse a las nuevas tendencias del mercado, como nuevo hardware, nuevos frameworks, etc...
- **Correctivos**: son los que se realizan para la corrección de errores.

Si quieres ampliar información sobre las fases de desarrollo, en este documento se detallan los diferentes roles que podemos encontrar en el proceso de desarrollo y que funciones tienen dentro de este, así como sus interacciones.

1.6 Entornos de Desarrollo

En esta sección vamos a hablar de los entornos de desarrollo, herramientas que nos en la fase de codificación a la hora de programar nuestras aplicaciones.

En muchas ocasiones, las personas que empiezan a programar lo hacen usando un **editor simple**, un **compilador** y un **depurador**. Pero la mayoría de gente acaba usando un algún entorno de desarrollo integrado o IDE (Integrated Development Environment) para crear las aplicaciones.

Un **entorno de desarrollo integrado** (IDE), es un tipo de software compuesto por un conjuntos de herramientas que nos facilitan el proceso de programación. Estas herramientas suelen ser:

- **Editor** de código de programación.
- **Compilador**.
- **Intérprete**.
- **Depurador**.
- **Constructor de interfaz** gráfico.

Los primeros IDEs nacieron a principios de los años 70, pero fue en la década de los 90 cuando se popularizaron. Su objetivo principal es ganar fiabilidad y tiempo en los proyectos de software, proporcionando una serie de componentes bajo una misma interfaz gráfica, con la consiguiente comodidad, aumento de eficiencia y reducción del tiempo de codificación.

En un primer momento los IDEs estaban enfocados en un lenguaje de programación concreto, pero actualmente suelen soportar diferentes lenguajes de programación, algunos de forma predeterminada y otros mediante la instalación de plugins. Algunos ejemplos son Eclipse, Netbeans, VS Code, Intellij Idea, etc..

En este tema nos vamos a centrar en la instalación de **Netbeans** en Ubuntu y veremos como se configurar y se generan ejecutables, haciendo uso de sus componentes y herramientas.

1.6.1 Evolución Histórica

En la época de utilización de las **tarjetas perforadas** como sistema de almacenamiento los **Entornos de Desarrollo Integrado** sencillamente no tenían sentido. Los programas estaban escritos con **diagramas de flujo** y entraban al sistema a través de estas tarjetas para posteriormente compilarse.

El primer lenguaje de programación que utilizó un IDE fue BASIC, que fue el primero en abandonar las tarjetas perforadas. Este primer IDE estaba basado exclusivamente en la consola de comandos. Sin embargo, el uso que hacía de la gestión de archivos, compilación y depuración era totalmente compatible con la definición de IDE actual.

A nivel popular, el primer IDE puede considerarse que fue el IDE **Maestro**. Nació a principios de los años 70 y fue instalado por unos 22.000 programadores en todo el mundo, liderando este campo durante los años 70s y 80s.

El uso de los IDEs se popularizó en los años 90 y hoy en día contamos con una infinidad de IDEs, tanto de licencia libre como propietaria.

Tipos de entornos de desarrollo más relevantes en la actualidad.

Entorno de desarrollo	Lenguajes que soporta	Tipo de licencia
NetBeans.	C/C++, Java, JavaScript, PHP, Python.	De uso público.
Eclipse.	Ada, C/C++, Java, JavaScript, PHP.	De uso público.
Microsoft Visual Studio.	Basic, C/C++, C#.	Propietario.
C++ Builder.	C/C++.	Propietario.
JBuilder.	Java.	Propietario.

Figura 1.6.1.: Tipos de IDE actuales

1.6.2 Funciones de un Entorno de Desarrollo

En los puntos anteriores ya hemos visto los componente que tiene un IDE, como editor de textos, compilador, depurador, etc... En este punto vamos a ver sus funciones de estos componentes, que se podrían resumir en las siguientes:

- **Editor de código**: resaltado y coloreado de la sintaxis.
- **Autocompletado** de código, atributos, métodos, clases, etc...
- Identificación automática de código.
- Herramientas de **concepción visual** para crear y manipular componentes visuales.
- **Asistentes** y utilidades de gestión y **generación de código**.
- Archivos fuentes en unas carpetas y compilación en otras. (estructuración del código)
- **Compilación** de proyectos complejos en **un solo paso**.
- **Control de versiones**: tener un único almacén de archivos compartido por todos los desarrolladores de un proyectos con mecanismos de seguimiento de cambios, recuperación a un estado anterior, etc...
- Soporte **cambios** de varios usuarios de forma **simultánea**.
- **Generación de documentación** automática.
- **Detección** de errores de **sintaxis** en tiempo real.
- **Refactorización de código**: cambios menores en el código que aumentan la legibilidad sin alterar la funcionalidad.
- **Indentación** automática del código para aumentar la legibilidad.
- **Depuración**: seguimiento de variables, puntos de ruptura y mensajes de error del intérprete.
- Instalación de **plugins** y módulos que expanden la funcionalidad del IDE.
- **Administración** de las **interfaces de usuario**.
- **Administración** de las **cuentas de usuario**.

Como podemos ver en esta larga lista, los IDEs tiene un montón de funcionalidades que nos va a facilitar el proceso de codificación en muchos aspectos. Hoy en día es imprescindible su utilización para mejorar la productividad y centrarlos en la programación, dejando al IDE a cargo de otros aspectos más “secundarios”.

1.6.3 Estructura de un Entorno de Desarrollo

Como ya hemos comentado, un IDE esta formado por un conjunto de componentes software que determinan sus funciones. Los componente principales que podemos encontrar en un IDE, son los siguientes:

- **Editor de Textos**: resalta y colorea la sintaxis, tiene funciones para autocompletar el código, ayuda y listado de funciones y parámetros, inserción automática de paréntesis o corchetes, tabulaciones y espacios, etc...
- **Compilador/Intérprete**: detección de errores de sintaxis en tiempo real y funciones para la refactoriación.

- **Depurador**: botón de ejecución y traza, puntos de ruptura y seguimiento de variables. Opción de depurar en servidores remotos.
- **Generador automático de herramientas**: para la generación y manipulación de componentes visuales y todo un arsenal de asisten y utilidades de gestión y generación de código.
- **Interfaz Gráfica**: es la interfaz principal del IDE y nos permite usar todas las funcionalidades y utilizar diferentes lenguajes de programación en un mismo IDE, permitiendo el acceso a innumerables bibliotecas y plugins, aumentado las opciones de nuestro programa.

1.6.4 Entornos Integrados Libres y Propietarios

Según la licencia que usen los entornos de desarrollo, podemos encontrarnos entornos de desarrollo **libres** y entornos de desarrollo **propietarios**. La elección de la licencia del IDE en un proyecto es de vital importancia.

- **Entornos de Desarrollo Libres**

  Son aquellos que usan licencias open source, que permiten que el código del IDE este disponible para cualquier persona. Pueden ser gratuitos. En la siguiente figura se muestran los mas utilizados, aunque falta alguna opción como **Visual Studio Code** (no confundir con Microsoft Visual Studio, que es la opción propietaria de MS).

Tipos de entornos de desarrollo libres más relevantes en la actualidad.

IDE	Lenguajes que soporta	Sistema Operativo
NetBeans.	C/C++, Java, JavaScript, PHP, Python.	Windows, Linux, Mac OS X.
Eclipse.	Ada, C/C++, Java, JavaScript, PHP.	Windows, Linux, Mac OS X.
Gambas.	Basic.	Linux.
Anjuta.	C/C++, Python, Javascript.	Linux.
Geany.	C/C++, Java.	Windows, Linux, Mac OS X.
GNAT Studio.	Fortran.	Windows, Linux, Mac OS X.

Figura 1.6.2.: IDEs con licencia libre

- **Entornos de Desarrollo Propietarios**

  Son aquellos cuya licencia es propietaria, por lo tanto el código fuente no estará disponible para los usuarios. Se necesita pagar por su uso. En la siguiente figura se muestran algunos ejemplos.

Tipos de entornos de desarrollo propietarios más relevantes en la actualidad.

IDE	Lenguajes que soporta	Sistema Operativo
Microsoft Visual Studio.	Basic, C/C++, C#.	Windows.
FlashBuilder.	ActionScript.	Windows, Mac OS X.
C++ Builder.	C/C++.	Windows.
Turbo C++ profesional.	C/C++.	Windows.
JBuilder.	Java.	Windows, Linux, Mac OS X.
JCreator.	Java.	Windows.
Xcode.	C/C++, Java.	Mac OS X.

Figura 1.6.3.: Entornos de desarrollo propietarios

La elección del entorno de desarrollo así como de su licencia deberá ajustarse al lenguaje de programación que vamos a usar, así como a los requisitos del proyecto.

1.7 Conclusiones

En este tema hemos visto el ciclo de vida de software y la importancia que tiene en el desarrollo de aplicaciones de forma metódica. Algunas etapas tienen más importancia que otras, por ejemplo, si la primera etapa, **análisis**, no se realiza de forma meticulosa, puede lastrar el resto de etapas y perjudicar el proceso de desarrollo. Por ello, es de suma importancia dedicarle el tiempo necesario a cada etapa. Aunque hay diferentes metodologías de desarrollo, todos tienen las mismas etapas.En la actualidad se están imponiendo **metodologías de desarrollo ágil**.

Si queremos obtener más información, podemos consultar este enlace, donde se nos hace un resumen de las principales metodologías de desarrollo actuales.

Por otro lado hemos visto que es un entorno de desarrollo integrado y como su utilización nos facilita enormemente la programación. La elección que se haga de un entorno u otro dependerá de las preferencias de cada uno, aunque también de los requisitos del proyecto y del lenguaje de programación que vayamos a utilizar. Lo que está claro es que son una herramienta indispensable para cualquier programador.

A. Anexos Tema 1

A.1 Secuencias de Control en Programación Estructurada

En este punto vamos a hacer un breve repaso de las sentencias básicas de control en los lenguajes de programación estructurados.

- **Sentencias Secuenciales**

  Son aquellas sentencias que se ejecutan una detrás de otra.

```
printf ("declaración de variables");
int numero_entero;
espacio=espacio_inicio + veloc*tiempo;
```

Figura A.1.1.: Sentencias secuenciales en C

- **Sentencias Selectivas** (Condicionales)

  Son aquellas sentencias en las que se evalúa una condición, ejecutándose una serie de instrucción en el caso de que la condición sea verdadera y otras en el caso de que sea falsa. Si estructura es la siguiente:

```
if (condición) then
  instrucciones a ejecutar si es verdadera
else
  instrucciones a ejecutar si es falsa
```

Figura A.1.2.: Estructura sentencia condicional if

En la siguiente figura podemos ver un ejemplo básico en el lenguaje de programación C.

```
if (a >= b)
   c = a - b;
else
   c = a + b;
```

Figura A.1.3.: Sentencia if en el lenguaje C

- **Sentencias Repetitivas** (iteraciones o bucles)

Un bucle es una serie de acciones que se ejecutan mientras que se cumpla una condición. Su estructura básica es la siguiente.

```
// Bucle whle ... do

while (condición)
   instrucciones a repetir
done

// Bucle for

for (condición_inicial; incremento; condición_de_parada)
   instrucciones a repetir
```

Figura A.1.4.: Documento SGML simple

En la siguiente figura vemos un ejemplo de un while...do en C, en este lenguaje, la palabra done se omite.

```
int num;
num = 0;

while (num<=10) {
   printf("Repetición numero %d", num);
   num = num + 1;
};
```

Figura A.1.5.: Documento SGML simple

A.2 Máquinas Virtuales

Una máquina virtual es un tipo de software que separa el funcionamiento del ordenador de los componente hardware. Esta capa de software desempeña un papel muy importante en el funcionamiento de los lenguajes de programación, tanto compilados como interpretados.

Con el uso de máquinas virtuales podremos desarrollar y ejecutar código en cualquier equipo, independientemente de las características concretas de los componente físicos de este, garantizando la **portabilidad** de las aplicaciones.

Las **funciones** principales de una máquina virtual son las siguientes:

- Conseguir que las aplicaciones sean portables.
- Reservar memoria para los objetos que se crean y liberar la memoria no utilizada.
- Comunicarse con el sistema donde se instala la aplicación (host) para el control de los recursos hardware implicados en el proceso.
- Cumplimento de las normas de seguridad de las aplicaciones.

Las principales **características** de estás, son las siguientes:

- La máquina virtual **aísla la aplicación** de los detalles físicos en del equipo en cuestión. Cuando el código se compila se obtiene **bytecode**. La máquina virtual funciona como una capa de software

de bajo nivel y actual de puente entre el bytecode de la aplicación y los dispositivos físicos del sistema.

- La máquina virtual **verifica** todo el bytecode antes de ejecutarlo.
- También se encarga de **proteger** las **direcciones de memoria**.

A.2.1 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 de software donde están definidos bibliotecas, programas de soporte, lenguaje interpretado, etc...

Con el uso de frameworks podemos pasar más tiempo analizando los requisitos del sistema y las especificaciones técnicas de nuestra aplicación, ya que la tarea laboriosa de los detalles de programación queda bastante resuelta. Pero como todo, tiene sus ventajas e inconvenientes.

- **Ventajas** de usar un framework:
  - **Desarrollo rápido** de software.
  - **Reutilización** de partes de código para otras aplicaciones.
  - **Diseño uniforme** del software.
  - **Portabilidad** de las aplicaciones de un computador a otro, ya que los bytecode que se generan podrán ser ejecutados en cualquier sobre cualquier máquina virtual.
- **Inconvenientes** de usar un frameworks:
  - **Gran dependencia** del código respecto al framework utilizado.
  - La instalación y configuración del framework en nuestro equipo **consume** bastantes **recursos** del sistema.

Algunos ejemplos de frameworks son:

- **.NET**: framework para el desarrollo de aplicaciones sobre Windows. Ofrece el "Visual Studio .NET" que nos proporciona facilidades para construir aplicaciones y su motor es el ".NET Framework", que permite ejecutar dichas aplicaciones.
- **Java Spring**: son un conjunto de bibliotecas (API's) para el desarrollo y ejecución de aplicaciones. Es uno de los más empleados en Java y es recomendable saber manejarlo.

  Si quieres profundizar más en este framework, puedes consultar los siguientes enlaces:

  - Instalación de Spring e Información Básica
  - Desarrollando una aplicación Spring Paso a Paso

A.2.2 Entornos de Ejecución

Un entorno de ejecución es un servicio de una máquina virtual que sirve como base de software para la ejecución de aplicaciones. En ocasiones pertenece al propio sistema operativo, pero se puede instalar de forma independiente que funcionará por debajo del programa. Se denomina **runtime** al tiempo que tarda un programa en ejecutarse en la computadora.

Durante la ejecución un entorno se encargará de:

- Configurar la memoria principal disponible del sistema
- Enlazar los archivos del programa con las bibliotecas existentes y subprogramas creados.
- Depurar los programas: comprobar la existencia de errores semánticos del lenguaje.

Glosario

biblioteca Conjunto de subprogramas que sirven para desarrollar componentes software o que actúan como interfaz de comunicación entre componentes software. 13, 23

CASE Computer Aided Software Engineerig. 7, 23

Desarrollo de Software Conjunto de procesos desde que nace una idea hasta que se convierte en software. 6, 23

ERP Enterprise Resource Planning o Sistema de Planificación de Recursos Empresariales, son programas que se usan para la gestión empresarial y que se hacen cargo de distintas operaciones internas, desde la producción, la distribución o incluso los recursos humanos. 5, 23

IDE Integrated Development Environment. 7, 23

Linux Sistema Operativo tipo Unix compuesto por software libre y de código abierto que sirve como base para multitud de distribuciones como Debian, Slackware, Ubuntu, Gentoo.... 4, 23

macOS Sistema Opertativo basado en Unix, más concretamente en FreeBSD, y que está desarrollado y comercializado por Apple desde 2001. 4, 23

Microsoft Windows Sistema Operativo desarrollado por Microsoft que actualmente se encuentra en la versión 11. 4, 23

RAD Rapid Application Development. 7, 23

rutina Secuencia invariable de instrucciones que forman parte de un programa y que son reutilizables. 13, 23

tarjetas perforadas Tarjeta que almacenaba información que era leída por un lector específico. 17, 23

UML Unified Modeling Language. 8, 23

Bibliografía

[1] Wikipedia - Hardware. https://es.wikipedia.org/wiki/Hardware.

[2] Wikipedia - Software. https://es.wikipedia.org/wiki/Software.

[3] Joaquin Marquéz y Sergio Ernesto Gastón Perez. Tipos de software. https://es.slideshare.net/susahhreyyhha/tipos-de-software-15979630.

[4] Amna Batool. A survey of key challenges of agile in global software development: A case study with malaysia perspective. *International Journal of Industrial and Manufacturing Engineering, Vol13, nº10.*

[5] Wikipedia - Lenguaje de programación. https://es.wikipedia.org/wiki/Lenguaje_de_programaci%C3%B3n.

[6] Indice TIOBE. https://www.tiobe.com/tiobe-index/.