MÓDULO: PROGRAMACIÓN DE INTELIGENCIA ARTIFICIAL

Unidad 1.

LENGUAJES DE PROGRAMACIÓN

1. PROGRAMACIÓN: ELEMENTOS BÁSICOS

1.1. ALGORITMOS Y PROGRAMAS

Un **algoritmo** es por definición un «*conjunto ordenado y finito de operaciones que permite hallar la solución de un problema*». En realidad, los algoritmos están por todas partes: una receta, la puesta en marcha de un electrodoméstico, el cambio de aceite de un automóvil o el posicionamiento de páginas web en los buscadores.

Un algoritmo ha de ser **preciso**: debe dejar muy claro el orden en el que se han de ejecutar las operaciones o instrucciones que lo conforman.

Un algoritmo ha de estar bien **definido**: si se ejecuta *n* ocasiones en la misma situación inicial, ha de obtener siempre el mismo resultado.

El número **finito** de operaciones, instrucciones o reglas, obliga a que el algoritmo finalice en algún momento.

> ***Un algoritmo nos ayuda a resolver un problema de forma sistemática e inequívoca***

Podemos expresar un algoritmo de muchas maneras, incluyendo el lenguaje natural, pseudocódigo, diagramas de flujo o de cajas, y por supuesto, en lenguajes de programación reales.

Nombre de la variable	Descripción	Tipo
NP	Número de personas	Entero
TOT	Total que se va a pagar por el banquete	Real

```
1. Inicio
2. Leer NP
3. Si NP > 300
      Entonces
         Hacer TOT = NP * 75
      Si no
         Si NP > 200
            Entonces
               Hacer TOT = NP * 85
            Si no
               Hacer TOT = NP * 95
         Fin compara
   Fin compara
4. Escribir "El total es", TOT
5. Fin
```

Pseudocódigo

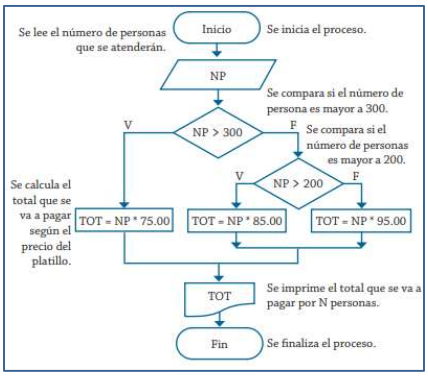


Diagrama de flujo

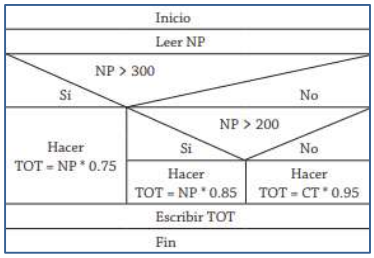


Diagrama de cajas

Un **programa** se asemeja a un algoritmo dado que es un «*conjunto de instrucciones que se introducen en una máquina para que esta lleve a cabo una determinada tarea*». De hecho, podemos considerar un programa como una implementación de un algoritmo.

Las instrucciones han de ser traducidas a un idioma que la computadora pueda aceptar. El **lenguaje** es la palabra clave.

1.2. LENGUAJES DE PROGRAMACIÓN

Un **lenguaje** es un medio (y una herramienta) para expresarse y comunicarse con los demás. Hay muchos lenguajes a nuestro alrededor. Podemos decir que cada lenguaje consta de los siguientes elementos:

- **Alfabeto:** conjunto de símbolos utilizados para formar palabras de un determinado lenguaje (alfabeto latino para el castellano, alfabeto cirílico para el ruso, kanji para el japonés, etc.)
- **Léxico o** diccionario: conjunto de palabras que el lenguaje ofrece a sus usuarios. Por ejemplo, la palabra "*computer*" proviene del diccionario en inglés, mientras que "*cmoptrue*" no; la palabra "*chat*" está presente en los diccionarios de inglés y francés, pero sus significados son diferentes.
- **Sintaxis:** conjunto de reglas que determinan si una cadena de palabras forma una oración válida. Por ejemplo, "Subo las escaleras" es una frase sintácticamente correcta, mientras que "Escaleras subo las" no lo es.
- **Semántica:** conjunto de reglas que deciden si una frase sintácticamente correcta tiene sentido. Por ejemplo, "Tengo hombro, vamos a comer" no tiene sentido.

Las computadoras también tienen su propio lenguaje, llamado **lenguaje máquina**, que es muy rudimentario. El *alfabeto del lenguaje máquina* lo constituyen los unos y los ceros. *El léxico o diccionario del lenguaje máquina* lo constituyen un conjunto de comandos llamado **lista de instrucciones** (**IL**) que la computadora reconoce y puede entender. Cada tipo de procesador tiene su IL particular; incluso las instrucciones pueden ser completamente diferentes en distintos modelos.

La programación directa en código máquina (el único lenguaje que realmente entiende una computadora) solo se usó en los primeros ordenadores. Los lenguajes que ahora se emplean para escribir programas se llaman **lenguajes de programación**. Son lenguajes artificiales creados para facilitar la elaboración de programas mucho más simples que el lenguaje natural y más complejos que el lenguaje máquina.

Oct 2025	Oct 2024	Change	Programming Language	Ratings	Change
1	1		Python	24.45%	+2.55%
2	4	↑	C	9.29%	+0.91%
3	2	↓	C++	8.84%	-2.77%
4	3	↓	Java	8.35%	-2.15%
5	5		C#	6.94%	+1.32%
6	6		JavaScript	3.41%	-0.13%
7	7		Visual Basic	3.22%	+0.87%
8	8		Go	1.92%	-0.10%
9	10	↑	Delphi/Object Pascal	1.86%	+0.19%
10	11	↑	SQL	1.77%	+0.13%
11	9	↓	Fortran	1.70%	-0.10%
12	29	⇈	Perl	1.66%	+1.10%
13	17	⇈	R	1.52%	+0.43%
14	15	↑	PHP	1.38%	+0.17%
15	16	↑	Assembly language	1.20%	+0.07%
16	13	↓	Rust	1.19%	-0.25%
17	12	⇊	MATLAB	1.16%	-0.32%
18	14	⇊	Scratch	1.15%	-0.26%
19	24	⇈	Ada	0.98%	+0.25%
20	21	↑	Kotlin	0.98%	+0.01%

Indice Tiobe: Lenguajes de Programación más utilizados en 2025

```
    xor r0, r0, r0
    movl r1, 100
    movh r1, 0

inicio_for:
    cmp r0, r1
    brnc fin_for;

    ; cuerpo del for
    movl r2, 00h
    movh r2, 10h
    add r2, r2, r0

    mov  r3, [r2]

    movl r4, 25
    movh r4, 0

    sub r3, r3, r4
    mov [r2], r3

    inc r0
    jmp inicio_for
fin_for:
```

```
      program compute_pi
      integer n, i
      real*8 w, x, pi, f, a
      f(a) = 4.d0 /(1.d0 + a*a)  !! function to integrate

      pi = 0.0d0
!$OMP PARALLEL private(x, w, n), shared(pi)
      n = 10000                  !! number of intervals
      w = 1.0d0/n                !! calculate the interval size
!$OMP DO reduction(+: pi)
      do i = 1, n
        x = w * (i - 0.5d0)
        pi = pi + f(x)
      enddo
!$OMP END DO
!$OMP END PARALLEL
      print *, "Computed pi = ", pi
      end
```

```
(defun factorial (n)
  (if (<= n 1)
      1
      (* n (factorial (1- n)))))
```

```
CREATE TABLE clientes (
    id INT PRIMARY KEY,
    nombre VARCHAR(50),
    edad INT,
    email VARCHAR(100)
);
INSERT INTO clientes (id, nombre, edad, email) VALUES
(1, 'Juan Pérez', 30, 'juan.perez@example.com'),
(2, 'María López', 25, 'maria.lopez@example.com'),
(3, 'Carlos Gómez', 35, 'carlos.gomez@example.com'),
(4, 'Ana Martínez', 28, 'ana.martinez@example.com');
```

Diferentes lenguajes de programación: ensamblador, FORTRAN, LISP, SQL

1.3. COMPILACIÓN E INTERPRETACIÓN

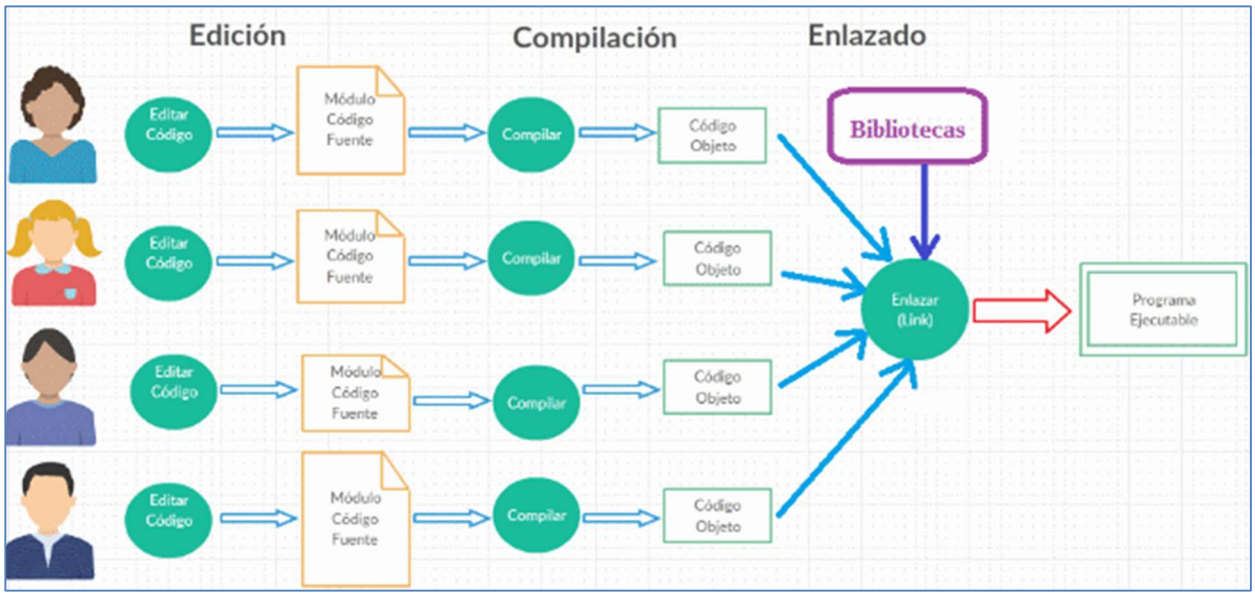
Código fuente es el código escrito por los programadores mediante algún editor de texto o alguna herramienta de programación utilizando un lenguaje de programación apropiado al problema que se trate. Como el código fuente no es comprensible por la máquina, habrá que traducirlo a lenguaje máquina que la computadora pueda entender y ejecutar.

Sólo se genera código ejecutable si el código fuente está libre de errores léxicos, sintácticos y semánticos.

El proceso de traducción puede realizarse de dos formas:

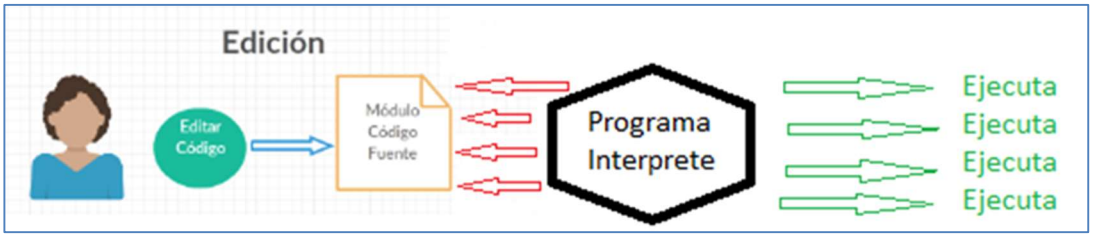
- **COMPILACIÓN**.

El proceso de compilación de un programa se lleva a cabo mediante dos programas: el compilador y el enlazador. El **compilador** traduce el código fuente (siempre que esté libre de errores sintácticos) en **código objeto**. El **enlazador** (linker) inserta en el código objeto las funciones de librería necesarias para producir el programa ejecutable que contiene el código máquina. Hecho esto ya tendremos un programa autónomo totalmente ejecutable para una plataforma concreta que podremos distribuir.



INTERPRETACIÓN.

Un programa **intérprete** realiza la traducción del código fuente, línea a línea, cada vez que se quiere ejecutar. *No se crea un código objeto intermedio*. Siempre que quiera ejecutar el programa necesitaremos el código fuente y el intérprete, es decir, no tenemos un programa autónomo ejecutable por sí solo. El proceso de traducción y ejecución hace que el programa se ejecute de forma más lenta, los errores aparecen en el momento en que se ejecute la instrucción problemática.



Compilador	***Intérprete***
Genera un archivo directamente ejecutable.	No genera un archivo directamente ejecutable. El usuario final debe disponer del intérprete para ejecutar el código.
El proceso de traducción se realiza una sola vez. El proceso de traducción en sí puede consumir mucho tiempo.	El proceso de traducción se realiza en cada ejecución.
La ejecución es muy rápida debido a que el programa ya ha sido traducido a código máquina.	La ejecución es más lenta, ya que para cada línea del programa es necesario realizar la traducción y posterior ejecución. Además, el intérprete y el código comparten los recursos de la máquina.
El ejecutable va dirigido a un procesador y un SO concreto. Hay que tener tantos compiladores como plataformas de hardware donde se desea ejecutar el código.	No hay ejecutable, así que si existe un intérprete para una plataforma concreta, el programa se podrá ejecutar en ambas. Suelen ser más portables que los compilados.
Una vez compilado el programa, el código fuente ya no es necesario para ejecutarlo, así que puede permanecer «en secreto» si se desea.	El código fuente es necesario en cada ejecución, así que no puede permanecer «en secreto».
Los errores sintácticos se detectan durante la compilación. Si el código fuente contiene errores sintácticos, el compilador no producirá un ejecutable.	Los errores sintácticos se detectan durante la ejecución, ya que traducción y ejecución se hacen simultáneamente. Algún error sintáctico podría quedar enmascarado, si para una ejecución concreta no es necesario traducir la línea que lo contiene. (Algunos intérpretes son capaces de evitar esto)
Un error grave en un programa compilado puede afectar seriamente a la estabilidad de la plataforma, llegando incluso a colgar el equipo.	Un programa interpretado con un comportamiento torpe normalmente puede ser interrumpido sin dificultad, ya que su ejecución está bajo el control del intérprete, y no solo del sistema operativo.

Cuadro comparativo Compilación e Interpretación

1.4. EVOLUCIÓN DE LOS LENGUAJES DE PROGRAMACIÓN.

⇨ **1ª GENERACIÓN**. Aparecen entre los años 1954 y 1958, lenguajes como FORTRAN y ALGOL. En esta etapa los programas tienen una sola línea principal de ejecución. En estos lenguajes no hay separación clara entre datos y programas y como mecanismo de reutilización de código se propone la biblioteca de funciones.

⇨ **2ª GENERACIÓN.** Entre 1959 y 1961 surgen FORTRAN II, COBOL, LISP. En todos estos lenguajes, aunque ya existe separación entre datos y programas, el acceso a los datos es bastante desordenado, y no proporcionan mecanismos para preservar la integridad de los datos (como podría ser una distinción variables locales o globales o paso de parámetros).

⇨ **3ª GENERACIÓN.** Entre 1962 y 1975 nacen multitud de lenguajes: PASCAL, C, Simula. Aparecen los conceptos de programación estructurada basada en el teorema de Dijkstra —cualquier programa puede escribirse con instrucciones secuenciales, condicionales e iterativas— y la abstracción de datos, que consiste en la definición de tipos complejos de datos y su asociación a operadores para tratarlos. Sin embargo no formalizan mecanismos de protección de acceso a los datos. Tampoco añaden ningún mecanismo para la reutilización de código diferente a la biblioteca de funciones.

⇨ **4ª GENERACIÓN.** En las décadas de 1980 y 1990 aparecen los lenguajes orientados a objetos (C, Smalltalk, Java). Incorporan mecanismos que permiten restringir el acceso a los datos que forman parte de los objetos. Es responsabilidad de cada objeto el mantenimiento de sus datos y la interacción entre objetos se hace a través de una interfaz bien definida. Aparece en estos lenguajes también el concepto de herencia que permite la reutilización de código.

Casi todos los lenguajes evolucionan desde sus orígenes. Por ejemplo, las versiones más actuales de Cobol o Pascal incorporan características de orientación a objetos. Destaca una tendencia hacia los lenguajes más visuales donde se utilizan muchas herramientas gráficas.

1.5. CLASIFICACIÓN DE LOS LENGUAJES DE PROGRAMACIÓN

1.6. SEGÚN EL NIVEL DE ABSTRACCIÓN

⇨ **LENGUAJES DE BAJO NIVEL.**

- El **lenguaje máquina** fue el primer lenguaje utilizado. Todas las instrucciones son cadenas de 0s y 1s que especifican las operaciones a realizar y los datos con los que se trabaja. Es el único lenguaje que no necesita traducción. No es portable, solo se puede ejecutar en procesadores que dispongan de la misma lista de instrucciones con que se ha implementado el programa.
- El **lenguaje ensamblador** sustituyó al lenguaje máquina para facilitar la programación. En lugar de los códigos numéricos representativos de cada instrucción, se utilizan mnemotécnicos que recuerdan su función (sum, mov, jmp, …). Los programas necesitan ser traducidos a lenguaje máquina, pero la traducción es casi inmediata. Sin embargo, aún se trata de código no portable, y obliga a los programadores a dominar, no solo el lenguaje, sino también el arquitectura interna del procesador —posiciones de memoria, registros, etc.—.

⇨ LENGUAJES DE ALTO NIVEL.

Se crearon con el objetivo de conseguir programas portables, independientes de la plataforma o máquina sobre la que se ejecuta. Se acercan más al lenguaje humano, y sustituyen los mnemotécnicos del ensamblador por palabras, habitualmente en inglés, lo que redujo considerablemente la curva de aprendizaje. Necesitan una traducción a lenguaje máquina para poder ejecutarse, ya sea compilación o interpretación.

Según el modelo o paradigma de programación que utilice para desarrollar programas, un lenguaje de alto nivel se puede clasificar como:

- ✓ **Orientado al procedimiento**: se expresa de manera imperativa en forma de algoritmos. Aporta ventajas cuando son tareas sencillas que se pueden describir con pocos pasos. Por ejemplo C y Fortran
- ✓ **Orientados a funciones:** basado en el concepto matemático de funciones, se expresa de manera declarativa. Permite construir programas concisos, fáciles de probar, en los que se requiera un alto grado de fiabilidad. Por ejemplo Lisp, SML, Hasckel.
- ✓ **Orientado a la lógica:** se expresa por metas en forma de cálculos de predicados. Utilizan reglas e inferencias lógicas. Facilita la implementación de sistemas expertos en los que se deba manejar una base de conocimiento. Por ejemplo Prolog
- ✓ **Orientado a objetos:** se expresa en forma de relaciones entre objetos. Es útil en una amplia variedad de problemas, incluso como marco de alto nivel para integrar sistemas desarrollados siguiendo diferentes paradigmas. Por ejemplo C++, Java.

Esta clasificación no es estanca, algunos lenguajes pertenecen a varios paradigmas. Existen paradigmas más generales como el **imperativo** o el **estructurado** que engloban a otros. Además continuamente aparecen nuevos paradigmas, como la **orientación a aspectos**.

En general, con mayor o menor dificultad, se puede usar cualquier lenguaje de programación de alto nivel para cualquier paradigma

1.7. SEGÚN EL PROPÓSITO

- **Lenguajes de propósito general**: aptos para todo tipo de tareas, por ejemplo Java.
- **Lenguajes de propósito específico**: hechos para un objetivo muy concreto; por ejemplo el lenguaje CSound está pensado para la creación de música.
- **Lenguajes de programación de sistemas**: diseñados para implementar sistemas operativos o drivers, por ejemplo el C.
- **Lenguajes de script**: adecuados para la automatización y ejecución de tareas específicas. Por ejemplo JavaScript.

2. DESARROLLO DE SOFTWARE.

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

El proceso de desarrollo, consta de una serie de pasos de obligado cumplimiento, pues solo así podremos garantizar que los programas creados sean eficientes, fiables, seguros y respondan a las necesidades de los usuarios finales (aquellos que van a utilizar el programa).

Las fases principales por las que pasa el sistema que se está desarrollando desde que nace la idea inicial hasta que el software es retirado o reemplazado por otro más adecuado son: análisis, diseño, codificación, pruebas y mantenimiento.

2.1. ANÁLISIS

Esta es la primera fase del proyecto y por no estar automatizada, es la más complicada: depende en gran medida de la pericia del analista que la realice. Las necesidades del cliente deben quedar claramente identificadas desde todos los puntos de vista.

En la fase de análisis todos los requisitos del sistema, funcionales y no funcionales, deben quedar claramente recogidos en el documento **Especificación de Requisitos Software** (**ERS**).

2.2. DISEÑO

Una vez identificados los requisitos es necesario establecer la forma en que se solucionará el problema: Se deducen entre otras cosas las estructuras de datos, la interfaz de usuario y los procedimientos; se selecciona el lenguaje de programación, el Sistema Gestor de Base de Datos, etc. El desarrollo del algoritmo que va a permitirnos resolver el problema planteado se realiza en esta fase, y es fundamental para que el programa funcione correctamente.

En aplicaciones grandes, para enfrentarnos a la complejidad del software se recurre a la descomposición del problema en problemas menores e independientes. Desde el punto de vista de la programación se puede hablar de dos formas de descomposición:

- **Descomposición algorítmica**: el problema se descompone en tareas más simples, cada una de estas tareas se divide a su vez en otras tareas más simples y así sucesivamente. Esta técnica aporta ventajas cuando el problema a resolver puede considerarse “pequeño”
- **Descomposición orientada a objetos**: el problema se descompone en objetos de cuya interacción surge la solución. Esta técnica aporta ventajas cuando se aplica a proyectos grandes: facilita la reusabilidad de elementos y se adapta mejor a los cambios.

2.3. CODIFICACIÓN

La codificación del programa no es más que la traducción del algoritmo a un lenguaje de programación en concreto, lo cual nos da como resultado un conjunto de instrucciones almacenadas en un fichero llamado **código fuente**. Es una fase relativamente sencilla, dado que el núcleo del trabajo, la obtención del algoritmo, ya está hecha.

En esta fase se incluye el **montaje** o **linkado**. El linkado permite enlazar el programa con las librerías necesarias para que las instrucciones del código funcionen correctamente, y crear con ello el fichero ejecutable.

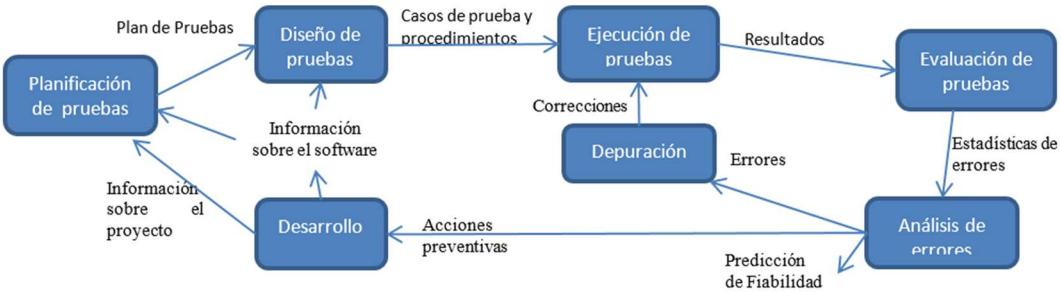
Las librerías son archivos que contienen código, datos y en general recursos prefabricados que pueden ser usados por el programador y se utilizan para no tener que volver implementar funciones ya desarrolladas y probadas. Pueden ser creadas por el programador o bien instaladas desde otras fuentes. En los entornos de desarrollo modernos vienen incluidas muchas de estas librerías que permiten realizar una gran variedad de funciones.

2.4. PRUEBAS

Durante la prueba del software se realizarán tareas de verificación y validación del software (V&V).

- Pruebas de **verificación**. Conjunto de actividades que tratan de comprobar si se está construyendo el producto correctamente, es decir, si el software implementa correctamente la función para la que fue diseñado.
- Pruebas de **validación**. Conjunto de actividades que tratan de comprobar si el producto es correcto, es decir, si el software construido se ajusta a los requisitos del cliente.

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



En esta fase son especialmente importantes las pruebas de rendimiento o de carga, en las que se busca, entre otras cosas, intervalos de ineficiencia durante el tiempo de ejecución de la aplicación. Hay muchas herramientas para llevar a cabo estas pruebas: Apache JMeter, Tsung, Gatling, Siege, Locust, Taurus, etc.

2.5. MANTENIMIENTO

Esta fase tiene lugar después de la entrega del software al cliente. En ella hay que asegurar que el sistema pueda adaptarse a los cambios. Se producen cambios porque se han encontrado errores, es necesario adaptarse al entorno (por ejemplo se ha cambiado de sistema operativo) o porque el cliente requiera mejoras funcionales.

3. PROGRAMACIÓN ORIENTADA A OBJETOS. POO

La POO es un método de desarrollo en el que los programas se organizan como colecciones de objetos que cooperan para resolver un problema. Estos objetos pueden corresponderse con entidades del mundo real (un coche o un gato), acciones (acelerar o realizar una transacción bancaria) o *procesos* (viaje o aprendizaje).

> Un objeto es algo a lo que se le puede enviar ***mensajes*** y que puede responder a los mismos, que tiene un ***estado***, un ***comportamiento*** bien definido y una ***identidad***.

El ***estado*** de un objeto está definido por el valor de ciertas variables internas al objeto. Este estado puede cambiar dependiendo de los mensajes que reciba desde el exterior o de un cambio interno al propio objeto.

El ***comportamiento*** varía en función del estado en que se encuentra y se percibe por los valores que devuelve ante los mensajes que recibe y por los cambios que produce en los objetos con los que se relaciona.

Finalmente la ***identidad*** de un objeto es aquello que lo hace distinguible de otros objetos.

La programación orientada a objetos se basa en el ***Modelo de Objetos***. Este modelo se fundamenta en el uso de 7 capacidades, 4 de las cuales se consideran principales y 3 secundarias. Los lenguajes de programación orientados a objetos se caracterizan porque proporcionan mecanismos que dan soporte a estas capacidades.

Las capacidades principales son: Abstraer, Encapsular, Jerarquizar y Modularizar.
Las capacidades secundarias son: tipo, concurrencia y persistencia

3.1. CAPACIDAD DE ABSTRAER.

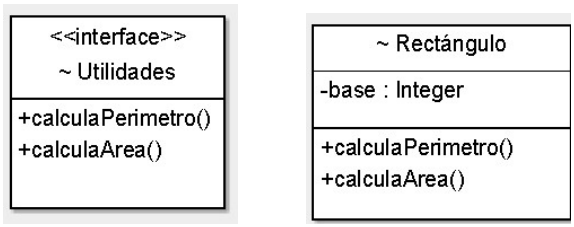
La abstracción surge de reconocer las similitudes entre objetos, situaciones o procesos en el mundo real y la decisión de concentrarse en esas similitudes (características y comportamientos) e ignorar las diferencias.

Los lenguajes de POO permiten abstraer ya que permiten definir ***interfaces*** comunes para comunicarse con conjuntos de objetos. Estas interfaces están compuestas por los ***métodos***, que son funciones que pueden aplicarse sobre el objeto y que *pueden entenderse como los mensajes que es posible enviar al objeto*. Al conjunto de mensajes a los que puede responder un objeto se le conoce como ***protocolo*** del objeto.

En algunos lenguajes de POO aparece el concepto de ***clase de objetos***, que une las interfaces definidas en el proceso de abstracción con la implementación del comportamiento deseado. Es decir, las clases *añaden a la definición de los métodos la implementación* de los mismos. También *añaden las* ***propiedades***, que son las características comunes a un conjunto de objetos: variables internas al objeto o a la clase, que definen el estado del objeto. ***Métodos*** y ***propiedades*** se conocen como miembros de la clase.

Los objetos que hay en un sistema siempre pertenecen a una determinada clase, que define su comportamiento, la manera de interactuar con él y sus posibles estados.

Además, existe un «***contrato***» que establece las responsabilidades de un objeto respecto a las operaciones que puede realizar, y que puede implicar unas ***precondiciones*** y ***postcondiciones*** que deben quedar satisfechas. Cuando al enviar un mensaje a un objeto se cumple las precondiciones pero el objeto no puede cumplir las postcondiciones se produce una ***excepción***. Los lenguajes orientados a objetos también suelen dar soporte al manejo de excepciones.



UML: representación de interface y Clase

3.2. CAPACIDAD DE ENCAPSULAR.

Permite mantener oculta la implementación de una abstracción (una interface o una clase) a los usuarios de la misma, para que ninguna parte de un sistema complejo dependa de cómo se ha implementado otra parte.

Para facilitar la encapsulación los lenguajes de POO ofrecen mecanismos como los modificadores de visibilidad, que se aplican a las propiedades y a los métodos.

(+) public: se puede acceder libremente a unas partes de la clase

(-) private: se prohíbe el acceso a unas partes de la clase.

(#) protected: se puede acceder a unas partes de la clase pero con restricciones.

Como norma general a la hora de definir la visibilidad tendremos en cuenta que:

- El estado debe ser privado. Los atributos de una clase se deben modificar mediante métodos de la propia clase creados a tal efecto.
- Los métodos de la clase deben ser públicos, es la forma de interactuar con la clase desde otras partes.
- Las operaciones que ayudan a implementar los métodos deben ser privadas o protegidas.

3.3. CAPACIDAD DE JERARQUIZAR.

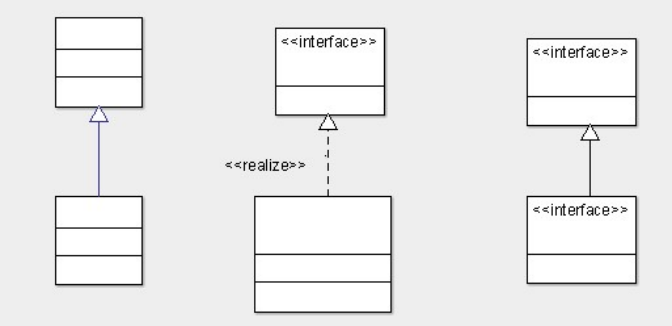
Jerarquizar es una capacidad que permite ordenar abstracciones para detectar estructuras y comportamientos comunes para simplificar la comprensión del problema y su desarrollo. En el esquema de POO se definen dos formas básicas de jerarquías:

- ***Jerarquía entre clases e interfaces***.

  Estas relaciones se denominan ***relaciones de herencia*** y cumplen que los elementos de los que se hereda son más ***generales***, y los elementos que heredan son más ***especializados***. Los lenguajes de POO proporcionan mecanismos para definir clases o interfaces nuevas, o que heredan de otras ya existentes.

- Herencia ***entre interfaces***. Como las interfaces no implementan métodos, la interface heredera obtiene la declaración de los métodos de la interface heredable.
- Herencia ***entre clase e interfaz***. Si una clase hereda de una interface, se dice que `implementa` la interface (implementa el método o métodos de la interface).
- Herencia ***entre clases***. Una clase que hereda de otra, hereda tanto su interface (métodos) como sus propiedades públicas y /o protegidas

Tras heredar siempre *se pueden añadir nuevos métodos* a la abstracción que hereda para especializarla. En el caso en el que la que hereda sea una clase, también se pueden añadir nuevas propiedades.



Por otro lado, atendiendo a la forma de las jerarquías, la herencia puede ser simple o múltiple.

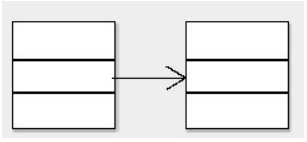
- Herencia ***simple***. Ocurre cuando un elemento sólo puede heredar de una jerarquía.
- Herencia ***múltiple***. Ocurre cuando un elemento hereda de varias jerarquías simultáneamente. Este tipo de herencia da lugar a lo que se llama *problema de la ambigüedad o del diamante*: una clase hereda de varias que tienen un método con idéntico nombre y parámetros pero que tiene definidos comportamientos diferentes en cada clase. La herencia múltiple de interfaz no genera este problema, ya que las interfaces no implementan los métodos.

El ***polimorfismo*** es la propiedad que permite emplear el mismo nombre para nombrar métodos de diferentes abstracciones. La consecuencia de esto es que los efectos serán diferentes dependiendo del método que se ejecute.

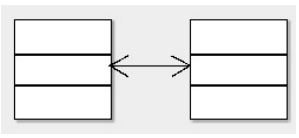
➢ *Jerarquía entre objetos*.

La jerarquía entre objetos se puede clasificar en dos tipos de relaciones: relaciones de asociación y relaciones de dependencia.

- ***Relaciones de asociación***. Permiten construir objetos mediante la asociación de otros objetos menores. Los lenguajes de POO facilitan estas relaciones permitiendo que cualquier clase se pueda utilizar para ***definir una propiedad*** dentro de una clase.

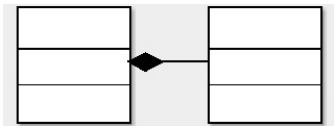


Asociación navegable en una dirección

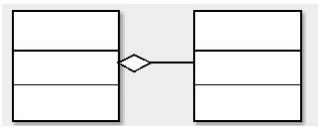


Asociación navegable en dos direcciones

Las relaciones de asociación pueden ser relaciones de **composición** (la existencia de los objetos menores está supeditada a la existencia del objeto mayor) o de **agregación** (en otro caso)

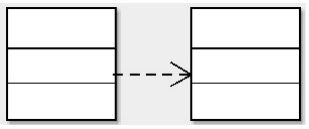


Composición



Agregación

- ***Relaciones de dependencia o de uso***. "Tal objeto usa tal objeto". Los lenguajes de POO facilitan estas relaciones permitiendo que un método pueda utilizar un objeto de manera local. Pero el ámbito y el tiempo de uso de un objeto desde otro es más limitado.



Dependencia

3.4. CAPACIDAD DE MODULARIZAR.

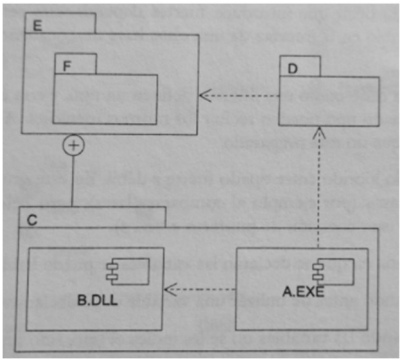
Modularizar es la capacidad para dividir un programa en agrupaciones lógicas de sentencias (módulos). Las ventajas que ofrece la modularidad son:

- Facilidad de mantenimiento, diseño y revisión.
- Aumento de la velocidad de compilación (los compiladores compilan por módulos)
- Mejora en la organización y en la reusabilidad.

A la hora de diseñar módulos debe tenerse en cuenta:

- Maximizar la ***coherencia***: agrupar en un módulo abstracciones relacionadas lógicamente.
- Minimizar las dependencias entre módulos: para compilar un módulo no se necesite compilar otros.
- Controlar el tamaño de los módulos: módulos pequeños aumentan la desorganización pero módulos muy grandes son menos manejables y aumentan tiempos de compilación.

Los conceptos de fichero, biblioteca, paquete o componente son resultados de la modularización.



El paquete D depende del paquete E y el componente A.EXE depende del componente B.DLL y del paquete D.

Por otro lado, los componentes A.EXE y B.DELL están contenidos en el paquete C que, igual que F, está contenido en el paquete E.

3.5. CAPACIDAD DE TIPADO.

En programación orientada a objetos los objetos que comparten una misma interface se dice que tienen el mismo tipo. La asociación del tipo a un objeto se conoce como tipado. El tipado impide que se confundan abstracciones diferentes y dificulta el uso de abstracciones de manera no prevista.

Los lenguajes de programación pueden clasificarse respecto a las restricciones que impone al tipo:

- *Lenguajes **con tipado fuerte*** en los que no es posible mezclar variables de tipos diferentes. Obliga a hacer conversiones de tipo previas; en ocasiones esta conversión puede no ser factible. Por ejemplo un entero puede convertirse en un real, pero una referencia no puede convertirse en un char.
- *Lenguajes **con tipado débil*** en los que es posible mezclar variables de diferentes tipos.

Por otro lado, dependiendo de la forma en la que se declaran las variables se puede hablar de:

- *Lenguajes **con tipado explícito*** cuando antes de utilizar una variable debe declararse el tipo al que pertenece.
- *Lenguajes **con tipado implícito*** cuando el tipo de una variable no se indica, si no que se deduce del código.

En cuanto al momento en que se comprueba el tipo de una variable, se pueden clasificar en:

- *Lenguajes **con tipado estático*** en los que el tipo se comprueba en compilación.
- *Lenguajes **con tipado dinámico*** en los que el tipo se comprueba en ejecución.

3.6. CAPACIDAD DE CONCURRENCIA.

La concurrencia es la capacidad que permite la ejecución paralela de varias secuencias de instrucciones. Por ejemplo, si se dispone de múltiples procesadores se pueden utilizar todos a la vez para resolver un problema.

Generalmente los lenguajes de programación no dan soporte a la concurrencia, sino que esta facilidad es proporcionada por los sistemas operativos: dividen la línea de ejecución, creando múltiples líneas de ejecución, también llamadas hilos o threads.

3.7. CAPACIDAD DE PERSISTENCIA.

La persistencia es la capacidad que permite que la existencia de los datos trascienda en el tiempo y en el espacio. Podemos clasificar los datos en relación a su vida según los siguientes tipos:

- Ámbito de los lenguajes de programación:
  - Expresiones: su vida no supera el ámbito de una línea de código.
  - Variables locales: su vida se supedita a la vida de una función
  - Variables globales: existen mientras se ejecuta un programa
- Ámbito de las bases de datos:
  - Datos que persisten de una ejecución a otra.
  - Datos que sobreviven a una versión de un programa.

- Datos que sobreviven cuando ya no existe el programa, el SO o el ordenador en que se crearon.

Un lenguaje orientado a objetos que de soporte a la persistencia debe permitir grabar los objetos que existan, así como la definición de sus clases, de manera que puedan cargarse más adelante sin ambigüedad, incluso en otro programa distinto al que lo ha creado: existe una interface que define métodos que permiten almacenar los objetos en soportes permanentes.