UD1. DESARROLLO DEL SOFTWARE

1. Programa informático

Un **programa informático** es un conjunto de instrucciones que una vez ejecutadas realizarán una o varias tareas en un ordenador. Al conjunto general de programas, se le denomina software.

Las instrucciones se escriben en un lenguaje de programación y son traducidas al único idioma que la máquina comprende, combinaciones de ceros y unos llamada código máquina.

La interacción con el sistema no siempre es una relación directa, no todos los programas tienen acceso libre y directo al hardware, es por lo que se definen los programas en dos clasificaciones generales: software del sistema y software de aplicación. Es el software del sistema el que se encarga de controlar y gestionar el hardware, así como de gestionar el software de aplicación, de hecho, en un software del sistema, como el sistema operativo, es en donde se ejecuta realmente el software de aplicación. Será el software de aplicación el que incorpore las librerías necesarias para entenderse con el sistema operativo, y este a su vez sería el que comunicase con el hardware.

2. Obtención de código ejecutable

Nuestro programa independientemente en que lenguaje esté si se quiere ejecutar en la arquitectura que sea, necesita ser traducido para poder ser ejecutado (con la excepción del lenguaje máquina). Por lo que, aunque tengamos el código de nuestro programa escrito en el lenguaje de programación escogido, no podrá ser ejecutado a menos que lo traduzcamos a un idioma que nuestra máquina entienda.

2.1. Tipos de código (fuente, objeto y ejecutable)

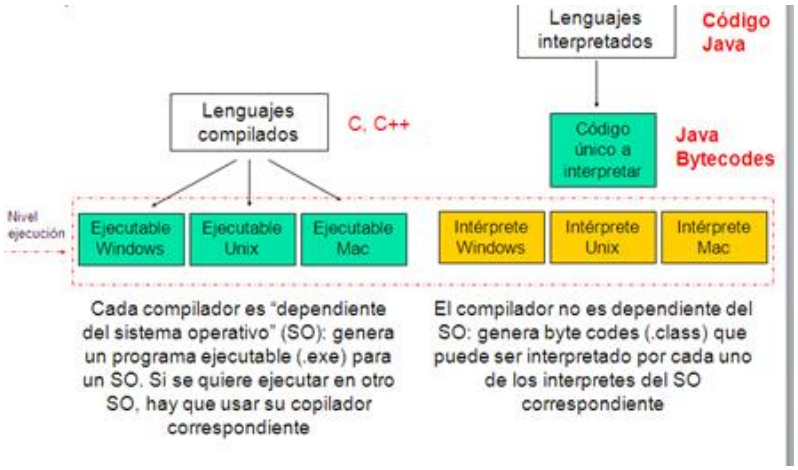
El código de nuestro programa es manejado mediante programas externos comúnmente asociados al lenguaje de programación en el que está escrito nuestro programa antes de ser ejecutado por el sistema.

Para ello, debemos definir los distintos **tipos de código** por los que pasará nuestro programa antes de ser ejecutado por el sistema.

- **Código fuente**: el código fuente de un programa informático es un conjunto de instrucciones escritas en un lenguaje de programación determinado. Es decir, es el código en el que nosotros escribimos nuestro programa en la maquina.
- **Código objeto**: el código objeto es el código resultante de compilar el código fuente. Si se trata de un lenguaje de programación compilado, el código objeto será código máquina, mientras que si se trata de un lenguaje de programación virtual, será código bytecode, código intermedio (cercano a código máquina).
- **Código ejecutable**: el código ejecutable es el resultado obtenido de enlazar nuestro código objeto con las librerías. Este código ya es nuestro programa ejecutable, programa que se ejecutará directamente en nuestro sistema o sobre una máquina virtual en el caso de los lenguajes de programación virtuales.

Cabe destacar que, si nos encontramos programando en un lenguaje de programación interpretado, nuestro programa no pasaría por el compilador y el enlazador, sino que solo tendríamos un código fuente que pasaría por un intérprete interno del sistema operativo o de la máquina que realizaría la interpretación y la ejecución línea a línea en tiempo real.

- **Proceso de compilación de un programa**



Tradicionalmente se han dividido los lenguajes en compilados e interpretados. Los primeros necesitan ser traducidos por un programa llamado compilador que convierte de código fuente a código objeto y otro programa enlazador que unirá el código objeto del programa con el código objeto de las librerías necesarias para producir el programa ejecutable. Como ejemplo de estos lenguajes podríamos citar a C, C++, Visual Basic, Clipper,etc. Los interpretados, en cambio, son traducidos mientras se ejecutan, sin que se genere código objeto, por ejemplo HTML, WML o XML, por lo cual no necesitan ser compilados.

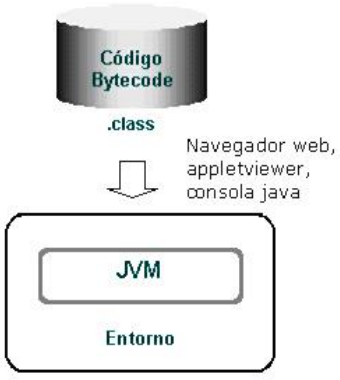
Así pues la diferencia entre estos lenguajes radica en la manera de ejecutarlos. Mientras que los compilados sólo se compilan una vez y lo hacen pasando todo el programa a código máquina (si da un error aunque sea en la última línea no podríamos ejecutar nada de nada), en el momento que lo hemos compilado correctamente y linkado se genera un archivo .exe que se puede ejecutar tantas veces como queramos sin tener que volver a compilar. Los interpretados en

cambio, cada vez que los queramos ejecutar tendremos que interpretarlos línea a línea, es más lento, pero puede ocurrir un error en la última línea y a diferencia de los compilados, el programa se ejecuta justo hasta la línea que produce el error.

Los actuales lenguajes virtuales tienen un funcionamiento muy similar al de los lenguajes compilados, pero, a diferencia de éstos, no es código objeto lo que genera el compilador, sino un bytecode que puede ser interpretado por cualquier arquitectura que tenga la máquina virtual que se encargará de interpretar el código bytecode generado para ejecutarlo en la máquina; aunque de ejecución lenta (como los interpretados), tienen la ventaja de poder ser multisistema y así un mismo código bytecode sería valido para cualquier máquina.

Java está diseñado para que un programa escrito en este lenguaje sea ejecutado independientemente de la plataforma (hardware, software y sistema operativo) en la que se esté actuando. Esta portabilidad se consigue haciendo de Java un lenguaje medio compilado medio interpretado. Se coge el código fuente, se compila a un lenguaje intermedio cercano al lenguaje máquina pero independiente del ordenador y el sistema operativo en que se ejecuta (llamado en el mundo Java bytecodes).

Finalmente, se interpreta ese lenguaje intermedio por medio de un programa denominado máquina virtual de Java (JVM), que sí depende de la plataforma.



Los java bytecodes permiten el ya conocido "write once, run anywhere" (compila una sola vez y ejecútalo donde quieras). Podemos compilar nuestros programas a bytecodes en cualquier plataforma que tenga el compilador Java. Los bytecodes luego pueden ejecutarse en cualquier implementación de la máquina virtual de Java (JVM). Esto significa que mientras el ordenador tenga un JVM, el mismo

programa escrito en Java puede ejecutarse en Windows, Solaris, iMac, Linux, etc.

La plataforma Java tiene 2 componentes:

- La máquina virtual de Java (JVM)
- El Java API (Application Programming Interface)

El Java API es una gran colección de componentes de software que proporcionan muchas utilidades para el programador

2.2. Compilación

Aunque el proceso de obtener nuestro código ejecutable pase tanto por un compilador como por un enlazador, se suele llamar al proceso completo "compilación".

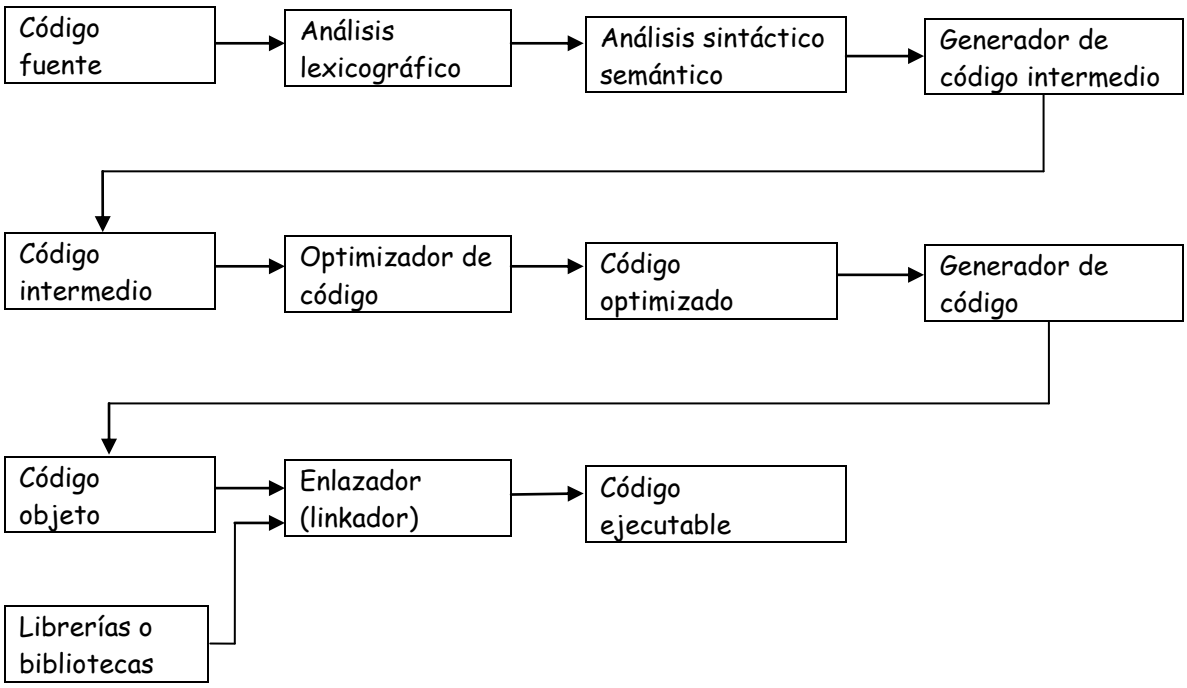
Todo este proceso se lleva a cabo mediante dos programas: el compilador y el enlazador. Mientras que el enlazador solamente une el código objeto con las librerías, el trabajo del compilador es mucho más completo.

Fases de compilación

- **Análisis lexicográfico**: se leen de manera secuencial todos los caracteres de nuestro código fuente, buscando palabras reservadas, operaciones, caracteres de puntuación y agrupándolos todos en cadenas de caracteres que se denominan lexemas, eliminando a su vez toda información superflua como pueden ser los comentarios y espacios en blanco no significativos del programa fuente.
- **Análisis sintáctico-semántico**: agrupa todos los componentes léxicos estudiados en el análisis anterior en forma de frases gramaticales. Con el resultado del proceso del análisis sintáctico, se revisa la coherencia de las frases gramaticales, si su "significado" es correcto, si los tipos de datos son correctos, si los arrays tienen el tamaño y tipo adecuados, y así consecutivamente con todas las reglas semánticas de nuestro lenguaje.
- **Generación de código intermedio**: una vez finalizado el análisis, se genera una representación intermedia a modo de pseudoensamblador con el objetivo de facilitar la tarea de traducir al código objeto. Es conveniente tener en cuenta que el código intermedio está enfocado a que un programa en dicho código pueda ser, con algunas modificaciones, entendido por cualquier procesador.
- **Optimización de código**: revisa el código pseudoensamblador generado en el paso anterior optimizándolo para que el código resultante sea más fácil y rápido de interpretar por la máquina al que va dirigido.
- **Generación de código**: genera el código objeto de nuestro programa en un código en lenguaje máquina relocalizable, con diversas posiciones de memoria sin establecer, ya que no sabemos en qué parte de la memoria volátil se va a ejecutar nuestro programa.

- **Enlazador de librerías**: como hemos comentado, se enlaza nuestro código objeto con las librerías necesarias, produciendo en último término nuestro código final o código ejecutable.

Veamos una ilustración que muestra el proceso de compilación de un modo gráfico más claro.



(*) Lenguajes de alto nivel = son aquellos en los que las instrucciones o sentencias son escritas con palabras similares a las de los lenguajes humanos (mayormente en Inglés). Esto facilita la escritura y comprensión del código al programador alejándose del código máquina.

Veamos un ejemplo de niveles de abstracción desde el más alejado del código máquina al más cercano.

Lenguaje de alto nivel	Lenguaje ensamblador	Lenguaje máquina

A = B + C	LDA 0,4,3
	LDA 2,3,3
	ADD 2,0
	STA 0,5,3
		021404
		031403
		143000
		041405

3. Ingeniería de software

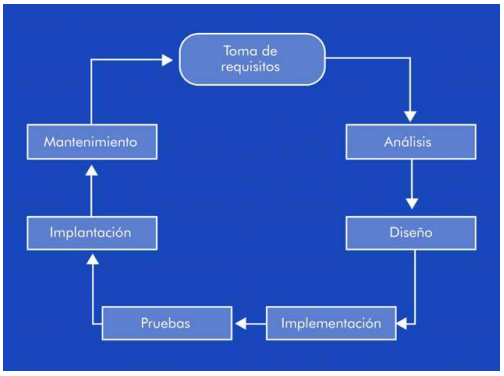
La ingeniería del software es casi tan antigua como el propio software y parte de los problemas que hicieron nacer este sector de la industria del software siguen vigentes, la mayoría sin una solución clara y definida. Pese a ello, se han hecho grandes avances en todos los campos desde la estimación de costes, pasando por la gestión de los equipos de trabajo, documentación, y muy notablemente en pruebas, calidad y seguridad.

La ingeniería del software trata de métodos y técnicas para desarrollar y mantener software desde que es concebido, se produce la necesidad de crear un software hasta que está listo para usarse.

4. Fases del Proceso de desarrollo del software

El desarrollo de un software o de un conjunto de aplicaciones pasa por diferentes etapas desde que se produce la necesidad de crear un software hasta que se finaliza y está listo para ser usado por un usuario. Ese conjunto de etapas en el desarrollo del software responde al concepto de ciclo de vida del software. No todos los programas ni en todas las ocasiones el proceso de desarrollo llevará finalmente las mismas etapas en el proceso de desarrollo; no obstante, son unas directrices muy recomendadas.

Hay más de un modelo de etapas de desarrollo (en cascada, evolutivo: iterativo incremental, en espiral) que de modo recurrente suelen ser compatibles y usadas entre sí, sin embargo vamos a estudiar uno de los modelos más extendidos y completos, el modelo en cascada.



Análisis de Requisitos: Esta fase se centra sobre el ¿QUÉ? Es decir, durante la definición, el que desarrolla el software intenta identificar:

1. ¿Qué información ha de ser procesada?
2. ¿Qué función y rendimiento se desea?
3. ¿Qué interfaces van a ser establecidas?

Se analizan y especifican los requisitos o capacidades que el sistema debe tener porque el cliente así lo ha pedido.

Se especifican dos **tipos de requisitos**:

Requisitos funcionales que describen con detalle la función que realiza el sistema, cómo reacciona ante determinadas entradas, como se comporta en situaciones particulares, etc.

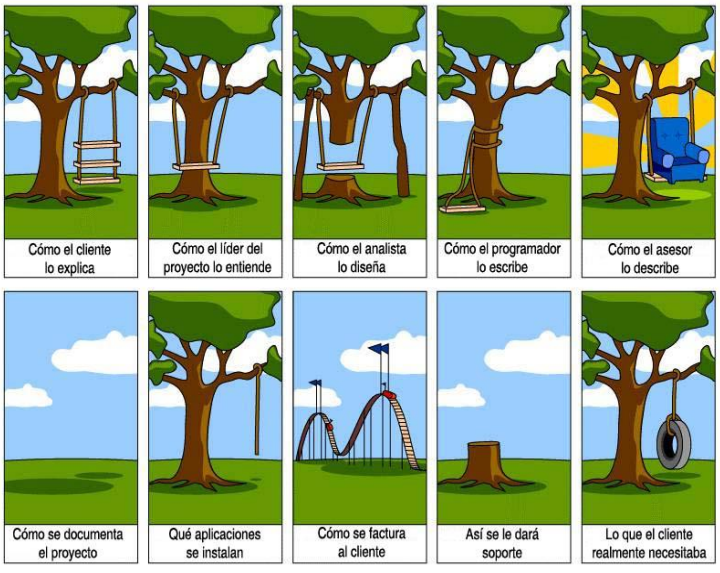
Requisitos no funcionales que tratan sobre las características del sistema, como puede ser la plataforma hardware, software, limitaciones de costes y plazos, la fiabilidad, mantenibilidad, restricciones, etc.

Para **representar los requisitos** se utilizan diferentes **técnicas**: Diagramas de flujo de datos (DFD), Diagramas de flujo de control (DFC), Diagramas de transición de estados (DTE), Diagramas de Entidad/Relación (DER), Diccionario de datos (DD), Diagramas de Clases (DC), Casos de Uso esenciales de alto nivel (CU)

El resultado del análisis de requisitos con el cliente se plasma en el documento **ERS, *Especificación de Requerimientos del Software***. Este documento no debe tener ambigüedades, debe ser completo, consistente, fácil de verificar y modificar, fácil de utilizar en la fase de explotación y mantenimiento, y fácil de identificar el origen y las consecuencias de los requisitos. Sirve como entrada para la siguiente fase en el desarrollo de la aplicación. De esta etapa depende en gran medida el logro de los objetivos finales.

Mientras que los clientes piensan que ellos saben lo que el software tiene que hacer, se requiere de habilidad y experiencia en la ingeniería de software para reconocer requisitos incompletos, ambiguos, contradictorios o incluso necesarios.

Esta es la realidad de un proyecto informático, sus VERDADERAS 10 fases de desarrollo… explica la diferencia entre lo que quiere el cliente y lo que el equipo de desarrollo ofrece luego de su solicitud.



Diseño y Arquitectura

Esta fase se centra en el **¿CÓMO?** Es decir, componer la forma en que se solucionara el problema. Se traducen los requisitos funcionales y no funcionales en una representación del software. Durante el desarrollo un ingeniero del software intenta definir:

1. ¿Cómo han de diseñarse las estructuras de datos?
2. ¿Cómo ha de implementarse la función dentro de una arquitectura de software?
3. ¿Cómo han de caracterizarse interfaces?
4. ¿Cómo ha de traducirse el diseño en un lenguaje de programación (o lenguaje no procedimental)?

Principalmente hay dos tipos de diseño, el **diseño estructurado** que está basado en el flujo de los datos a través del sistema; y el **diseño orientado a objetos** donde el sistema se entiende como un conjunto de objetos que tienen propiedades y comportamientos, además de eventos que activan operaciones que modifican el estado de los objetos; los objetos interactúan de manera formal con otros objetos.

El **diseño estructurado** (diseño clásico) produce un modelo de diseño con 4 elementos:

- Diseño de datos. Se encarga de transformar el modelo de dominio de la información creado durante el análisis, en las estructuras de datos que se utilizarán para implementar el software. El diseño de datos está basado en los datos y las relaciones definidos en el diagrama entidad relación y en la descripción detallada de los datos (definidos en el diccionario de datos).
- Diseño arquitectónico. Se centra en la representación de la estructura de los componentes del software, sus propiedades e interacciones. Un componente del software puede ser tan simple como un módulo de programa, pero también puede ser algo tan complicado como una base de datos o los conectores que permitan la comunicación, coordinación y cooperación entre los componentes. Partiendo de los DFD se establece la estructura modular del software que se desarrolla.
- 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 (diseño procedimental). Transforma los elementos estructurales de la arquitectura del software en una descripción procedimental de los componentes del software. El resultado de esta tarea es 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 u ordinograma, diagramas de cajas, tablas de decisión, pseudocódigo, etc. Las **construcciones lógicas** fundamentales con las que puede formarse cualquier programa son: secuencial, condicional y repetitiva.

El **diseño de software orientado a objetos (DOO)** es difícil. Para llevarlo a cabo hay que partir un análisis orientado a objetos (AOO). En dicho análisis se definen todas las clases que son importantes para el problema que se trata de resolver, las operaciones y los atributos asociados, las relaciones y comportamientos y las comunicaciones entre clases.
Define 4 capas de diseño:

- Subsistema. Se centra en el diseño de los subsistemas que implementan las funciones principales del sistema.

- Clases y Objetos. Especifica la arquitectura de objetos global y la jerarquía de clases requerida para implementar un sistema.
- Mensajes. Indica cómo se realiza la colaboración entre los objetos.
- Responsabilidades. Identifica las operaciones y atributos que caracterizan cada clase.

Para el análisis y diseño orientado a objetos se utiliza **UML** *(Unified Modeiing Langitage - Lenguaje de Modelado Unificado).* Es un lenguaje de modelado basado en diagramas que sirve para expresar modelos (un modelo es una representación de la realidad donde se ignoran detalles de menor importancia).
Se pasará de casos de uso esenciales a su definición como **casos de uso expandidos** reales
Se realizarán **diagramas de secuencia** entre otros para una representación temporal de los objetos y sus relaciones
Con todos estos diagramas y la información de entrada y salida, las estructuras de datos y la división modular se obtendrá el **cuaderno de carga**.

Programación o implementación o desarrollo del programa

Transformar un diseño mediante un lenguaje de programación en un programa puede ser la parte más obvia del trabajo de ingeniería de software, pero no es necesariamente la porción más larga. La complejidad y la duración de esta etapa está íntimamente ligada al o a los lenguajes de programación utilizados.

Gracias a las etapas anteriores, el programador contará con un análisis completo del sistema que hay que codificar y con una especificación de la estructura básica que se necesitará, por lo que en un principio solo habría que traducir el cuaderno de carga en el lenguaje deseado para culminar la etapa de codificación, pero esto no es siempre así, las dificultades son recurrentes mientras se codifica. Por supuesto que cuanto más exhaustivo haya sido el análisis y el diseño, la tarea será más sencilla, pero nunca está exento de necesitar un reanálisis o un rediseño al encontrar un problema al programar el software.

Pruebas

En esta fase se comprueba el funcionamiento de cada programa y esto se hace con datos reales o ficticios. Con una doble funcionalidad, las pruebas buscan confirmar que la codificación ha sido exitosa y el software no contiene errores, a la vez que se comprueba que el software hace lo que debe hacer, que no necesariamente es lo mismo.

- 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, sí el software construido se ajusta a los requisitos del cliente.

Para llevar a cabo el diseño de casos de prueba se utilizan dos técnicas:

- Prueba de caja blanca: se centran en validar la estructura interna del programa (necesitan conocer los detalles procedimentales del código)

- Prueba de caja negra: se centran en validar los requisitos funcionales sin fijarse en el funcionamiento interno del programa (necesitan saber la funcionalidad que el código ha de proporcionar).

Estas pruebas no son excluyentes y se pueden combinar para descubrir diferentes tipos de errores

El objetivo en esta etapa es planificar y diseñar pruebas que sistemáticamente saquen a la luz diferentes clases de errores, haciéndolo con la menor cantidad de tiempo y de 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.

No es un proceso estático, y es usual realizar pruebas después de otras etapas, como la documentación. Una técnica de prueba es probar por separado cada módulo del software, y luego probarlo de forma integral, para así llegar al objetivo. Se considera una buena práctica el que las pruebas sean efectuadas por alguien distinto al desarrollador que la programó, sin perjuicio de lo anterior el programador debe hacer sus propias pruebas. En general hay dos grandes formas de organizar un área de pruebas, la primera es que esté compuesta por personal inexperto y que desconozca el tema de pruebas, de esta forma se evalúa que la documentación entregada sea de calidad, que los procesos descritos son tan claros que cualquiera puede entenderlos y el software hace las cosas tal y como están descritas. El segundo enfoque es tener un área de pruebas conformada por programadores con experiencia, personas que saben sin mayores indicaciones en qué condiciones puede fallar una aplicación y que pueden poner atención en detalles que personal inexperto no consideraría.

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.

- Proporcionar información para ayudar a planificar la gestión del presupuesto y programar el proceso del desarrollo del software.

- Ser un repositorio de información del sistema para ser utilizado por el personal de mantenimiento.

- Algunos de los documentos 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. Son documentos en los que se indican planes, estimaciones y horarios que se utilizan para predecir y controlar el proceso de software, que informan sobre cómo usar los recursos durante el proceso de desarrollo, sobre normas de cómo se ha de implementar el proceso.

- **La documentación del producto**. Esta documentación describe el producto que está siendo desarrollado. Define dos tipos de documentación: la documentación del sistema que describe el producto desde un punto de vista técnico, orientado al desarrollo y mantenimiento del mismo y la documentación del usuario que ofrece una descripción del producto orientada a los usuarios que utilizarán el sistema.

Los usuarios finales utilizan el software para realizar alguna tarea, quieren saber cómo el software les ayuda a realizar su tarea, no están interesados en el equipo informático o en los detalles del programa. La documentación para el usuario debe mostrar una información completa y de calidad que ilustre mediante los recursos más adecuados cómo manejar la aplicación. Una buena documentación debería permitir a un usuario cualquiera comprender el propósito y el modo de uso de la aplicación sin información previa adicional.(Manual de usuario).

Los administradores del sistema son responsables de la gestión de los programas informáticos utilizados por los usuarios finales, pueden actuar como un gestor de red si el sistema implica una red de estaciones de trabajo o como un técnico que arregla problemas de software de los usuarios finales y que sirve de enlace entre los usuarios y el proveedor de software. La documentación técnica, destinada a equipos de desarrollo, explica el funcionamiento interno del programa, haciendo especial hincapié en explicar la codificación del programa. Se pretende con ello permitir a un equipo de desarrollo cualquiera poder entender el programa y modificarlo si fuera necesario. En casos donde el software realizado sea un servicio que pueda interoperar con otras aplicaciones, la documentación

técnica hace posible que los equipos de desarrollo puedan realizar correctamente el software que trabajará con nuestra aplicación.

En definitiva, la documentación del proceso se utiliza para gestionar todo el proceso de desarrollo del software. La documentación del producto se utiliza después de que el sistema está en funcionamiento, aunque también es esencial para la gestión del desarrollo del sistema.

Todo lo concerniente a la documentación del propio desarrollo del software y de la gestión del proyecto, pasando por modelaciones (UML), diagramas, pruebas, manuales de usuario, manuales técnicos, etc; todo con el propósito de eventuales correcciones, usabilidad, mantenimiento futuro y ampliaciones al sistema.

Implantación

En esta etapa se lleva a cabo la instalación y puesta en marcha del producto software en el entorno de trabajo del cliente. Para ello se implementa el software en el sistema elegido o se prepara para que se implemente por sí solo de manera automática.
Cabe destacar que en caso de que nuestro software sea una versión sustitutiva de un software anterior es recomendable valorar la convivencia de sendas aplicaciones durante el proceso de adaptación.

En esta etapa se llevan a cabo las siguientes tareas:

- 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 un seguimiento de los problemas y para probar el producto software en el entorno de trabajo.
- **Pruebas de operación**. Para cada producto software, se llevarán a cabo pruebas de funcionamiento y tras satisfacerse los criterios especificados, se libera el software para uso operativo.
- **Uso operacional del sistema**. El sistema entrará en acción en el entorno previsto de acuerdo con la documentación de usuario.
- **Soporte al usuario**. Se deberá proporcionar asistencia y consultoría a los usuarios cuando la soliciten. Estas peticiones y las acciones subsecuentes se deberán registrar y supervisar.

Mantenimiento

El mantenimiento del software se define como la modificación de un producto de software después de la entrega para corregir los fallos, para adaptar el producto a un entorno modificado o para mejorar el rendimiento u otros atributos.
Existen **cuatro tipos de mantenimiento** del software que dependen de las demandas de los usuarios del producto software a mantener:

- Mantenimiento correctivo. Es muy probable que después de la entrega del producto, el cliente **encuentre errores o defectos**, a pesar de las pruebas y verificaciones

realizadas en las etapas anteriores. Este tipo de mantenimiento tiene como objetivo corregir los fallos descubiertos.

- Mantenimiento adaptativo. Con el paso del tiempo es posible que se produzcan cambios en el entorno original (CPU, S.O., reglas de la empresa, etc.) para el que se desarrolló el software. El 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. Este tipo de mantenimiento es el más usual debido a los rápidos cambios que se producen en la tecnología informática, que en la mayoría de ocasiones dejan obsoletos los productos software desarrollados.
- Mantenimiento perfectivo. Conforme el cliente utiliza el software puede descubrir **funciones adicionales** que le pueden aportar beneficios. El mantenimiento perfectivo es la modificación del producto de software orientado a incorporar nuevas funcionalidades (más allá de los requisitos funcionales originales) y nuevas mejoras en el rendimiento o el mantenimiento del producto.
- Mantenimiento preventivo. Consiste en la modificación del producto de software sin alterar las especificaciones del mismo, con el fin de **mejorar** y facilitar las tareas de mantenimiento. Este tipo de mantenimiento hace cambios en programas con el fin de que se puedan corregir, adaptar y mejorar más fácilmente, por ejemplo, se pueden reestructurar los programas para mejorar su legibilidad o añadir nuevos comentarios que faciliten la comprensión del mismo. A este mantenimiento también se le llama *reingeniería del software.*

Esto puede llevar más tiempo incluso que el desarrollo inicial del software. Alrededor de 2/3 de toda la ingeniería de software tiene que ver con dar mantenimiento. Una pequeña parte de este trabajo consiste en arreglar errores, o *bugs*. La mayor parte consiste en extender el sistema para hacer nuevas cosas.

Cuando el mantenimiento que hay que realizar consiste en una ampliación, el modelo en cascada suele volverse cíclico, por lo que, dependiendo de la naturaleza de la ampliación, puede que sea necesario analizar, diseñar, codificar, probarla, documentar, implementar y, dar soporte de mantenimiento sobre la misma, por lo que al final este modelo es recursivo y cíclico para cada aplicación y no es un camino rígido de principio a fin.

5. Metodología de desarrollo de software

Metodología de desarrollo de software en ingeniería de software es un marco de trabajo usado para estructurar, planificar y controlar el proceso de desarrollo en sistemas de información. Para el desarrollo del software tenemos que utilizar una metodología. Se llama Metodología al conjunto de pasos, métodos, técnicas y herramientas que se deben utilizar o seguir para el desarrollo del software.

En una metodología se definen:

- Una estructura de proyecto que sirva de guía al equipo de trabajo, involucre a los usurarios en su desarrollo y en sus puntos decisivos.
- Un conjunto de productos finales a desarrollar.
- Un conjunto de técnicas para obtener los productos finales.
- Las diferentes responsabilidades y funciones de los miembros del equipo de proyecto y de los usuarios.

Proceso Unificado Racional (RUP), es una metodología de desarrollo de software, basado en UML(Unified Modeling Language)

6. Relación del software con los componentes del sistema

Hay cuatro componentes de tipo hardware:

Dispositivos de Entrada, Dispositivos de salida, Unidad Central de Proceso, Memoria Principal y Secundaria

- Dispositivo de entrada

Los Dispositivos de Entrada son aquéllos que permiten introducir la información al ordenador. El mouse, El teclado, El micrófono, La webcam, el escáner entre otros...

-Dispositivo de salida

Los Dispositivos de Salida son aquéllos que envían la información del interior de la computadora al exterior, mostrándola al usuario como caracteres, como imágenes, como páginas impresas. La impresora, El monitor, Los altavoces...

-Unidad Central de proceso CPU

Constituye la parte operacional más importante de todo sistema computacional. La CPU es el cerebro de la computadora.

-Memoria Principal y Secundaria

La Memoria Principal es un área de almacenamiento interno de la computadora, de acceso rápido, donde se almacenan las instrucciones y los datos que la CPU necesita para ejecutar alguna tarea.

La Memoria Secundaria es un conjunto de componentes electrónicos que almacenan programas y datos de forma permanente. Disco duro, CD, DVD, Pen Drive,…

-Hardware

Se refiere a todos los componentes físicos que intervienen en un sistema computacional. Es todo lo tangible. El disco duro, impresoras, mouse, chips, teclado, monitores.

-Software

Se refiere a las instrucciones, datos o programas con los cuales opera la computadora. Todo lo que puede ser almacenado electrónicamente es software. Es todo lo intangible.

7. Roles que interactúan en el proceso de desarrollo del software

A lo largo del proceso de desarrollo de un software deberemos realizar, como ya hemos visto anteriormente, diferentes y diversas tareas. Es por ello que el personal que interviene en el desarrollo de un software es tan diverso como las diferentes tareas que se van a realizar.

Los roles no son necesariamente rígidos y es habitual que participen en varias etapas del proceso de desarrollo.

- Analista de sistemas
  - Uno de los roles más antiguos en el desarrollo del software. Su objetivo consiste en realizar un estudio del sistema para dirigir el proyecto en una dirección que garantice las expectativas del cliente determinando el comportamiento del software.
  - Participa en la etapa de análisis.
- Diseñador de software
  - Nace como evolución del analista y realiza, en función del análisis de un software, el diseño de la solución que hay que desarrollar.
  - Participa en la etapa de diseño.
- Analista programador
  - Comúnmente llamado "desarrollador", domina una visión más amplia de la programación, aporta una visión general del proyecto más detallada diseñando una solución más amigable para la programación y participando activamente en ella.
  - Participa en las etapas de diseño y programación.
- Programador
  - Se encarga de manera exclusiva de crear el resultado del estudio realizado por analistas y diseñadores. Escribe el código fuente del software.
  - Participa en la etapa de programación.
- Arquitecto de software
  - Es la argamasa que cohesiona el proceso de desarrollo. Conoce e investiga los frameworks y tecnologías revisando que todo el procedimiento se lleva a cabo de la mejor forma y con los recursos más apropiados.

8. Arquitecturas de Software

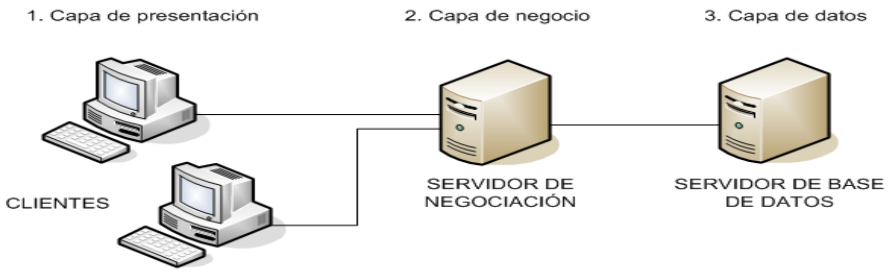
La Arquitectura de software es el diseño de nivel más alto de la estructura de un sistema, enfocándose más allá de los algoritmos y estructuras de datos. La Arquitectura de software es un conjunto de decisiones que definen a nivel de diseño los componentes computacionales y la interacción entre ellos para garantizar que el proyecto llegue a buen término.
Una Arquitectura de software, también denominada *Arquitectura lógica*, se refiere a la estructura del sistema a desarrollar.

La Arquitectura de software establece los fundamentos (normas, pautas) para que analistas, diseñadores, programadores, etc. trabajen en una línea común que permita alcanzar los objetivos del sistema de información, cubriendo todas las necesidades.

La arquitectura del software es la base para comenzar con un proyecto que nos asegure la calidad del mismo.

Toda arquitectura lógica debe ser implementable en una arquitectura física, que consiste simplemente en determinar qué computadora tendrá asignada cada tarea.

En la actualidad cualquier aplicación cuenta generalmente con tres partes diferenciadas:
1. Una interfaz de usuario: Elemento con el que interacciona el usuario de la aplicación, ejecutando acciones, introduciendo u obteniendo información.
2. Lógica ó Reglas de negocio: Son las que procesan la información para generar los resultados que persiguen, siendo un elemento que diferencia unas aplicaciones de otras.
3. Gestión de datos: Se ocupa del almacenamiento y recuperación de la información.



Todas estas capas pueden residir en un único ordenador, si bien lo más usual es que haya una multitud de ordenadores en donde reside la capa de presentación (son los clientes de la arquitectura cliente/servidor). Las capas de negocio y de datos pueden residir en el mismo ordenador, y si el crecimiento de las necesidades lo aconseja se pueden separar en dos o más ordenadores. Así, si el tamaño o complejidad de la base de datos aumenta, se puede separar en varios ordenadores los cuales recibirán las peticiones del ordenador en que resida la capa de negocio.

Si, por el contrario, fuese la complejidad en la capa de negocio lo que obligase a la separación, esta capa de negocio podría residir en uno o más ordenadores que realizarían solicitudes a una única base de datos. En sistemas muy complejos se llega a tener una serie de ordenadores sobre los cuales se ejecuta la capa de negocio, y otra serie de ordenadores sobre los cuales se ejecuta la base de datos.

En una arquitectura de tres niveles, los términos "capas" y "niveles" no significan lo mismo ni son similares.

En cambio, el término "nivel" corresponde a la forma en que las capas lógicas se encuentran distribuidas de forma física. Por ejemplo:

- Una solución de tres capas (presentación, lógica del negocio, datos) que residen en un solo ordenador (Presentación+lógica+datos). Se dice que la arquitectura de la solución es de tres capas y *un nivel*.
- Una solución de tres capas (presentación, lógica del negocio, datos) que residen en dos ordenadores (presentación+lógica por un lado; lógica+datos por el otro lado). Se dice que la arquitectura de la solución es de tres capas y *dos niveles*.

Las arquitecturas más universales son:

- Monolítica. Donde el software se estructura en grupos funcionales muy acoplados, involucrando los aspectos referidos a la presentación, procesamiento y almacenamiento de la información. Todas las capas en un solo ordenador.

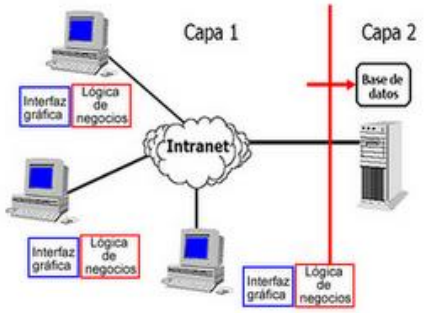
  Están considerados las distintas aplicaciones para escritorio: sistemas operativos, ofimática, juegos monousuario, etc.

- Cliente-servidor. Donde el software se reparte en dos partes independientes pero sin reparto claro de funciones.

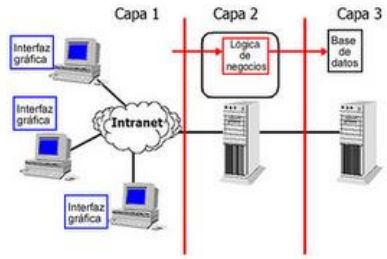
  Por un lado tenemos una computadora que solicita información la cual genera una serie de peticiones. Esta computadora es generalmente llamada Cliente. Por otro lado, tenemos una computadora más potente que se encarga de recibir dichas peticiones y procesarlas para luego devolver los datos procesados a la computadora que ha solicitado peticiones. Esta computadora es llamada Servidor.

  Esta arquitectura consiste básicamente en un cliente que realiza peticiones a otro programa (el servidor) que le da respuesta. Aunque esta idea se puede aplicar a programas que se ejecutan sobre una sola computadora es más ventajosa en un sistema operativo multiusuario distribuido a través de una red de computadoras.

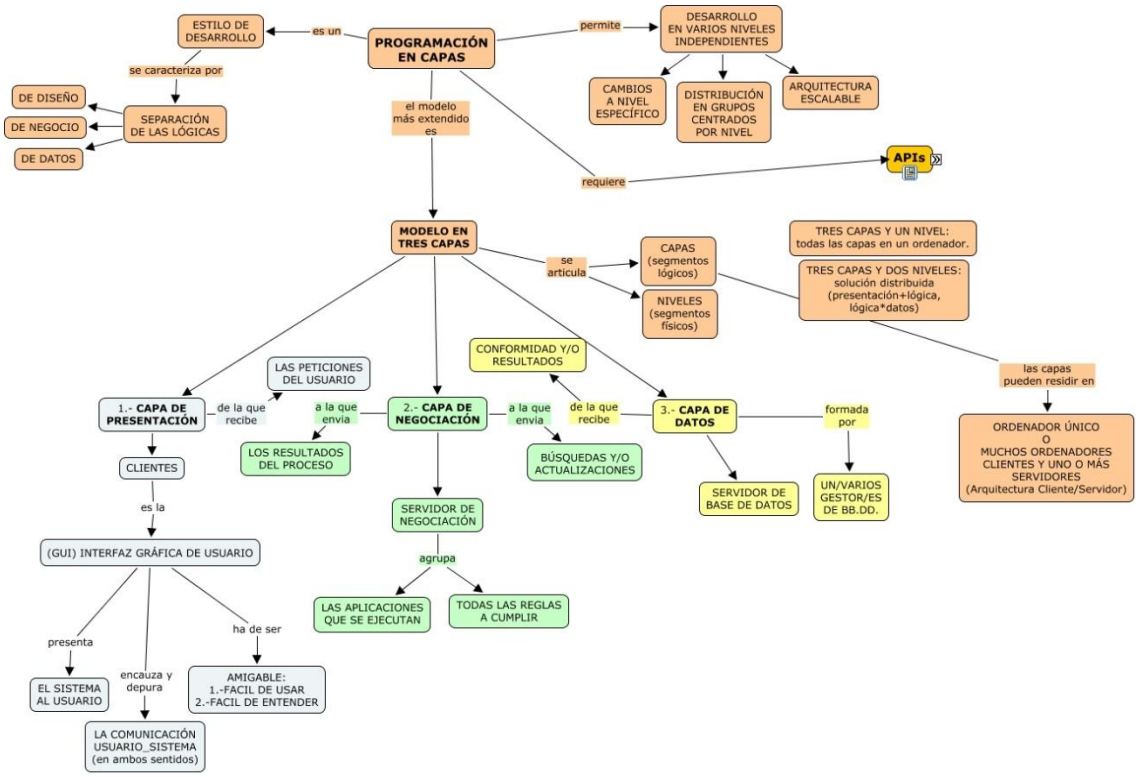
  La separación entre cliente y servidor es una separación de tipo lógico, donde el servidor no se ejecuta necesariamente sobre una sola máquina ni es necesariamente un sólo programa. Los tipos específicos de servidores incluyen los servidores web, los servidores de archivo, los servidores del correo, etc. Mientras que sus propósitos varían de unos servicios a otros, la arquitectura básica seguirá siendo la misma.



- Arquitectura de tres niveles. Especialización de la arquitectura cliente-servidor donde la carga se divide en tres partes (o capas) con un reparto claro de funciones: una capa para la presentación (interfaz de usuario), otra para el cálculo (donde se encuentra modelado el negocio) y otra para el almacenamiento



(persistencia).



El **objetivo** principal de la arquitectura de software consiste en proporcionar elementos que ayuden a la toma de decisiones abstrayendo los conceptos del sistema mediante un lenguaje común. Dicho conjunto de herramientas, conceptos y elementos de abstracción se organizan en forma de **patrones y modelos**.

Los resultados obtenidos después de efectuar buenas prácticas de arquitectura de software deben proporcionar capas de abstracción y encapsulado, organizando el software de manera diferente dependiendo de la visión o criterio de la estructura.

8.1. Patrones de diseño

Los patrones de diseño, también llamados **patrones de desarrollo**, ofrecen excelentes soluciones a muchos de los problemas que como desarrolladores de software encontramos día a día.

Los patrones de diseño establecen los componentes de la arquitectura y la funcionalidad y comportamiento de cada uno.

Las directrices marcadas por los patrones de diseño facilitan la tarea de diseñar un software correctamente en menos tiempo, ayudan a construir soluciones reutilizables y extensibles, y facilita la documentación. Los patrones nos dicen cómo aplicar de manera eficaz la herencia, el polimorfismo y todas las ventajas que posee el Paradigma Orientado a Objetos.

Los patrones no especifican todas las características o relaciones de los componentes de nuestro software, sino que están centrados en un ámbito específico. Cada patrón determina y especifica los aspectos de uno de los cuatro ámbitos: creacionales, estructurales, de comportamiento y de interacción.

Los **patrones creacionales** definen el modo en que se construyen los objetos, principalmente con el objetivo de encapsular la creación de los mismos haciendo que los constructores sean privados y el modo de crear una instancia sea mediante un método estático que devuelva el objeto. La característica fundamental de la programación orientada a objetos en la que recaen estas prácticas de patrones creacionales es el polimorfismo.

Los **patrones estructurales** establecen las relaciones y organizaciones entre los distintos componentes de nuestro software, resolviendo de una manera elegante diversos problemas que nos encontramos al implementar soluciones sin haber evaluado previamente todas las consecuencias y posibilidades. No hay que olvidar que el uso de patrones no es una cuestión unitaria, para solucionar un problema o eventualidad cuando surja, sino que nos plantea una serie de directrices que pueden solventar problemas futuros, hay que pensar a largo plazo y valorar siempre la expandibilidad del proyecto.

Los **patrones de comportamiento** describen las comunicaciones entre objetos y clases, estableciendo directrices sobre cómo utilizar los objetos y clases para optimizar y organizar el comportamiento, interacción y comunicación entre ellos.

Los **patrones de interacción** tienen por objetivo definir diferentes directrices para la creación de los interfaces, en donde prima la usabilidad. El aspecto gráfico y la usabilidad de los controles ofrecidos al usuario para manejar la aplicación son sin duda una parte sumamente importante en un software, y no es algo que se deba menospreciar. Los patrones de interacción se alejan parcialmente del ámbito de los otros patrones de diseño y no entran es su totalidad en los patrones de diseño.

Los **antipatrones** son la contraparte de los patrones, es decir, que si los patrones son buenas prácticas de desarrollo de software, los antipatrones son justamente lo contrario, situaciones o prácticas que no tenemos que hacer o tenemos que evitar a la hora de desarrollar un software, por ejemplo, presencia de saltos constantes a modo de llamadas o bucles (código spaghetti), fragmentos de código sin terminar fruto de cambios recurrentes (flujo de lava), etc.

8.2. Desarrollo en tres capas: Modelos

El desarrollo en capas nace de la necesidad de separar la lógica de la aplicación del diseño, separando a su vez los datos de la presentación al usuario. Para solventar esa necesidad, se ideó el desarrollo en tres capas, que separa la lógica de negocio, el acceso a datos y la presentación al usuario en tres capas que pueden tener cada una tantos niveles como se quiera o se necesite.

Presentación	• Comunica a la aplicación con el usuario. • Solo se comunica con la capa de negocio.
Negocio	• Alberga los programas de la aplicación. • Se comunica con la capa de presentación y de datos.
Datos	• Es el acceso a datos, tanto para solicitar como persistir. • Se comunica con la capa de negocio.

Desarrollo en tres capas

Capa de presentación

- Esta capa es la que ve el usuario final en su pantalla, sobretodo en aplicaciones totalmente enfocadas en la filosofia cliente-servidor.
- Lo que ve el usuario final da lo mismo lo que sea, puede ser un formulario web, un formulario de Windows, etc…
- Esta capa es la que permite la entrada de datos hacía el programa y a partir de allí que empiece la gestión de los datos.

Capa de negocio

- Esta capa se considera el corazón del software. El programa tiene que pasar gran parte del proceso de datos en esta capa intermedia, ya que su función es muy importante: Aplicar las reglas de negocio para la gestión de datos.
- Es decir, si por ejemplo tenemos un campo de texto en un formulario, tendremos un método que se encargue de pasar el dato que hay en la caja de texto hacia la capa de negocio. Esta analizará el texto si tenemos aplicado alguna regla de negocio (que no contenga números, que contenga una longitud predeterminada, etc…) y, si pasa el filtro, los datos pasaran a la capa de datos. Y en caso contrario, saltará un aviso de error para que se pueda corregir por el usuario.
- Es decir, la capa de presentación solamente sirve para interactuar con el usuario. Una vez el usuario introduzca o interaccione con el programa, la capa de presentación pasa a un 2º plano, delegando el potencial del programa a la capa de negocio.

Capa de datos

- Una vez los datos han pasado la regla de negocio, llega la capa de datos, en que los datos están preparados para ser guardados en una base de datos después de su correcto tratamiento. Y una vez guardados en la base de datos, se devuelve la confirmación del proceso. O en su caso contrario, saltará un error.

En un lenguaje POO (es decir, Programación Orientada a Objetos) las capas pueden ser conjuntos de clases y, en caso de ser un equipo de desarrollo de software, cada grupo se podría ocupar de programar una capa en concreto y así optimizar el proceso de mejoría en los programas, ya que se tardará menos tiempo en lanzar una nueva versión.

Es decir, puedes mejorar la capa de datos (mejoría en la gestión de consultas, etc….) sin que la capa de negocio se vea afectada, por ejemplo. Programar un poco por módulos.

Además, las capas no hacen falta que estén todas en el mismo ordenador, sino que en caso de tener una red LAN, cada capa puede estar en un PC distinto. Y aquí es donde entra en juego la importancia de que **una aplicación esté bien estructurada**.

El desarrollo por capas no solo nos mejora y facilita la estructura de nuestro propio software, sino que nos aporta la posibilidad de interoperar con otros sistemas ajenos a nuestra aplicación. Por ejemplo, podríamos necesitar acceder y utilizar datos contenidos tanto en nuestra propia base de datos, como la base de datos de un banco, para ello utilizaremos la capa de datos, en donde accederíamos a nuestro gestor de base de datos y al servicio que nos ofrezca el banco para solicitar dichos datos. Esto podría hacerse sin necesitar un desarrollo en tres capas, pero la principal ventaja (aparte de la encapsulación y ocultación de código y datos entre capas) que nos aporta reside en evitar modificar la lógica de negocio por necesitar acceder a diferentes datos, todo está perfectamente estructurado y nos permite modificar las fuentes y el modo en que accedemos a los datos de los programas que trabajan con ellos.

9. Entorno de desarrollo y entorno de producción

En muchas ocasiones, trabajamos con equipos, servicios o aplicaciones críticas. Un fallo o error en la configuración del equipo, del servidor o de la aplicación implica un tiempo de inactividad inadmisible en ciertas tareas. Por ejemplo, una aplicación de tiempo real como la que controla el tráfico aéreo en España no puede interrumpirse sin causar graves perjuicios.

En estos casos es obligatorio (y en otros servicios menos críticos también es recomendable) emplear al menos dos entornos diferentes de trabajo. Por un lado tenemos el denominado **entorno de producción**, que engloba los equipos, servidores y aplicaciones que dan el servicio real. Por otro lado tenemos el conocido como **entorno de desarrollo** que incluye equipos, servidores y aplicaciones semejantes a los del entorno de producción.

Todos los pasos que realizamos para poner en marcha el entorno de producción se llevan a cabo previamente en el entorno de desarrollo, para probar que el funcionamiento es el

correcto. De la misma forma, cuando queremos realizar algún cambio o mejora en el entorno de producción, primero los llevamos a cabo en el entorno de desarrollo. Una vez que todos los cambios han sido aplicados y se ha verificado que funcionan correctamente y sin errores, dichos cambios se aplican en el entorno de producción. De esta manera se minimiza la posibilidad de que un cambio provoque un tiempo de inactividad en el equipo, en los servidores del equipo o en las aplicaciones instaladas en el mismo.

Se recomienda que el entorno de desarrollo y el de producción sean **lo más parecido posible** a nivel de hardware y software. Es decir, si tenemos un servidor funcionando en una máquina Intel de 64 bits corriendo el sistema operativo Linux, no será una buena idea utilizar como entorno de desarrollo un servidor AMD de 32 bits corriendo el sistema operativo Windows. En realidad, el entorno de desarrollo ideal es aquel que es **idéntico** al entorno de producción.

Cuando se aplican cambios en un entorno de desarrollo y se verifica que el funcionamiento es el esperado, existen dos posibilidades: volver a repetir manualmente los mismos cambios en el entorno de producción o programar los scripts que sean necesarios para poder aplicar los cambios de manera automática.