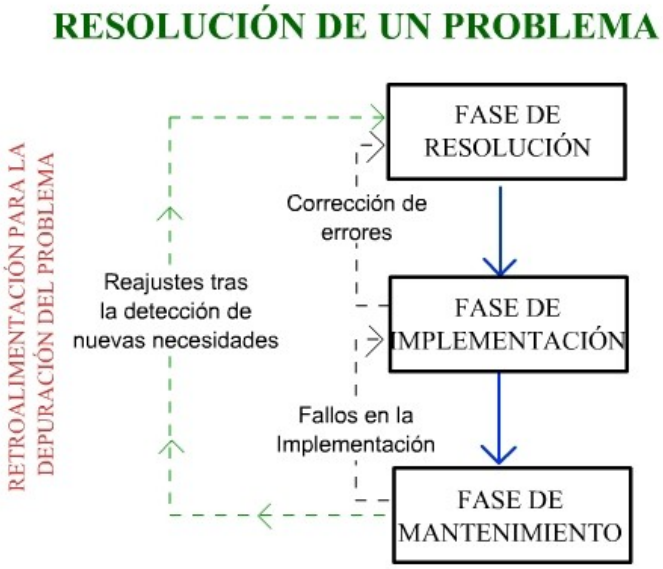
UNIDAD 1. PASOS PARA LA RESOLUCIÓN DE UN PROBLEMA

El tiempo que se dedica a preparar un programa **nunca es tiempo perdido**, ya que consiste en conocer todos y cada uno de los casos posibles y ello siempre redunda en la calidad del software (comprensión, facilidad de uso, facilidad de mejora, eficiencia, eficacia y perfecto control de excepciones).

Por ejemplo, imaginemos que se nos ha pedido que diseñemos una aplicación informática para el cálculo del Precio de Venta al Público de diferentes productos en función del tipo de IVA a aplicar.

El motivo por el que montones de personas aprenden lenguajes de programación es para poder usar los ordenadores como una herramienta para resolver los problemas. Consideramos que la solución de un problema comienza con la **definición** del mismo y termina con la verificación de que la solución encontrada es válida para todas las situaciones posibles.



La ***fase de resolución*** abarca hasta ese punto. Siguiendo nuestro ejemplo esta fase consistiría en entender bien el cálculo, cada uno de los tipos de IVA posibles, las situaciones en las que se aplican y a qué productos. Luego en conocer el método de cálculo (en este caso sumar el porcentaje correspondiente al precio del producto) y elaborar un algoritmo adecuado que permita llegar a una solución correcta. Finalmente una vez elaborado este algoritmo es preciso comprobar que es correcto y que los cálculos son exactos.

Pero en el caso de una solución informática, hay que ir un poco más lejos, ya que deberemos:

- "traducir" esa solución para que sea comprensible y ejecutable por un ordenador,
- corregir los errores que contenga y
- probar que su funcionamiento es el correcto.

Hasta aquí abarca la ***fase de implementación***. En el problema del cálculo del IVA, esta fase consistiría en codificar el algoritmo en un lenguaje de programación (por ejemplo Java, C, Basic, Pascal, etc.), para después compilarlo y una vez depurados todos los errores hacer todo tipo de pruebas para comprobar que su funcionamiento sea el esperado.

Pero incluso una vez que nuestra aplicación (o programa de ordenador) está terminada hay que procurar mantenerla actualizada, haciéndole ajustes, mejoras, adaptaciones, etc. siempre que sean necesarias. Esa es la ***fase de mantenimiento***. Esta fase en nuestro ejemplo del IVA, al ser tan sencillo podemos plantearla desde el punto de vista de alguien al que le gustan las aplicaciones con determinados colores, con sistemas de entrada de datos más sofisticados (códigos de barras) a través de bases de datos, o incluso la adaptación de este pequeño programa para su uso en aplicaciones mayores.
Naturalmente cada una de las fases enunciadas conviene dividirla a su vez en una serie de pasos que faciliten su desarrollo de forma más sencilla, cómoda y eficaz.

Fase de Resolución.

En esta fase **tratamos de acercarnos al problema**, **definirlo correctamente, e idear una forma de resolverlo**. También deberemos verificar que la solución ideada es válida para todos los casos posibles. Todo ello con independencia de que esa solución se vaya a llevar a cabo manualmente, con papel y lápiz, o con un ordenador. Intentamos describir esa solución de forma genérica, sin que todavía entremos en detalles asociados a un lenguaje de programación concreto o a las características de un

ordenador concreto. **El resultado de esta fase será un algoritmo, ya verificado**, expresado mediante alguna herramienta descriptiva. Los pasos que incluye esta fase (y que se muestran en la imagen) se detallan en los apartados siguientes.

Análisis del problema.

Aquí daremos una descripción muy superficial de esta fase, ya que es objeto de estudio del módulo profesional de Análisis y Diseño Detallado de Aplicaciones Informáticas de Gestión, dentro del ciclo formativo de DAI, tan extenso por cierto como el de Programación en Lenguajes Estructurados, que ahora es el que nos ocupa.

El análisis es la primera aproximación al problema. Consiste en **estudiar el problema**, asegurándonos de que lo entendemos bien, de que hemos tenido en cuenta todos los casos y situaciones posibles, etc. Una vez comprendido el problema, será necesario dar una definición lo más exacta posible del mismo, identificando qué tipo de **información** se debe producir y qué **datos** o elementos dados en el problema pueden usarse para obtener la solución.

Como resultado de un buen análisis del problema, obtendremos un **conjunto de especificaciones de entrada y de salida**, que nos definen los **requisitos que nuestra solución** al problema debe cumplir.

Las especificaciones de entrada deben responder a preguntas del tipo:

- ¿Qué **datos** son de entrada?
- ¿Cuál será el volumen de **datos** de entrada?
- ¿Cuándo se considerará que un dato de entrada no es válido?
- ¿En qué soporte están los **datos** de entrada?

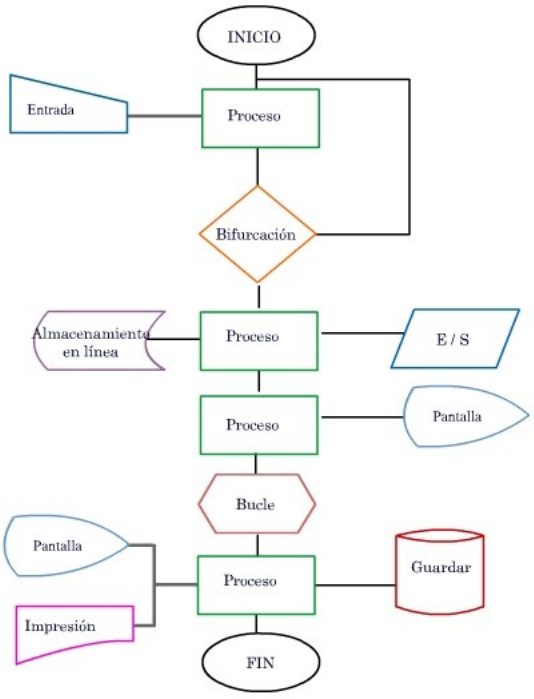
Las especificaciones de salida deben responder a preguntas del tipo:

- ¿Cuáles son los **datos** de salida?
- ¿Cómo se obtienen los **datos** de salida a partir de los de entrada?
- ¿Qué volumen de **datos** de salida se producirán?
- ¿Qué precisión deberán tener los resultados?
- ¿Cómo deben presentarse los **datos** de salida?
- ¿En qué soporte se almacenarán esos **datos**, si es que se almacenan?

Con esto ya debemos tener bastante claro nuestro problema, que está **definido de forma precisa y sin ambigüedades**, e incluso debemos tener alguna idea de cómo resolverlo.

Diseño del Algoritmo.

En este paso se trata de idear y representar de forma más o menos normalizada**, la solución del problema**. Para ello se usan métodos descriptivos o herramientas estándar, que puedan ser entendidos por cualquier **programador**, y que no admitan ambigüedades ni distintas interpretaciones, pero sin entrar en detalles de sintaxis del **lenguaje de programación**. **La idea es describir la solución al problema (algoritmo) con la claridad suficiente como para que trasladar la solución a cualquier lenguaje de programación concreto sea inmediata.**



En esta fase, frecuentemente nos encontraremos con la necesidad de resolver un problema complejo, en cuyo caso lo más adecuado es usar la técnica de "**divide y vencerás**", que consiste en **dividir el problema en subproblemas más sencillos, que pueden ser a su vez subdivididos hasta conseguir problemas de fácil solución.**

Esta técnica se conoce también como ***diseño descendente***, o ***diseño por refinamiento paso a paso o sucesivo***, ya que partimos de una descripción de un problema global, que en un primer refinamiento lo describimos como formado por varios subproblemas, cada uno de los cuales se irá detallando en cada refinamiento sucesivo, hasta llegar a los subproblemas básicos, que no necesitan ser subdivididos. **Se llama diseño descendente porque partimos de lo más complejo hasta llegar a lo más simple, descendiendo en nivel de dificultad en cada paso o refinamiento**.

Verificación "manual" del algoritmo.

Una vez que hemos descrito el algoritmo que da solución a nuestro problema, mediante alguna herramienta descriptiva (o lenguaje algorítmico) es necesario asegurarse de que efectivamente realiza todas las tareas para las que se ha diseñado de forma correcta, produciendo el resultado esperado. Para ello será necesario "ejecutar" manualmente el algoritmo, **probándolo para un conjunto de datos que incluyan todos los casos posibles,** incluidos los más extremos y los menos frecuentes, abarcando todo el rango de valores de entrada posibles. Deberemos ir anotando en una hoja los valores intermedios de todos los **datos** que se vayan calculando, y los resultados finalmente obtenidos.

Es muy posible que durante la verificación del algoritmo nos demos cuenta de que algunos fallos se deben a un mal diseño del algoritmo, o incluso a un análisis equivocado. Esto nos puede llevar a **volver a cualquiera de estas fases anteriores, bien añadiendo o modificando especificaciones, o bien proponiendo cambios en la manera de resolver el problema**. Después, sea cual sea el caso, deberemos volver a verificar el algoritmo, como es lógico, hasta que supere todas las pruebas realizadas.

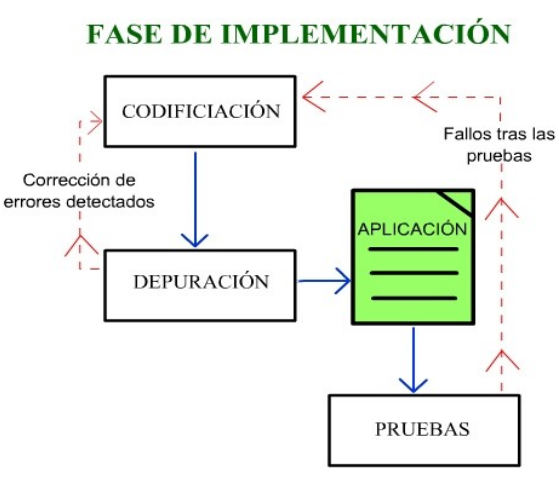
En la unidad didáctica 4, se explica más a fondo el concepto de algoritmo, y se ponen ejemplos tanto del análisis, como del diseño y de la verificación de algunos algoritmos. Tú también tendrás que realizar algunos como actividades, pero por ahora solo estamos contando los pasos que hay que dar. Aún no hemos empezado a andar. De momento es conveniente que sepas de qué hablamos, así que te proponemos que pruebes la siguiente aplicación en la que se explica el funcionamiento de un algoritmo para el cálculo del Precio de Venta al Público, dado su precio y el tipo de IVA que se le aplica. Se trata de una simulación de la verificación manual del algoritmo, tal y como debe llevarla a cabo el analista aplicándola a dos ejemplos en los que se detalla el comportamiento del algoritmo ante diferentes valores de entrada.

I. 1 Aplicación demostrativa de Verificación manual de Algoritmo. Pulse sobre la imagen.

AQUÍ PLE1_RECURSO1.SWF

Fase de Implementación.

A esta fase llegamos con una solución a nuestro problema, que ya sabemos que es correcta. Y además la tenemos descrita de una forma clara, sin ambigüedades, por lo que si queremos resolver el problema con un ordenador bastará con **pasar (traducir o codificar) del lenguaje algorítmico a lenguaje de programación concreto**, como por ejemplo Java, o C, o Delphi, o Visual Basic, etc.



En esta fase de implementación, además deberemos corregir los errores del programa escrito en el **un**

lenguaje de programación elegido

Los pasos a seguir en la fase de implementación son los que se describen en los apartados siguientes.

Codificación.

Este paso consiste en pasar de la descripción del algoritmo a un lenguaje de programación concreto, tal como Java. Una vez que hayamos elegido el lenguaje de programación concreto, sólo tendremos que ir siguiendo el algoritmo, y **escribiendo con la sintaxis de ese lenguaje las sentencias o instrucciones que hacen lo que se indica en el algoritmo**.

Por ejemplo, si el algoritmo indica:

```
nombre ⟵  “Juan”
Si (nombre = “Pepe”)
        Escribir ( “El nombre del empleado es ”, nombre)
Caso Contrario
        Escribir (“No lo conozco, pero se llama ” , nombre)
Fin-Si
```

En Java deberemos escribir algo como lo que sigue:

```
String nombre = “Juan” ;
if (nombre.equals(“Pepe”)){
        System.out.println(“El nombre del empleado es ”+ nombre) ;
}else {
        System.out.println(“No lo conozco, pero se llama ” + nombre) ;
}
```

Aún sin conocer todavía la sintaxis de Java ni la forma de describir el algoritmo en lenguaje algorítmico, vemos que existe bastante similitud entre la descripción más o menos formal, pero cercana a nuestra forma de hablar del algoritmo y el código escrito en lenguaje Java. La diferencia es que en el segundo tenemos que tener en cuenta una serie de detalles de **sintaxis del lenguaje**, tales como que las sentencias terminan con punto y coma, o las llaves para delimitar bloques de sentencias, o las palabras concretas que usamos para comprobar si se cumple o no una condición, o la forma de asignar un valor a una variable… En Java la sintaxis es esa y hay que escribirlo así, porque de otro modo, no funcionaría, el ordenador no lo entendería.

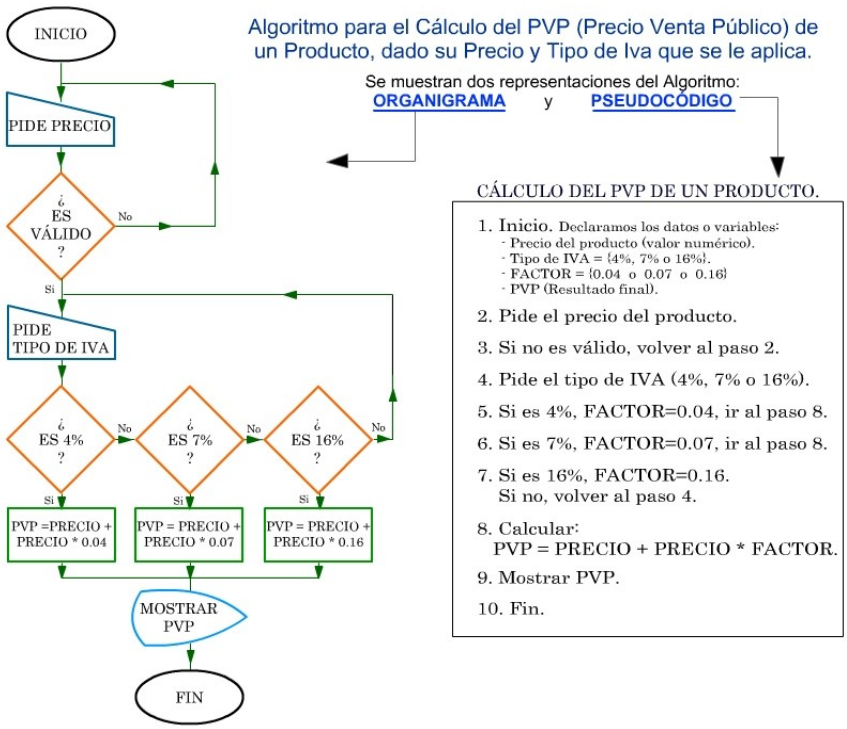
Por el contrario, en el algoritmo la sintaxis no es nada rígida. Podíamos haberlo expresado de otra forma, ya que el único requisito a cumplir es que quede claro para cualquier persona lo que hay que hacer en cada paso y el orden en el que se deben dar, sin ambigüedades.

Dicho de otra forma, **el lenguaje algorítmico es independiente del lenguaje de programación**, y por tanto no tiene porqué atenerse a una sintaxis determinada, por lo que cada persona puede expresarlo de forma distinta, siempre y cuando cumpla con las **dos reglas**:

- Sea fácil de comprender por cualquier persona.
- No sea ambiguo.

A continuación mostramos dos formas de representar el algoritmo para el cálculo del PVP de un producto. Se trata del mismo que se ha utilizado en la demostración del apartado de verificación manual del algoritmo.

Estas dos representaciones del algoritmo nos pueden llevar al mismo código en el lenguaje de programación elegido. Por tanto la codificación, por ejemplo en Java, va a consistir en la traducción de cada uno de los pasos de una de estas representaciones de la solución, a instrucciones del lenguaje, obteniendo como resultado un programa que posteriormente se escribirá en el ordenador para su compilación y ejecución.



I. 2 Dos formas de representar una solución al problema.

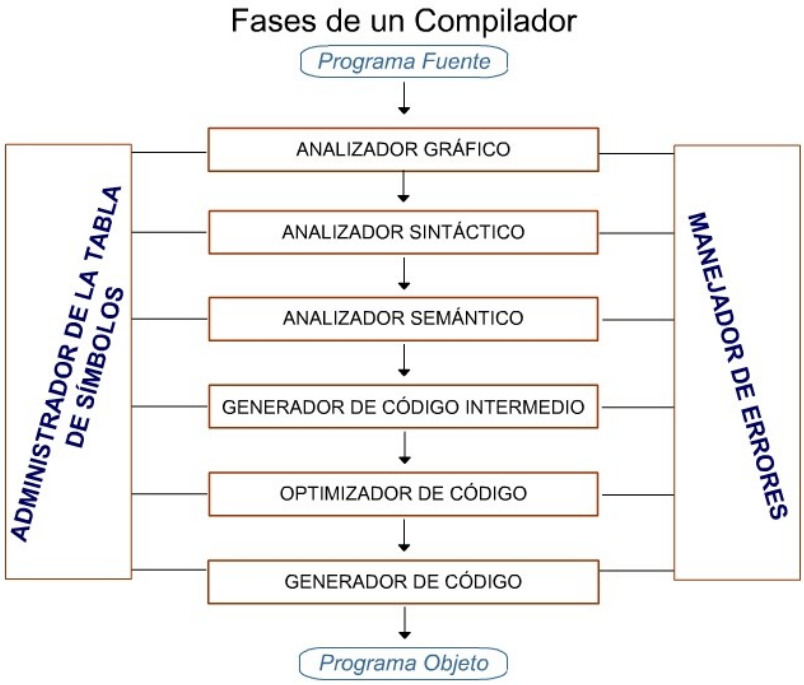
Traducción a código máquina.

Todos los lenguajes de programación que se usan normalmente son llamados **lenguajes de alto nivel,** en alusión a su parecido con nuestra forma de pensar y de expresarnos, aunque con una sintaxis mucho más rígida, que no de lugar a posibles interpretaciones, y que defina de forma clara lo que el ordenador debe hacer en cada momento. Incluso para construir las sentencias usan un buen número de palabras de nuestro idioma (aunque ese idioma es normalmente inglés). **Los lenguajes de alto nivel tienen como características básicas que son más fáciles de comprender y su portabilidad, o independencia del tipo de ordenador en el que se vayan a ejecutar los programas**. No obstante, los ordenadores funcionan con corriente, y eso es lo único que pueden entender. Concretamente, funcionan con sólo dos valores de corriente distintos y bien diferenciados (Por Ejemplo: En una línea de un circuito, en un momento dado hay una tensión de +5V o una tensión de -5V, y ningún otro valor es posible). **A esos dos valores distintos los identificamos como 0 y 1**, y decimos que los ordenadores son binarios, o que trabajan en binario. Pues bien, cualquier instrucción que queramos que ejecute el ordenador, deberemos expresársela en forma de ceros y unos, que es lo único que entiende. Ese lenguaje es lo que llamamos **lenguaje máquina**, y decimos que es un **lenguaje de bajo nivel**, ya que está alejado de nuestra forma de pensar y expresarnos.

Pero, ¿quién realiza esa traducción de lenguaje de alto nivel a código máquina? La respuesta es que **la realizan automáticamente unos programas llamados Traductores**. Existen dos tipos de traductores, que son los **compiladores** y los **intérpretes.**

Compiladores.

La analogía puede establecerse con el traductor que traduce un libro de inglés a español, produciendo un ejemplar escrito en español, que puede leerse tantas veces como se quiera sin tener que volver a traducirlo.



Un compilador es un programa traductor al que se le pasa como dato de entrada el fichero de texto con el programa escrito en lenguaje de alto nivel, al que se llama **código fuente** (por ejemplo, fichero con el programa escrito en lenguaje C). El compilador analiza todo el texto, y si encuentra errores, nos da un listado completo de los errores encontrados, para que editemos el código fuente y los corrijamos. Este proceso se repetirá mientras el programa contenga errores, se denomina **depuración de errores**, Si por el contrario no hay errores, **efectúa la traducción de todo el programa a código máquina, proporcionando el resultado grabado en un fichero al que se le denomina código objeto, y que es ejecutable tantas veces como se desee, sin tener que volver a traducirlo**, por lo que se ejecutará más rápidamente.

En la imagen podemos ver las fases de un compilador. Aunque en ella apreciamos que se trata de un proceso complejo, como programadores no necesitamos entrar en tantos detalles. Nos basta saber que **los compiladores traducen todo el programa escrito en un determinado lenguaje de alto nivel**, obteniéndose como resultado un fichero con el programa traducido a código máquina que es ejecutable sin tener que volver a traducir.

AQUÍ PLE1_RECURSO2.SWF

Tienen la desventaja de que en el proceso de depuración de los errores, al intentar analizar y traducir todo el texto y generar un listado con todos los errores encontrados, a veces se incluyen en ese listado errores que no son reales o mensajes de error poco claros, que se deben a que un error previo hace que el compilador no termine de entender correctamente el código que hay detrás de ese error. Pulse sobre la imagen para ejecutar una aplicación con ejemplos de aclaración de errores detectados por el compilador.

En el caso particular del lenguaje **Java**, no se trata de un lenguaje compilado, si no pseudocompilado, ya que al compilar ***no*** traduce directamente a código máquina, sino a un código intermedio que denomina **Bytecodes**, y que sigue siendo portable, independiente de la máquina, es decir, independiente del microprocesador concreto que tenga el ordenador en el que se ejecute.

PARA SABER MÁS:

En los siguientes enlaces encontrarás información algo más detallada acerca de las características de los compiladores y de los lenguajes compilados.

Lenguajes Compilados
http://es.wikipedia.org/wiki/Lenguajes_compilados

En este enlace aparece información exhaustiva sobre el funcionamiento de los compiladores.

Funcionamiento de Compiladores
http://www.monografias.com/trabajos11/compil/compil.shtml

Intérpretes.

Puede establecerse una analogía de los programas Intérpretes con el traductor simultáneo, al que se le da un libro en inglés para que vaya leyendo en español lo que en él dice. Después de haberlo leído, si quiero volver a escuchar lo que dice el libro en español, tengo que volver a pedirle al traductor que venga a leerlo.

Un intérprete es un programa traductor al que se le pasa como dato de entrada el fichero de texto con el programa escrito den lenguaje de alto nivel (**código fuente**) pero en vez de hacerlo de golpe y producir una versión ejecutable en **código máquina**, lo que hace es leer una a una las instrucciones del programa, analizarla, y si contiene errores parar el proceso y notificarlo para editar esa instrucción en el código fuente. Si no hay errores, **la traduce a código máquina en memoria, y la ejecuta**, sin guardar la traducción en ningún sitio. Por tanto, si quiero volver a ejecutar de nuevo el programa, deberé volver a traducir una a una cada línea antes de ejecutarla, por lo que la ejecución será más lenta. Por el contrario, el tiempo invertido en la depuración suele ser menor, ya que los mensajes de error suelen ser más precisos y estar más localizados.

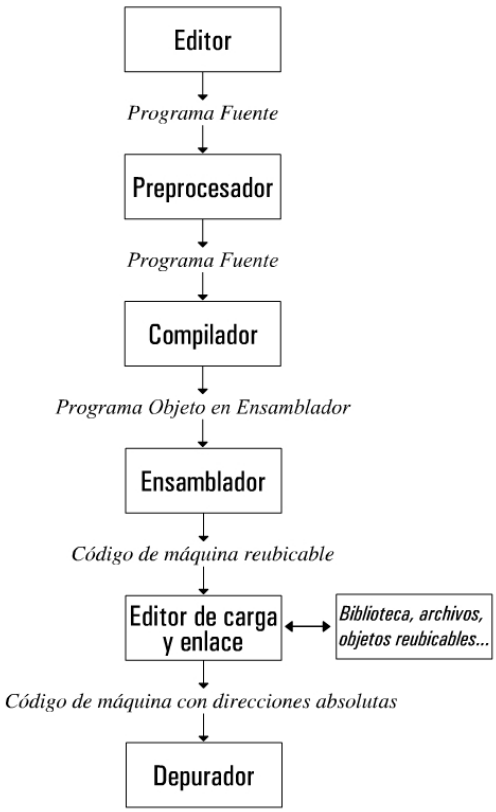
PARA SABER MÁS:

El siguiente enlace ofrece información adicional sobre las características de los intérpretes y los lenguajes interpretados.

Lenguajes Interpretados
http://es.wikipedia.org/wiki/Lenguajes_interpretados

Depuración.

Programas relacionados con un compilador



Ya tenemos el programa escrito en el lenguaje de programación, pero como somos humanos, nos hemos podido confundir, por ejemplo:

- olvidando un punto y coma al final de una sentencia,
- no cerrando un paréntesis,
- olvidando escribir una sentencia,
- escribiéndola mal, etc.

Todos esos errores deberán ser corregidos para que el programa pueda ser ejecutado por el ordenador. El proceso de depuración incluye la realización de un ciclo de tareas, hasta que el resultado es correcto. **Ése ciclo consiste fundamentalmente en:**

1. **compilar el programa**
2. **analizar el listado de errores**
3. **editar y corregir el texto del programa**
4. **volver a compilar el programa.**

Para entender ese proceso hay que detenerse un momento a explicar el significado de la palabra **compilar.** Mira la imagen para entender el camino que sigue un programa durante la compilación.

Pruebas.

Una vez finalizada la depuración, dispondremos de la posibilidad de ejecutar el programa. Ahora bien, es posible que aunque el programa funcione no haga exactamente lo que se necesita, o que funcione mal para algún dato de entrada, etc.

Será por ello necesario probar el funcionamiento del programa con un conjunto de datos lo suficientemente significativos, incluyendo valores extremos o raros. También debemos asegurarnos de haber probado toda la casuística prevista en el programa, de forma que no queden bloques de código sin someter a su oportuno test de prueba. **Cualquier fallo detectado nos puede llevar a replantear cualquiera de las etapas anteriores**, incluyendo el análisis o el diseño del algoritmo, o la codificación. No obstante, lo más frecuente es que se trate de un error en la etapa inmediata anterior.

También debemos señalar que los tests de pruebas deben diseñarse como parte de las especificaciones del análisis (en la fase de resolución), y no después de haber desarrollado la aplicación. A ser posible, incluso por otra persona que no haya sido el que ha desarrollado la aplicación. El motivo es que cuando nosotros desarrollamos y codificamos una aplicación en parte estamos viciados incluso inconscientemente por nuestro conocimiento de su funcionamiento interno, y tendemos de forma involuntaria a probarla sólo con los valores para los cuales la hemos pensado y para los casos en los que sabemos que funciona. Si el conjunto de pruebas se elaboró previamente, y por otra persona, tal circunstancia no es posible.

Fase de Explotación y Mantenimiento.

Una vez que la aplicación ya ha sido probada con éxito es el momento de instalarla para ponerla en marcha. Esta es la **fase de explotación** de la aplicación, y suele ser la más larga en el tiempo, y la que mayor coste supone a lo largo de la vida útil de la aplicación. A lo largo de la misma, será necesario ir haciendo algunas **tareas de mantenimiento de la aplicación**, para corregir errores, introducir mejoras, adaptaciones, etc.

En las grandes empresas de desarrollo del software, suele utilizarse un equipo de personas especializado en esta fase que además recoge todos los problemas encontrados en un documento de evaluación de la aplicación. Una vez concluida esta fase comprobando que la aplicación hace lo que debe sin errores y sin incoherencias es cuando se da por finalizado el servicio.

Ajustes.

Incluso después de haber superado con éxito todas las pruebas realizadas, no es infrecuente que una vez que la aplicación está funcionando, aparezcan pequeños fallos en casos en los que a nadie se le ocurrió pensar. O incluso funcionamientos que inicialmente no se consideraron erróneos, pero que sí tienen esa consideración por parte de los usuarios de la aplicación una vez que la tienen operativa y requieren que el funcionamiento sea diferente al previsto. Por ejemplo suele ocurrir que a la hora de imprimir la nómina de un trabajador, por el tratamiento manual de los documentos, se necesiten varias copias y así se incluye en los requerimientos hechos por el cliente. Pero ahora, una vez que tenemos la aplicación funcionando en red, cada uno de los departamentos implicados puede acceder al documento original e imprimirlo si lo necesita, así que no tiene sentido sacar varias copias de un documento si no es necesario y eso se va a decidir en cada departamento.

También es muy frecuente la distribución en el mercado de versiones Beta, para que los usuarios las prueben a fondo, descubriendo fallos ocultos, que deben ser corregidos.

Esto nos llevará a **replantear la aplicación desde la fase que sea necesaria**, y que normalmente será la codificación, pero que puede ser incluso el análisis, hasta obtener una nueva versión corregida y someterla de nuevo a un conjunto de pruebas específicas para probar el correcto funcionamiento donde antes fallaba.

Mejoras.

Aunque el funcionamiento de la aplicación sea correcto, y no se hayan detectado errores, habrá situaciones en las que se descubrirá que es posible realizar la misma tarea de forma distinta, de manera que se **aumente la rapidez, o se abaraten costes**, (**mejoras en la eficiencia**) o se consiga que el manejo sea **más intuitivo y fácil o más ameno** para los usuarios. En todos estos casos, deberán realizarse los cambios oportunos en la aplicación, siguiendo para ello el mismo planteamiento de revisión de todas las etapas anteriores que sean necesarias, incluidas las pruebas.

Adaptaciones.

Con el paso del tiempo, es frecuente que surjan **situaciones nuevas, no previstas inicialmente, que requieran incluir funcionalidad nueva en nuestra aplicación**, o que se produzcan cambios normativos que obliguen a que la aplicación se adapte a los mismos modificando aquellas partes que se vean afectadas por los cambios. Incluso esos cambios pueden venir impuestos más por gustos que por exigencias objetivas reales, por ejemplo casos en los que se prefiere que la nómina se haga sobre un documento preimpreso de forma que la aplicación complete espacios reservados de un documento previamente diseñado y no imprima todo el contenido del mismo. Esto puede suponer un ahorro de tinta o tóner de impresora importante, y si el documento tiene un formato menor, ahorrará papel.

En todos los casos, si se decide adaptar la aplicación habrá que replantear las partes de las fases anteriores (normalmente desde el análisis).