

## Capítulo 4

# Desarrollo del software de gestión de sistemas

### Contenido

1. Introducción
2. Análisis de especificaciones para el desarrollo de software de gestión de sistemas
3. Técnicas de programación presentes en lenguajes de uso común aplicables al desarrollo de software de gestión de sistemas
4. Técnica de programación de software de gestión de sistemas
5. Control de calidad del desarrollo del software de gestión de sistemas
6. Herramientas de uso común para el desarrollo de software de sistemas
7. Resumen

## 1. Introducción

Ya se ha indicado en capítulos anteriores que no existe un modelo de desarrollo de software universal, que cada modelo tiene que ajustarse a las características y naturaleza del producto final.

En este capítulo, se empieza describiendo la etapa de análisis orientado a un modelo basado en componentes para un sistema. Se sabe que existen muchos modelos de desarrollo, pero quizás el modelo que mejor se ajusta a un desarrollo de software de sistemas es el

que permite descomponer dicho sistema en pequeños subsistemas que se relacionan y comunican entre sí. Esto es así debido a la complejidad de los procesos que deben abordar este tipo de software y que, de otra forma, pueden resultar en soluciones poco eficientes y que son difíciles de mantener.

A continuación, se definen los conceptos básicos y el tipo de programación en el que se ha apoyado a lo largo de los años este tipo de software: la programación estructurada y cómo ha sido sustituida por la programación orientada a objetos para resolver algunas de las técnicas más famosas de la computación.

También se introduce la disciplina del control de calidad del software y cómo se puede utilizar para mejorar el producto final de forma objetiva. La calidad ha sido un problema recurrente desde los inicios de la programación y es ahora cuando más esfuerzos se realizan por comprenderla y mejorarla.

Por último, se verán las diferentes herramientas que permiten llevar a cabo los conceptos e ideas estudiadas hasta ese momento, dando forma al concepto de software.

## 2. Análisis de especificaciones para el desarrollo de software de gestión de sistemas

Hoy en día, existe una necesidad de diseñar el software lo más rápido posible, acortando tanto el tiempo como el coste de desarrollo. La metodología orientada a objetos introdujo una nueva forma de desarrollo basada en la idea de clases y objetos que representaran términos y conceptos del mundo real. Esto mejoró enormemente los procesos de análisis y desarrollo, pero no era suficiente para encontrar un enfoque de reutilización del software. Aunque se podían encontrar entidades similares en varios proyectos de software, el uso de

estas entidades a través de una clase que la representara estaba determinado por un conocimiento detallado de la misma, lo que implicaba conocer el desarrollo interno de la clase para su uso en otros proyectos. La ingeniería del software basada en componentes surge como una aproximación a la idea de reutilización del software en el desarrollo y salva esta limitación de la anterior metodología.

Según Roger Pressman, la Ingeniería del Software Basada en Componentes (ISBC) es un proceso que concede particular importancia al diseño y la construcción de sistemas basados en computadoras que utilizan "componentes" de software reutilizables.

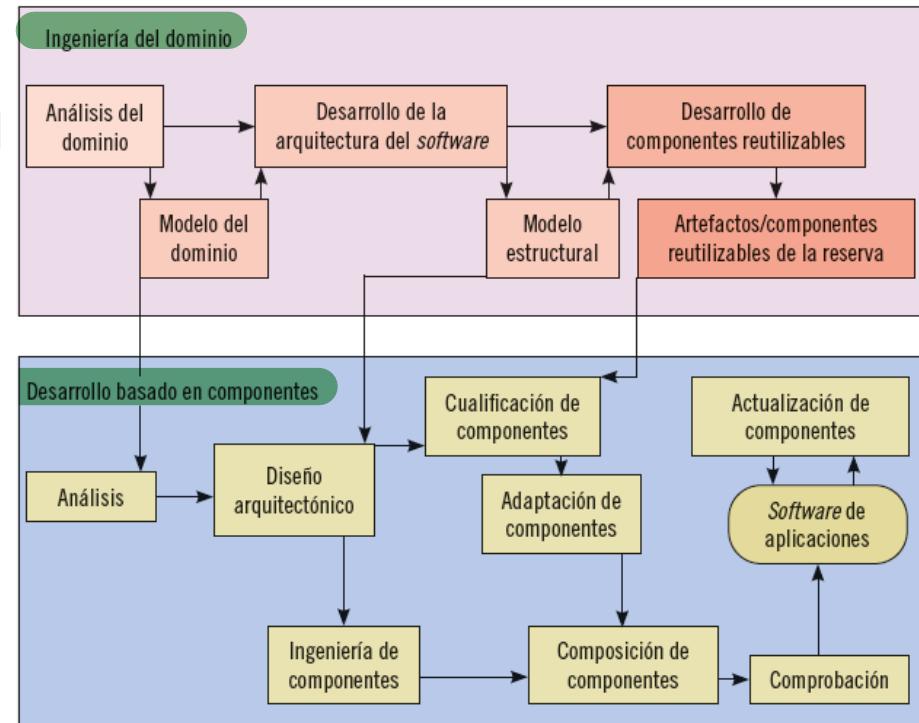
Y su objetivo es identificar, desarrollar, catalogar e integrar un grupo de componentes que resultan del análisis del dominio de aplicación, con el fin de que, en conjunto, sean capaces de alcanzar los objetivos del sistema y a la vez sirvan a otros proyectos como componentes reutilizables.



#### Recuerde

Un componente es la unidad principal de composición de aplicaciones de software cuyo desarrollo se basa en componentes. Ese componente debe de implementar algún requisito funcional o variar de la lista de requisitos obtenida en la fase de análisis.

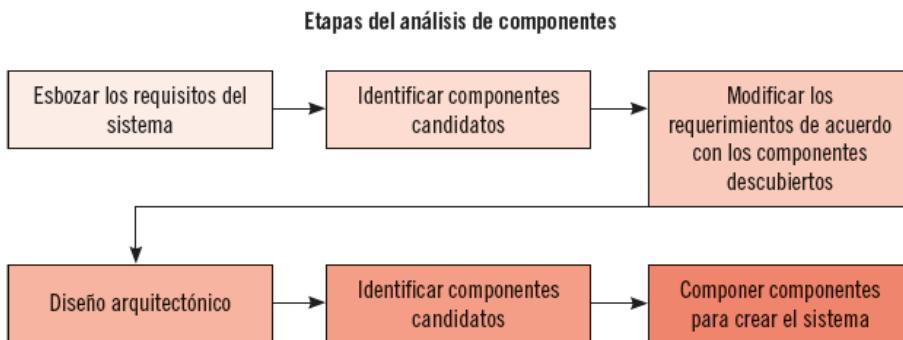
Fases que se producen en el análisis de sistemas con desarrollo basado en componentes



## 2.1. Identificación de los componentes necesarios según las especificaciones

La identificación de los componentes se lleva a cabo a través de la ingeniería del dominio. La ingeniería del dominio se encarga de definir el dominio del sistema, clasificar y categorizar los elementos extraídos, recopilar las aplicaciones del dominio, analizar cada aplicación y definir las clases, y, por último, desarrollar el modelo de análisis de clases.

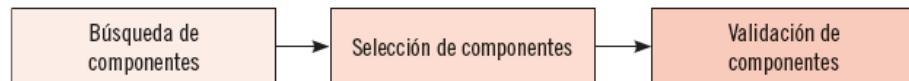
El modelo de análisis es el punto de origen para la identificación de componentes. Se analiza para determinar los requisitos que apuntan a componentes reutilizables. Sin embargo, debe existir una concordancia entre esos requisitos y las especificaciones. Esta concordancia puede producir componentes reales del sistema y, por lo tanto, el diseñador se encargaría de obtenerlos de la biblioteca de componentes reutilizables y añadirlos al sistema. Pero, si por el contrario, no existiera dicho componente, el diseñador debería aplicar los métodos de diseño convencionales para desarrollarlos.



Es importante que la definición de requisitos sea lo más flexible posible para evitar que el número de componentes para cada requisito sea reducido. Es decir, es necesaria la capacidad de identificar la mayor cantidad de componentes posibles que satisfacen los requisitos del sistema.

Tras obtener el diseño arquitectónico, también existe una fase de identificación de componentes. Esto es debido, principalmente, a que los componentes que se han identificado en las fases anteriores pueden no trabajar correctamente con otros componentes identificados. Para lo cual se necesita volver a replantear la selección de algunos de los componentes.

Etapas hasta la validación de componentes



A día de hoy, no existe un mercado de *software* al que uno se pueda dirigir para obtener los componentes que se han identificado para el sistema. La principal consecuencia es que la selección está restringida a la organización que desarrolla el *software*. Para ello, la organización debe formar su propia base de datos de componentes.

En la mayoría de los casos, seleccionar un componente de la base de datos será una tarea sencilla, porque existirá una correspondencia directa entre el requisito del usuario y el componente. Sin embargo, en otras ocasiones puede resultar una tarea compleja, ya que puede ocurrir que para un requisito se tengan varios posibles componentes que lo satisfacen e incluso que no exista una correspondencia directa entre un requisito y un componente y que habrá que utilizar varios componentes para satisfacerlo.

Tras la selección, es necesaria una etapa de validación, en la que se pueda realizar todas las pruebas necesarias para determinar si un componente cumple o no los requisitos. Los programadores se encuentran con el problema de que, a veces, los componentes no están lo suficientemente detallados para construir una batería de pruebas que garantice el buen funcionamiento completo del componente. En ocasiones, dichos componentes poseerán más funcionalidad de la necesaria para satisfacer el requisito y eso puede representar un potencial peligro para el funcionamiento final del sistema.



### Actividades

1. Haga un diagrama con todos los pasos que se llevan a cabo desde la identificación de los componentes hasta su validación y aceptación para el proyecto.

## 2.2. Análisis de los componentes reutilizables

Como se ha comentado en el apartado anterior, cada componente requiere una fase de validación. Esto es porque algunos componentes se desarrollan para el dominio específico del sistema, mientras que otros se extraen de la base de datos de los componentes e incluso de aplicaciones de terceros existentes.

Cuando se seleccionan los componentes no se asegura que puedan integrarse de forma simple y proporcionando la mayor eficiencia al sistema. Son necesarias una serie de etapas que permitan analizar si un componente es adecuado para la integración o no.



### Importante

Un componente queda caracterizado por su interfaz.

**La calificación de componentes** permite asegurar que la funcionalidad del componente encaja adecuadamente en el diseño elaborado para el sistema y que, además, cumple con todas las características de calidad, fiabilidad, uso, etc. de la aplicación.

Una descripción completa de la interfaz proporciona información muy importante para saber si es posible reutilizar un componente en la aplicación. Pero, además, existe otro tipo de información también muy útil para este propósito: Interfaz de Programación de la Aplicación (IPA), herramientas de integración requeridas por el componente, tiempo de ejecución, recursos utilizados, tiempos o velocidades de comunicación, requisitos de servicio del sistema operativo, seguridad, uso de diseño anidado, manejo de excepciones, etc.

Todos estos atributos o factores ayudan a realizar un análisis con detalle de los componentes. Cuando el componente es diseñado específicamente para el cumplimiento del modelo de análisis, obtener estos datos es relativamente sencillo. El problema surge con los componentes reutilizados. Para ellos, es más complicado, porque se necesitan conocer algunos aspectos internos del componente, que en la mayoría de los casos se desconocen.



### Actividades

2. ¿Cómo cree que debe actuar el equipo de desarrollo si en medio de la fase de construcción se comprueba que un determinado componente no encaja en el sistema?

## 2.3. Análisis de la integración de los componentes en la arquitectura del sistema

Cuando un componente ha sido calificado y es válido para formar parte del sistema, existen dos escenarios posibles:

1. El componente se integra sin problemas, es decir, el componente satisface los requisitos de interoperabilidad con otros componentes y su interfaz se integra correctamente en el diseño de la arquitectura del sistema.
2. El componente no se integra fácilmente, existen conflictos en alguna área donde se debe aplicar el componente.

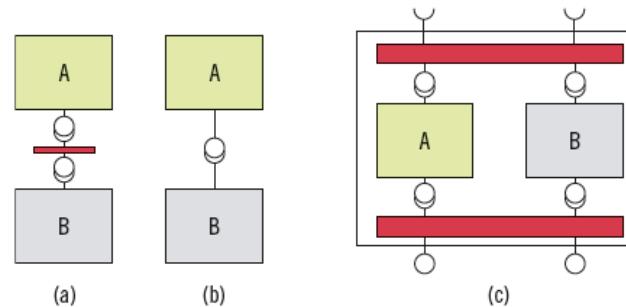
En este último caso, los conflictos se solucionan utilizando una técnica de adaptación denominada **encapsulación de componentes**. Si se puede acceder al contenido interno del componente, se pueden realizar los cambios necesarios para resolver los conflictos, lo que se conoce como **encubrimiento de caja blanca**. Si el contenido del componente no es accesible, la mayoría de las veces es necesario aplicar una traducción de pre y pos procesamiento en la interfaz del componente que permita enmascarar y resolver los conflictos, lo que se conoce como **encubrimiento de caja negra**.

A veces, no es suficiente con estas técnicas para resolver los conflictos. En ese caso, el programador debe evaluar la posibilidad de desechar el componente y diseñar uno nuevo.

Después de aplicar la adaptación a los componentes, es hora de ensamblarlos. La composición de componentes es el proceso que se encarga de crear el sistema a partir de todos los componentes disponibles. Pero la integración no es algo sencillo, existen varios tipos de composiciones:

- **Composición secuencial:** cuando los componentes se deben ejecutar en secuencia. Hace falta escribir código para enlazar las interfaces de los dos componentes (a).
- **Composición jerárquica:** cuando un componente realiza una llamada directa a la interfaz de otro componente (b).
- **Composición aditiva:** se da cuando dos o más componentes se agrupan para formar un nuevo componente donde su interfaz es una combinación de las interfaces de los componentes internos. En este caso, se debe añadir código que controle las operaciones de entrada y salida sobre las interfaces internas (c).

Tipos de integración de componentes

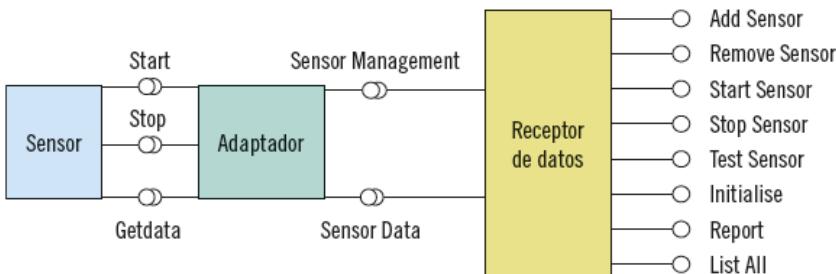


Se ha visto que a veces es necesario incluir código extra para ensamblar los componentes. Esto suele ocurrir con mucha frecuencia cuando los componentes no son diseños específicos para el sistema en concreto. En estos casos, las incompatibilidades entre las interfaces pueden referir alguno de los siguientes tipos:

- **Incompatibilidad de parámetros:** el nombre de las interfaces es el mismo, pero difieren en el número y/o tipo de parámetros.
- **Incompatibilidad de operaciones:** los nombres de las interfaces son distintos.
- **Operaciones incompletas:** la interfaz de uno de los componentes es un subconjunto funcional de la interfaz de otro, por lo que requiere de nuevo componente.

En estos casos, es necesario reescribir código o crear un nuevo componente adaptador que haga compatibles las interfaces que se van a utilizar.

#### Ejemplo de cómo los distintos componentes usan interfaces para la comunicación



#### Actividades

3. ¿Sería posible desarrollar un sistema solo con componentes reutilizables sin añadir código de adaptación de sus interfaces? Si la respuesta es afirmativa, ponga un ejemplo. Y, si es negativa, razoné la respuesta.

#### 2.4. Identificación de los modelos funcionales y de datos de los componentes

El modelo de objetos está compuesto por las propiedades estructurales del sistema; el modelo dinámico y el funcional, por el contrario, describen su comportamiento.

Un **modelo funcional** describe la dependencia de datos en el sistema y la computación dentro de este. Por ejemplo, cómo se obtienen los valores de salida a partir de los valores de entrada.

El modelo funcional es producto de los diagramas de flujo de datos y de la especificación de procesos. Además, sirve para identificar nuevos objetos, atributos y operaciones dentro del sistema, que serán añadidos al modelo de objetos. También sirve como medio para identificar restricciones a operaciones en el mundo real.

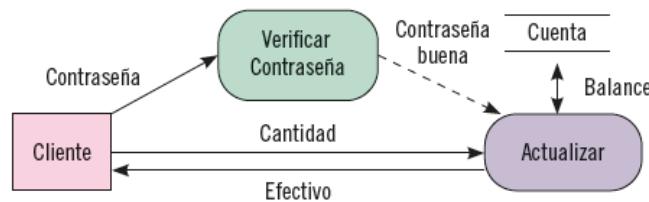
Los diagramas de flujo son útiles para mostrar la funcionalidad de un sistema a diferentes niveles. E incluso se pueden encontrar anidados a diferentes profundidades, dando una descripción completa del modelo funcional. Describen detalladamente las operaciones

definidas en el modelo de objetos y las actividades definidas en el modelo dinámico. Los atributos de los objetos se modelan como flujos de datos en el modelo funcional.

Se componen de los siguientes elementos:

- **Procesos:** representan transformaciones de valores de datos. Los procesos en el modelo funcional se corresponden con las operaciones (y métodos) en el modelo de objetos.
- **Flujo de datos:** representan las entradas y salidas de las transformaciones y computaciones de los procesos, y también valores intermedios dentro de una computación.
- **Almacenamiento de datos:** objetos que guardan datos de forma persistente; no poseen operaciones, pero permiten el acceso para resolver consultas sobre los datos.
- **Actores:** los elementos activos del diagrama. Su objetivo es producir y consumir datos. Para ello, se representan como fuentes y terminales dentro del flujo de datos del sistema.
- **Flujo de control:** representan el control de las operaciones. El momento de su ejecución se describe en el modelo dinámico.

Diagrama de flujo de datos con flujo de control para el retiro de una cuenta



El **modelo de datos** intenta representar las estructuras lógicas y abstractas, junto con los datos de todo el sistema que representan los conceptos del mundo real expuestos en la fase de análisis. Se suelen utilizar diagramas de entidad-relación, partiendo de una situación real donde se definen entidades y relaciones entre dichas entidades.



## Definición

### Entidad

Concepto u objeto del mundo real que interviene en el funcionamiento del sistema y, por lo tanto, hay que modelarlo para que se cumplan los objetivos.

Las entidades pueden contener **atributos**, que son los datos que definen el objeto. Algunos atributos tendrán valores que no puedan ser repetidos; es lo que se conoce como **clave de la entidad**. En algunos casos puede haber más de una y habrá que escogerla usando las siguientes normas:

- Tiene que ser **única**.
- Tiene que existir un **pleno conocimiento** de ella.
- Tiene que ser **mínima**.

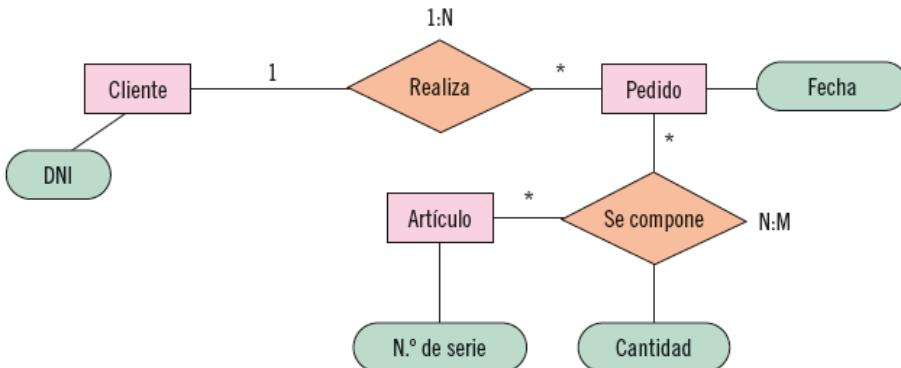
La **relación** es la asociación entre entidades; no tiene existencia propia en el mundo real, pero es necesaria para reflejar las interacciones entre entidades. Puede ser de tres tipos:

- **Relaciones 1-1:** las entidades de la relación intervienen una a una. Por ejemplo: en la relación matrimonio, entre dos personas.

• **Relaciones 1-n:** en este caso una entidad puede estar asociada con muchas otras mediante la relación. Por ejemplo: la entidad empresa y la entidad trabajador. Una misma empresa tiene más de un trabajador.

• **Relaciones n-n:** en este otro caso cada entidad puede estar asociada a otras muchas y viceversa. Por ejemplo: las matrículas entre alumnos y entidades de formación. Un alumno puede tener varias matrículas en cada entidad de formación. Y estas, a su vez, pueden tener varias matrículas abiertas para el alumno. **n-m**

Ejemplo de diagrama de datos



### 3. T閏nicas de programaci髇 presentes en lenguajes de uso com黨 aplicables al desarrollo de software de gesti髇 de sistemas

Antes de nombrar y describir t閏nicas de programaci髇 para el desarrollo de *software*, conviene identificar qu閚 enfoque de desarrollo es el m醩 adecuado al proyecto. En una aproximaci髇, se podr韆 elegir una metodolog韆 basada en funciones, donde los datos se consideran separados de los procesos que los transforman y, por lo tanto, dando m醩 importancia a una descomposici髇 funcional del sistema. Pero tambi閑n podr韆 ajustarse a un enfoque orientado a objetos, que se centra en primer lugar en identificar los objetos o datos del dominio de aplicaci髇 para finalmente establecer los comportamientos sobre estos.

Se ha demostrado con la experiencia que este ltimo enfoque se comporta mejor ante los cambios de requerimientos, porque se basa en la estructura subyacente del dominio de aplicaci髇 en vez de en los requerimientos funcionales de un determinado problema. A continuaci髇, se explicarn los dos paradigmas de programaci髇 que estn detr醤s de estos dos enfoques opuestos entre s.

#### 3.1. Programaci髇 estructurada

La programaci髇 estructurada surge a finales de la d閡ada de los 60 como una forma de programar donde lo m醩 importante era producir y mantener c骻ido que facilitara la compresi髇 por parte del programador.

Se basa en la idea de que todo programa puede escribirse como una combinación de tres tipos de instrucciones de control: secuenciales, condicionales e iterativas.



### Sabía que...

La idea de que todo programa puede ser escrito mediante tres tipos de instrucciones es fruto de un teorema matemático cuyo autor fue Edsger Dijkstra.

Esta forma de programar es la más extendida hasta la llegada de la programación orientada a objetos. Además, aplica la idea de que todo programa debe tener un diseño modular, intentando descomponer lo máximo posible el problema que se intenta solucionar. En la mayoría de los casos, esta descomposición adquiere un enfoque descendente, lo que implica que el problema se divide en módulos más pequeños que resuelven problemas más sencillos y que ayudan a resolver el problema principal.

Por lo tanto, se pueden resumir las características de la programación estructurada de la siguiente forma:

- El programa completo posee un diseño modular.
- Los módulos se diseñan con un enfoque descendente o ascendente.
- Cada módulo se programa usando únicamente instrucciones secuenciales, selectivas y repetitivas.
- Los conceptos de estructuración y modularidad se complementan.



### Ejemplo

Calcular la media de una serie de números positivos. Se supone que se introducen los valores mediante el terminal y se acaba cuando se escribe un 0.

Inicio

1. Inicializar contador de números C y variable suma S a cero ( $S \leftarrow 0, C \leftarrow 1$ ).  $C \leftarrow 0$

2. Leer un número en la variable N (leer(N))

3. Si el número leído es cero: (si ( $N = 0$ ) entonces)

    3.1. Si se ha leído algún número (Si  $C > 0$ )

        | calcular la media; ( $media \leftarrow S/C$ )

        | imprimir la media; (Escribe(media))

    3.2. si no se ha leído ningún número (Si  $C=0$ )

        | escribir no hay datos.

    3.3. fin del proceso.

4. Si el numero leído no es cero : (Si ( $N < 0$ ) entonces)

        | calcular la suma; ( $S \leftarrow S+N$ )

        | incrementar en uno el contador de números; ( $C \leftarrow C+1$ )

        | ir al paso 2.

Fin

La codificación de este algoritmo en pseudocódigo podría ser el siguiente:

```
algoritmo media
inicio
    variables
        entero: n, c, s;
        real: media;

        C ← 0;
        S ← 0;
        repetir
            leer(N)
            Si N <> 0 Entonces
                S ← S+N;
                C ← C+1;
            fin si
            hasta N=0
            si C>0 entonces
                media ← S/C
                escribe(media)
            sino
                escribe('no datos')
            fin si
        fin
```



## Actividades

4. ¿Por qué cree que ha sido tan importante el enfoque estructurado para los comienzos del software?

El lenguaje C es conocido como "lenguaje de programación de sistemas" y fue desarrollado por Dennis Ritchie para *Unix*. Es un tipo de lenguaje estructurado y, aunque es un lenguaje de alto nivel, brinda la posibilidad de programar a bajo nivel, por lo que a veces se considera un lenguaje de bajo nivel. La potencia de este lenguaje es tal que se ha convertido en el más utilizado en el desarrollo de sistemas operativos. A partir de ahora, cuando se hable de técnicas para el desarrollo de sistemas se usará como ejemplo este lenguaje para describir todos los conceptos necesarios para entenderlas.

### Tipos primitivos y estructurados

La programación estructurada se basa, como se ha dicho, en ejecutar secuencialmente instrucciones. Las instrucciones realizan operaciones sobre un conjunto de datos alterando lo que se conoce como entorno del programa. Todos los datos que el programa maneja tienen asociado un tipo, que determina su naturaleza.

Cada dato, a parte de su tipo, posee unas propiedades que lo caracterizan en cada momento de la ejecución del programa:

- Tiene un valor concreto (estado).

- Tiene una representación concreta:

- Algorítmica (notación sintáctica).
- Interna (codificación).

- Se pueden aplicar acciones sobre él.
- Posee ámbito.

El tipo de datos viene determinado por las siguientes propiedades:

- Un dominio de valores.
- Una representación (interna y algorítmica).
- Conjunto de operadores asociado.
- Nombre identificativo.



### Ejemplo

Tipo de dato Entero:

Dominio: [-32768...0...32767]

Representación:

Interna (2 bytes).

Externa: <entero>::=[+|-]<dígito>{<dígito>}

**representación matemática**

Conjunto de operadores: +, \*, -, div, mod, sqr, etc.

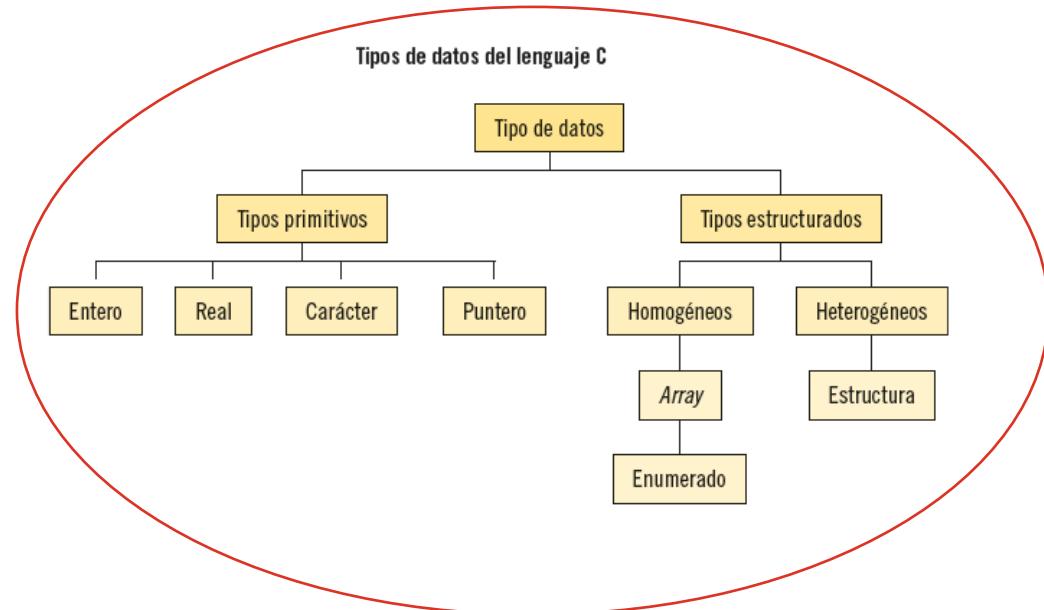
En todos los lenguajes de programación existen tipos de datos fundamentales que se denominan primitivos y que sirven para definir otros tipos de datos complejos o compuestos, denominados estructurados.



### Sabía que...

Cada lenguaje proporciona unos tipos de datos primitivos diferentes.

En C, la clasificación de tipos se puede realizar según el siguiente diagrama:



En C, los tipos primitivos son: enteros (**int**, **short**, **long**), reales (**float**, **double**), caracteres (**char**) y punteros (\*).

Enteros		
<b>int</b>	Máximo = 2,147,483,647 Mínimo = -2,147,483,648 <b>Unsigned (sin signo)</b> Máximo = 4294967295 Mínimo = 0	4 bytes
<b>short</b>	Máximo = 32767 Mínimo = -32768 <b>Unsigned (sin signo)</b> Máximo = 65,535 Mínimo = 0	2 bytes
<b>long</b>	Máximo = 9,223,372,036,854,775,807 Mínimo = -9,223,372,036,854,775,808 <b>Unsigned (sin signo)</b> Máximo = 18,446,744,073,709,551,616 Mínimo = 0	8 bytes
<b>float</b>	Exponente máximo = $10^{37}$ Exponente mínimo = $10^{-37}$	4 bytes

<b>double</b>	Exponente máximo = $10^{308}$ Exponente mínimo = $10^{-308}$	8 bytes
---------------	---	---------

Caracteres		
<b>char</b>	Máximo = 255 Mínimo = 0	1 byte

El tipo **puntero** representa un valor que corresponde con una dirección de memoria, de forma que cualquier variable de este tipo apuntará a dicha dirección de memoria.

Los tipos estructurados están compuestos de tipos primitivos y se puede hacer una distinción clara entre aquellos tipos estructurados que proporciona el lenguaje por defecto y aquellos que puede definir uno mismo agrupando datos de distintos tipos.

En C, se pueden encontrar hasta tres clases de tipos estructurados: *arrays*, estructuras y enumerados.

Un **array** es una colección de datos del mismo tipo almacenados consecutivamente en memoria, y que proporciona un mecanismo fácil de acceso a cada elemento de la colección.

Por ejemplo, un *array* de seis elementos de tipo entero:

0	1	2	3	4	5
1	3	1	7	4	2

Un tipo **enumerado** es una colección de elementos identificados alfanuméricamente en el momento de definir el tipo enumerado y en el que cada elemento corresponde a un número entero.

Por ejemplo: un tipo enumerado de 7 valores podría ser:

```
{lunes, martes, miercoles, jueves, viernes, sabado, domingo}
```

Donde el valor lunes corresponde a la constante 0 y el valor domingo a la constante 6.

Una **estructura** es una colección de diferentes tipos de datos que se agrupan.

Por ejemplo: esta es una estructura formada por un *array* de 4 elementos, 1 entero, 1 real y 1 carácter.

0	1	2	3
1	3	1	7
3			
2.42			
'c'			



## Actividades

- Realice un resumen en forma de diagrama con los tipos básicos y los tipos estructurados vistos hasta ahora.



## Aplicación práctica

Está desarrollando una aplicación informática con la que se quiere gestionar los datos personales de los alumnos de un colegio. Cada alumno tendrá como datos su nombre, apellidos, edad, dirección, población, provincia y código postal del domicilio. ¿Cómo elaboraría un diagrama visual donde queden representadas todas las estructuras de datos necesarias para almacenar los datos de hasta 100 alumnos y teniendo en cuenta que no puede almacenarse una cadena de más de 20 caracteres y usando el lenguaje C?

## SOLUCIÓN

Nombre	0	1	...	19
	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>
Apellidos	0	1	...	19
	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>
Edad	0	1	...	19
	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>
Dirección	0	1	...	19
	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>
Población	0	1	...	19
	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>
Provincia	0	1	...	19
	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>
Código postal				

## Variables. Ámbito de utilización

En un lenguaje de programación es necesario poder almacenar la información para procesarla. Una **variable** es un espacio de almacenamiento identificado por un nombre. De esta forma, se puede guardar y obtener información del espacio almacenado a través de ese nombre o identificador. En la mayoría de los lenguajes, es necesario declarar las variables antes de usarlas. Declararla significa que el lenguaje entiende que es necesario reservar una zona de memoria e identificarla para usarla más tarde. Sin embargo, en otros lenguajes, solamente con usar un identificador como variable es suficiente para que se realice la reserva de memoria.



### Importante

En C, es necesaria la declaración y, además, especificar el tipo de dato que será almacenado en la variable.

Por ejemplo: la siguiente sentencia declara una variable con nombre x que puede almacenar un valor entero de 4 bytes. Por lo tanto, el valor de x podrá ser cualquier número entre -2.147.483.648 y 2.147.483.647.

```
int x;
```

Después de la declaración, es necesario asignarle un valor. Esto es lo que se conoce como **inicialización**. La inicialización se consigue mediante el símbolo igual.

Por ejemplo: se está inicializando una variable denominada **x** a un valor entero de 323.

```
x = 323;
```

En contraposición a lo que se ha comentado, existe la posibilidad de establecer un espacio de almacenamiento para guardar datos que no van a poder ser modificados durante la ejecución del programa. Los identificadores de estos espacios se denominan **constantes**.

En C, se puede definir una constante igual que una variable, pero anteponiendo la palabra reservada **const**.

Por ejemplo: se declara una constante denominada **a** y se inicializa con el valor 43;

```
const int a = 43;
```

Es importante aclarar que, cuando se declara una constante, es necesario al mismo tiempo inicializarla con el valor que va a contener durante toda la ejecución del programa.

El **ámbito de utilización** de una variable es la zona desde la que esa variable es accesible dentro del programa. Cuando se declara una variable, automáticamente se está definiendo su ámbito. Este ámbito depende del lugar donde se declare la variable. Se volverá a este concepto y a desarrollarlo más adelante cuando se vea qué son las funciones.

## Operadores aritméticos y lógicos

Los operadores aritméticos, como su propio nombre indica, son operadores que permiten realizar operaciones aritméticas. La siguiente tabla muestra los operadores aritméticos en el lenguaje C:

Operador	Acción	Ejemplo
-	Resta	x = 5 - 3; // x vale 2
+	Suma	x = 2 + 3; // x vale 5
*	Multiplicación	x = 2 * 3; // x vale 6
/	División	x = 6 / 3; // x vale 2
%	Módulo	x = 5 % 2; // x vale 1
--	Decremento	x = 1; x--; // x vale 0

++	Incremento	$x = 1; x++; // x vale 2$
----	------------	---------------------------

Durante la ejecución de un programa, se producen múltiples situaciones en las que es necesario evaluar que una determinada condición sea cierta o no. Para ello, se utilizan los operadores relacionales.

Operador	Acción	Ejemplo	Significado
<	Menor que	$a < b$	$a$ es menor que $b$
>	Mayor que	$a > b$	$a$ es mayor que $b$
==	Igual a	$a == b$	$a$ es igual a $b$
!=	Distinto a	$a != b$	$a$ es distinto a $b$
<=	Menor que o igual a	$a <= 5$	$a$ es menor que o igual a 5
>=	Mayor que o igual a	$a >= b$	$a$ es mayor que o igual a $b$

Por último, los operadores lógicos permiten evaluar condiciones y/o expresiones lógicas.

Operador	Acción
&&	Conjunción (y)
	Disyunción (o)
!	Negación

Las tablas de verdad de estos operadores son:

Conjunción		
Operador1	Operador2	Resultado
Falso	Falso	Falso
Falso	Cierto	Falso
Cierto	Falso	Falso
Cierto	Cierto	Cierto

Disyunción		
Operador1	Operador2	Resultado
Falso	Falso	Falso
Falso	Cierto	Cierto
Cierto	Falso	Cierto
Cierto	Cierto	Cierto

Negación	
Falso	Cierto
Cierto	Falso



### Actividades

6. Determine la veracidad o falsedad de las siguientes sentencias sabiendo que  $a = 2$  y  $b = 3$ .

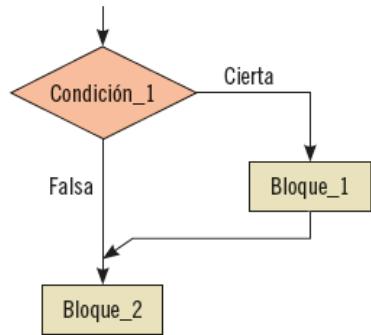
Expresión	Resultado
$(a == 3) \parallel (b == 3)$	v
$(a > 3) \&\& (b == 3)$	f
$(a > 3) \&\& (b != 3)$	f
$(a > 3) \parallel ((a == 2) \&\& (b > 1))$	v

### Estructuras de control. Bucles, condicionales y selectores

Las estructuras de control permiten que los programas puedan tomar decisiones y realizar repeticiones de procesos sin tener que multiplicar el código. Son la clave de la programación estructurada.

Un programa es un conjunto de instrucciones que se ejecutan de forma secuencial. El orden secuencial no altera el flujo de control. Sin embargo, es necesario y normal que un programa tome decisiones, teniendo en cuenta las condiciones que pueden influir en la secuencia de pasos que el programa debe dar.

Las estructuras de control condicionales permiten alterar el flujo del programa, en función de ciertas condiciones:



En C, la sintaxis de estas sentencias es la siguiente:

```

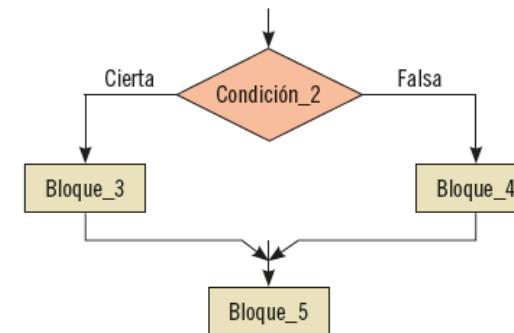
if (condición)
    sentencia;

if (condición) {
    bloque
}

```

Donde bloque es un conjunto de sentencias.

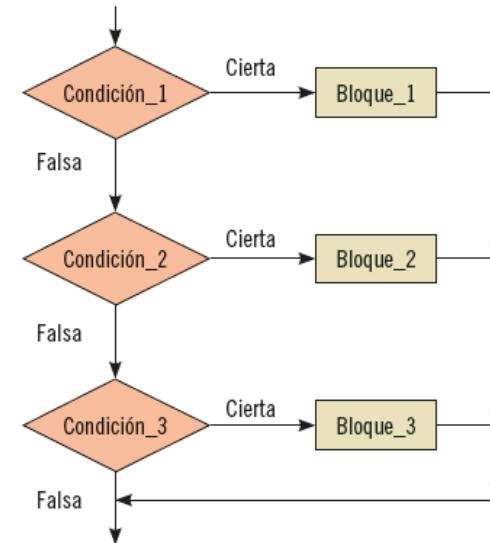
Otra forma de utilizar esta estructura de control es incluyendo un camino alternativo para el flujo del programa:



Su sintaxis en lenguaje C sería:

```
if (condición)
    sentencia1;
else
    sentencia2;

if (condición) {
    bloque_1;
} elseif {
    bloque_2;
}
```



Donde **bloque\_1** y **bloque\_2** son conjuntos de sentencias.

La tercera y última forma de utilizar la estructura condicional **if** es con múltiples caminos, de la siguiente forma:

La sintaxis en C sería:



```

switch (expresión) {
    case expr_cte1:
        bloque_1;
        break;
    case expr_cte2:
        bloque_2;
        break;
    case expr_cte3:
        bloque_3;
        break;
    default:
        bloque_4;
}

```

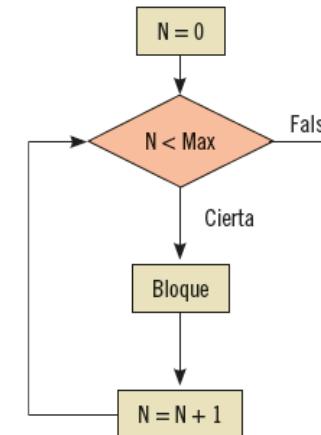
Donde bloque\_1, bloque\_2, bloque\_3 y bloque\_4 pueden ser un conjunto de una o más sentencias.

A partir de la evaluación de la expresión, que se realiza una sola vez, se compara con las constantes de cada caso empezando por la primera hasta que alguna de las igualdades sea cierta, en cuyo caso se ejecuta el bloque de sentencias que le sigue. La etiqueta **default**

indica el conjunto de sentencias que se debe ejecutar si no se produce ninguna coincidencia en la comparación de expresiones. Sin embargo, esta etiqueta no es obligatoria.

El otro conjunto de sentencias importantes en la programación estructurada son las estructuras de control iterativas: los bucles **for** y **while**.

Estas permiten resolver de forma elegante problemas que aparecen de forma repetitiva y que de otra manera implicarían el uso de una gran cantidad de sentencias. El bucle for permite repetir un conjunto de sentencias un número determinado de veces, mientras que el bucle while permite la repetición mientras se cumpla una determinada condición.





## Actividades

7. Se quiere usar la función printf() para mostrar por pantalla la nota textual de un alumno. Es decir, si el alumno ha sacado un 7.5 se quiere que el programa imprima por pantalla la palabra "notable". Codifique, usando la sentencia apropiada, si la variable nota contiene el valor numérico de un alumno.

El bucle for se compone de tres partes: inicialización, condición y actualización. La inicialización se ejecuta en la primera iteración y sirve para ajustar el valor inicial de la variable que llevará la cuenta de las veces que se ejecuta el bucle. La condición se evalúa siempre al inicio de una iteración y determinará si se ejecuta dicha iteración o termina el bucle. Por último, la actualización sirve para modificar la variable que se encarga de llevar la cuenta.

La sintaxis en C sería:

```
for (inicialización; condición; actualización) {  
    bloque;  
}
```

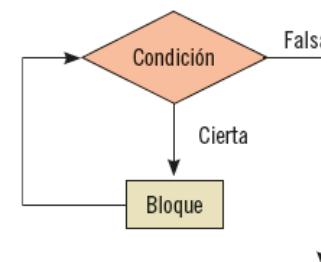
Donde bloque es un conjunto de una o más sentencias.



## Ejemplo

```
for (N = 0; N < Max; N = N + 1) {  
    bloque;  
}
```

El bucle while es una estructura aún más simple que el bucle for. La idea es que un bloque de instrucciones se repetirá mientras una condición sea cierta.





## Sabía que...

Es posible convertir cualquier bucle iterativo de tipo `for` en un bucle iterativo de tipo `while`.

La sintaxis en C sería:

```
while (condición) {  
    bloque;  
}
```



## Ejemplo

```
numero = int(input("Escriba un número positivo: "))  
while numero < 0:  
    print("¡Ha escrito un número negativo! Inténtelo de nuevo")  
    numero = int(input("Escriba un número positivo: "))  
  
print("Gracias por su colaboración")
```



## Actividades

8. Escriba un programa, usando la estructura de control adecuada, que escriba por pantalla los números del 1 al 100.



## Aplicación práctica

**A partir del siguiente trozo de código, le piden elaborar su diagrama de flujo.**

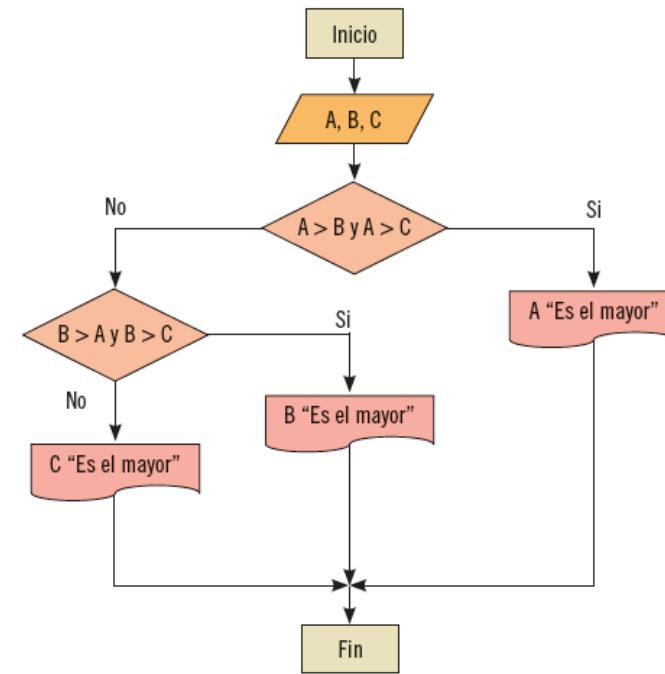
```

A = 2;
B = 3;
C = 3;

Si A > B y A > C Entonces
    Escribir A "Es el mayor"
Sino
    Si B > A y B > C Entonces
        Escribir B "Es el mayor"
    Sino
        Escribir C "Es el mayor"
    Fin_Si
Fin_Si

```

## SOLUCIÓN



## Funciones y procedimientos. Parámetros por valor y referencia

Las funciones y procedimientos son el concepto más importante de la programación modular y la clave de la metodología “divide y vencerás” en programación.

Tanto las funciones como los procedimientos son denominados **módulos**, es decir, un trozo de un programa que realiza unas tareas determinadas, y que el programa puede llamar o invocar todas las veces que quiera. En general, el módulo recibe unas entradas en forma de variables, denominadas **parámetros**, y produce o no una salida en forma de valores. El trozo de programa que realiza la llamada al módulo debe proporcionar esa entrada y obtener la salida si la hubiera.

El uso de módulos permite crear programas complejos, descomponiendo un problema en otros problemas más pequeños. El procedimiento sería el módulo que no devuelve ningún resultado a partir de la entrada, mientras que la función sería el módulo que genera una salida.

En ambos casos, se componen de un conjunto de sentencias a las que se le asocia un identificador (nombre), un conjunto de entradas (parámetros del procedimiento o función), un entorno (variables que usará) y una salida (valor o conjunto de valores) si se trata de una función.

En C, la sintaxis para declarar una función y un procedimiento es la siguiente:

```
void nombre_de_función(parámetros) {  
    sentencias;  
}
```

La palabra **void** indica que se trata de un procedimiento y que, por tanto, no se devolverá ningún valor.



### Ejemplo

```
void mostrar_area(float b, float h) {  
    printf("El área es: %f", b*h/2)  
}
```

En este caso, se tiene un procedimiento que muestra por pantalla el área total de un triángulo, como parámetros de entrada del procedimiento se tienen la base *b* y la altura *h*. El procedimiento se compone de una sentencia **printf** que muestra el total a partir de la base y la altura.

Un ejemplo de invocación de este procedimiento sería:

```
mostrar_area(5, 3);
```

El resultado en pantalla de la ejecución del procedimiento sería:

```
El área es: 7.5
```

La sintaxis para declarar una función es ligeramente diferente:

```
tipo nombre_de_función(parámetros) {  
    sentencias;  
    return valor_de_tipo;  
}
```

Ahora, la palabra **void** ha sido sustituida por el tipo de datos que va a devolver la función. Además, la devolución del valor se hace de forma explícita utilizando la palabra clave **return**, generalmente al final de la función.



### Ejemplo

```
float area(float b, float h) {  
    return b*h/2;  
}
```

En este caso, la función devuelve el valor del área calculada en función de los parámetros de entrada: base *b* y altura *h*.

Una vez declarada la función, se podrá utilizar invocándola desde el programa principal. Por ejemplo, en la siguiente invocación:

```
resultado = area(3,5);
```

Como la función devuelve un valor, es necesario que en la invocación de la función se asigne ese valor a un contenedor o variable que pueda almacenarlo para utilizarlo posteriormente. El resultado de la sentencia anterior será que la variable resultado tiene un valor de 7.5.

## Recursividad

La recursividad es una técnica de programación que permite dar una solución elegante y simple a problemas que se pueden descomponer en el mismo tipo de problemas, pero de menor tamaño.

La idea principal de la recursividad es que una función se llame a sí misma para resolver el mismo problema, pero, como se ha dicho, con un tamaño menor. De esta manera, el proceso tiene una naturaleza iterativa, pero es resuelta mediante una secuencia de invocaciones a funciones.



### Ejemplo

Calcular el factorial del número 8.

La declaración de la función recursiva sería la siguiente:

```
int factorial(int n) {  
    int res = 1;  
    if (n<2) {  
        res = 1;  
    } else {  
        res = n * factorial(n-1);  
    }  
    return res;  
}
```

A partir de esta definición, se puede invocar la función para obtener una solución recursiva del problema.

```
resultado = factorial(8);
```

La invocación anterior produce una secuencia de llamadas que van componiendo la solución, hasta que la última llamada devuelve el resultado final:

```
res = 1 * 8 * factorial(8-1)
res = 1 * 8 * 7 * factorial(7-1)
res = 1 * 8 * 7 * 6 * factorial(6-1)
res = 1 * 8 * 7 * 6 * 5 * factorial(5-1)
res = 1 * 8 * 7 * 6 * 5 * 4 * factorial(4-1)
res = 1 * 8 * 7 * 6 * 5 * 4 * 3 * factorial(3-1)
res = 1 * 8 * 7 * 6 * 5 * 4 * 3 * 2 * factorial(2-1)
res = 1 * 8 * 7 * 6 * 5 * 4 * 3 * 2 * 1
res = 40.320
```

Un analista programador le pide generar la secuencia de llamadas de la invocación suma(6) para el siguiente trozo de programa:

```
int suma(int n) {
    int res = 1;
    if (n=1) {
        res = 1;
    } else {
        res = n + suma(n-1);
    }
    return res;
}
```



## Actividades

9. ¿Cuál es la diferencia fundamental entre esta técnica y el uso de un bucle para realizar la misma función? Razone la respuesta.



## Aplicación práctica

¿Cuál sería esa secuencia de llamadas?

SOLUCIÓN

```
res = 6 + suma(6-1)
res = 6 + 5 + suma(5-1)
res = 6 + 5 + 4 + suma(4-1)
res = 6 + 5 + 4 + 3 + suma(3-1)
res = 6 + 5 + 4 + 3 + 2 + suma(2-1)
res = 6 + 5 + 4 + 3 + 2 + 1
res = 20
```

---

## Programación de elementos básicos: cadenas, fechas y ficheros

Muchos problemas en programación requieren el uso de tipos de datos más complejos que los definidos de forma nativa. Uno de estos tipos más complejos son las cadenas de caracteres. Estas permiten mostrar mensajes por pantalla que sean entendidos por el usuario que ejecuta el programa.

En C, no existe un tipo definido para las cadenas de caracteres. Alternativamente, el lenguaje permite usar un tipo estructurado para definir este nuevo tipo. En concreto, se puede ver una cadena de caracteres como un arreglo (*array*) de caracteres. La definición de este tipo complejo sería:

```
char color[] = "azul"
```

Esto indicará al compilador que la variable color almacenará la cadena de caracteres "azul", reservando espacio para un *array* de 5 caracteres: 'a', 'z', 'u', 'l' y '\0'.

Ejemplo de uso de cadenas de caracteres:

```
int main() {  
    char color[] = "azul";  
    printf("Mi color favorito es el %s", color);  
}
```

Este programa muestra por pantalla la frase: "Mi color favorito es el azul".

En C, cuando se representan cadenas de caracteres de esta forma, mediante *arrays*, se debe tener en cuenta que el *array* almacena los caracteres de la cadena y su terminación, el carácter '\0' para identificar el final de la cadena dentro del propio *array*.

Se ha visto cómo definir cadenas de caracteres en tiempo de compilación, es decir, antes de la ejecución del programa. Sin embargo, a veces se necesita conocer la cadena de caracteres en tiempo de ejecución. Por ejemplo, si se quiere que el usuario introduzca su nombre:

```
int main() {
    char nombre[20];
    printf("Introduce tu nombre: ");
    scanf("%s", nombre);
}
```

Aquí, el programa se queda a la espera de que se introduzca el nombre, que será almacenado en la variable **nombre**. El único problema es que se ha definido un *array* de 20 elementos como máximo, por lo que solo se permitirá guardar nombres que no superen los 20 caracteres. Existen formas avanzadas de resolver este problema, pero no es objetivo de este manual.

Otro de los tipos de datos que puede ser necesario utilizar con mucha frecuencia es el de fecha. El lenguaje C proporciona funciones y estructuras para el manejo de estas. En concreto, la estructura **struct\_tm** permite guardar una fecha concreta separando todos los componentes que la componen y la estructura **time\_t** permite guardar esa fecha en un formato entendible por el sistema.

```
#include <time.h>
#include <stdio.h>

int main(void)
{
    struct tm str_time;
    time_t time_of_day;

    str_time.tm_year = 2050-1990;
    str_time.tm_mon = 10;
    str_time.tm_mday = 23;
    str_time.tm_hour = 14;
    str_time.tm_min = 35;
    str_time.tm_sec = 15;
    str_time.tm_isdst = 0;

    time_of_day = mktime(&str_time);
    printf(ctime(&time_of_day));

    return 0;
}
```

La ejecución de este programa mostrará por pantalla la siguiente frase:

```
Thu Jul 05 11:03:05 2012
```

Los ficheros permiten almacenar información en memoria secundaria o permanente. Se identifican por un nombre compuesto por una ruta o *path* que indica dónde se encuentra el fichero, un nombre propio y, algunas veces, por una extensión que clasifica el tipo de fichero.

Por ejemplo: "C:\Mis documentos\apuntes.doc". El nombre de este fichero se puede descomponer en su *path*: C:\Mis documentos\ , su nombre: apuntes, y su extensión .doc.

Existen dos tipos de ficheros: los ficheros de texto y los binarios. Los **ficheros de texto** almacenan la información como una secuencia de caracteres, es decir, cada byte almacenado corresponde a un carácter imprimible. Los **ficheros binarios**, por el contrario, almacenan la información en bytes que no representan caracteres de texto, sino estructuras más complejas, por lo que imprimir por pantalla un fichero de este tipo no aclararía nada al usuario.

Para que se pueda manejar un fichero desde el código del programa, el lenguaje que se está utilizando debe proporcionar una forma de acceder a esa información almacenada además de operaciones que permitan la recuperación y modificación de esa información. El lenguaje C, el tipo de datos que permite asociar una variable a un fichero y manejarla como si se tratara directamente con el fichero es **FILE**. En concreto, se puede declarar una variable de la siguiente forma:

```
FILE * nom_var_fich;
```

En C todos los ficheros almacenan bytes y es en el momento de la apertura del fichero cuando se decide cómo y qué se almacena en el mismo. Es decir, en la operación de apertura se decide si el fichero va a ser de texto o binario.

### Apertura y cierre de ficheros

Antes de usar un fichero en el programa es preciso realizar la apertura del mismo; posteriormente, si se desean almacenar datos se realizará una operación de escritura y si se quiere leer una operación de lectura. Cuando ya no se quiera utilizar el fichero, es necesario realizar una operación de cierre para liberar la memoria principal que el fichero ocupó durante su uso (mientras esté abierto, el fichero ocupa memoria principal).

En C, la instrucción para abrir un fichero es **fopen** y se usa de la siguiente forma:

```
FILE * fichero;  
fichero = fopen ( nombre-fichero, modo);
```

La instrucción fopen devuelve un puntero a una estructura que representa un fichero dentro del programa. Si se produce algún tipo de error, devuelve el valor **NULL**.

El parámetro nombre-fichero es el nombre que identifica el fichero dentro del sistema operativo.

El modo se especificará como una cadena de caracteres y que definirá el tipo del fichero (texto o binario) y el tipo de apertura que se quiere realizar sobre él, ya sea de lectura, escritura, añadir datos al final, etc. Los modos disponibles son:

- r: sirve para abrir el fichero para lectura. Si el fichero no existe devuelve error.
- W: sirve para abrir el fichero para escritura. Si el fichero no existe, se crea; si el fichero existe, se destruye y se crea uno nuevo.
- a: sirve para abrir el fichero para añadir datos al final del mismo. Si no existe, se crea.
- +: sirve para abrir el fichero para lectura y escritura.
- b: sirve para abrir el fichero como tipo binario.
- t: sirve para abrir el fichero como tipo texto. Si no se pone ni b ni t, el fichero es de texto.

Los modos se pueden combinar para abrir el fichero en el modo adecuado.

Por ejemplo, para abrir un fichero binario existente en modo lectura se usa el modo "rb+"; para abrir un fichero en modo escritura se usa "wb+"; en este caso, si el fichero no existe, se creará.

La forma más normal de usar la función fopen es la siguiente:

```
FILE *fich;  
if ((fich = fopen("nomfich.dat", "r")) == NULL)  
{  
/* control del error de apertura */  
printf (" Error en la apertura. Es posible que el fichero  
no exista \n ");  
}
```

fopen devuelve un descriptor de fichero si todo ha ido bien, y se almacenará en la variable **fich**, después se compara fich con NULL para saber si se ha producido algún error.

Cuando ya no sea necesario el uso del fichero en el programa, se debe cerrar, para lo que se usa la función **fclose (fich)**.

Para usar todas las funciones necesarias para la gestión de ficheros en C, es necesario incluir la librería **<stdio.h>**.



### Actividades

10. ¿Es posible abrir un fichero binario como fichero de texto? ¿Qué resultado daría al leerlo?

## Lectura y escritura en ficheros

Para almacenar y recuperar datos en un fichero es necesario realizar operaciones de escritura y lectura. En C, existen muchas operaciones que permiten realizar varias operaciones entre las que se encuentran: fread - fwrite, fgetc - fputc, fgets - fputs, fscanf - fprintf.

Por lo general, estas operaciones vienen en parejas de funciones. Entre las más usadas, destacan **fread** y **fwrite**.

### Lectura y escritura de bloques (fread-fwrite)

Estas funciones permiten leer y escribir en ficheros que no sean de texto.

El formato de escritura sería:

```
fwrite (direcc_dato, tamaño_dato, numero_datos, punt_fichero);
```

En la función fwrite hay que especificar el número de datos (**numero\_datos**) que se van a escribir en el fichero, a partir de la dirección especificada en **direcc\_dato**. Además, hay que especificar el tamaño de lo que se va a considerar como un dato para la escritura, especificado en **tamaño\_dato**. Finalmente, hay que especificar el puntero al descriptor del fichero **punt\_fichero**. La función devolverá el número de datos escritos en el fichero.

El tamaño en bytes de un dato o un tipo de dato que se va a escribir en el fichero se puede determinar con la función **sizeof (dato)** o **sizeof (tipo-de-dato)**:

Por ejemplo:

```
int i, v[3]; → sizeof (i) daría lo mismo que sizeof(int)  
→ sizeof (v) daría 3 veces el resultado de sizeof (V[1])
```

Un ejemplo de programa que escribe en un fichero sería:

```

<FILE *f;
int v[6], elem_escritos, num;
f = fopen ("datos.cht ", "wb ");
/* Para escribir los 3 últimos elementos de v (el 2, el 3
y el 4) */
elem-escritos = fwrite
(&v[2], sizeof(int), 3, f );
/* Para escribir el primer elemento de v, valen las 2
instrucciones
siguientes */
fwrite (v, sizeof (int), 1, f );
fwrite (&v[0], sizeof(int), 1, f );
/* Para escribir un entero valen las dos siguientes */
fwrite (&
num, sizeof(int), 1, f );
fwrite (&num, sizeof(num), 1, f );

```

La lectura se realiza con la función fread es la siguiente:

```
<fread (direcc_dato, tamaño_dato, numero_datos,punt_fichero);
```

La función fread leerá tantos datos como indique número de datos (**numero\_datos**) en el fichero, los datos leídos serán almacenados en la variable de dirección especificada en **direcc\_dato**. Cada dato que se lee será del tamaño especificado en **tamaño\_dato**.

Por ejemplo:

```

f = fopen ("datos.dat ", "rb ");
elem-escritos = fread (&v[2], sizeof(int), 3, f );
fread (v, sizeof(int), 1, f );
fread (&V[0], sizeof(int), 1, f );
fread (&num, sizeof(int), 1, f );
fread (&num, sizeof(num), 1, f );

```

### **Recorrido de un fichero secuencial (feof)**

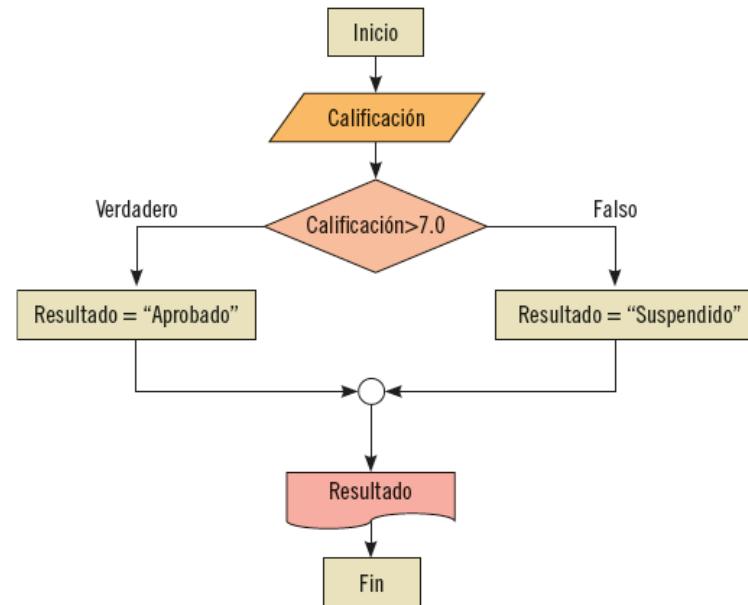
Las operaciones de lectura y escritura sobre el fichero se realizan a partir de la posición en la que se encuentra el cursor del fichero. Este cursor se modifica cada vez que se lee o

escribe. Cuando el cursor alcanza el final del fichero, por ejemplo en una operación de lectura, se devuelve un valor constante que identifica este final conocido como **eof (end of file)**. Se puede consultar si el cursor se encuentra en esta posición usando la función **feof(fichero)**.



### Aplicación práctica

A partir del siguiente diagrama de flujo, le piden como diseñador implementar el algoritmo.



SOLUCIÓN

```
Pedir(Calificación)
Si Calificación > 5.0 Entonces
    Resultado = "Aprobado"
Sino
    Resultado = "Suspensido"
Fin_Si
Mostrar(Resultado)
```

```
int cod;
char c;

c = 'A';
cod = (int) c;
```

En este caso, el carácter 'A' ha sido convertido a un valor numérico, en concreto al valor numérico entero que representa internamente el carácter 'A' en memoria.

## Conversiones de tipos

En los programas a veces es necesario convertir algún valor de un tipo a otro. Por ejemplo, si se está usando un valor numérico decimal y se quisiera solo su parte entera, se podría hacer una conversión, o bien si se tiene un carácter que representa un número y se quisiera obtener ese valor numérico.

C da la posibilidad de convertir el tipo de una variable o constante a otro tipo cualquiera. Para ello, solo es necesario escribir delante de la variable o constante a convertir entre paréntesis el nuevo tipo de valor.

Por ejemplo:

## Manejo de errores (excepciones)

En general, todo programa realiza una serie de operaciones que terminan dando algún resultado. Si todo ha ido bien, el resultado será el esperado, pero puede ocurrir que nunca se alcance ese resultado. Esto es debido a que el programa se comporta de forma errónea bajo ciertas circunstancias. Un ejemplo puede ser un programa que pida introducir un dato positivo y el usuario escriba un valor negativo. Dependiendo de cómo esté desarrollado el programa, esto puede suponer desde que el programa deje de funcionar hasta que los cálculos obtenidos al final no sean los esperados. Es responsabilidad del programador asumir que estas circunstancias se pueden dar y actuar en consecuencia para evitarlas o bien tratarlas de una forma correcta.

Una estrategia para el manejo de este tipo de errores es programar cada clase o función para que detecte sus propios errores y, en función de ese error, actúe en consecuencia y devuelva como resultado una indicación del tipo de error que se habría producido.

Ese valor devuelto se puede utilizar posteriormente para clasificar el tipo de error y, mediante alguna estructura de control de flujo, tratarlo en consecuencia. Esta sería la filosofía mantenida por el lenguaje C. Sin embargo, hoy en día existe un mecanismo de control que mejora el mantenimiento y la gestión de los errores. Es lo que se conoce como tratamiento de excepciones.



### Sabía que...

El lenguaje C no posee la capacidad de manejar excepciones. Sin embargo, C++ sí que permite la gestión de errores a través del manejo de excepciones.

La definición de excepción podría ser: un mensaje de error en tiempo de ejecución que avisa al programador de que se puede producir un mal funcionamiento del programa si no se trata. Como ejemplo, se puede poner una operación de división por cero, el direccionamiento incorrecto, la falta de memoria, etc. Un programa robusto debe controlar estas excepciones y actuar en consecuencia para evitar errores en los resultados. Para ello, C++ dispone de un manejador de excepciones.

La forma de operar una excepción es, en primer lugar, detectar el error y, una vez que se produce, lanzar la excepción. Tras ser lanzada la excepción, el programa debe recoger en alguna parte dicha excepción y darle el tratamiento adecuado. La idea se puede resumir en dos acciones: lanzamiento de la excepción (detección del error) y tratamiento (qué debe hacer el programa).

En C++, se dispone de las siguientes estructuras sintácticas para el manejo de las excepciones: **try**, **catch** y **throw**.

Se usa el siguiente bloque funcional para el tratamiento de errores mediante excepciones:

```
int main() {  
    try {  
        //código del programa donde se puede producir el error.  
    } catch (integer_constraint_error error) {  
        //manejo si se produce el tipo de error integer_constraint_error  
    } catch (integer_format_error error) {  
        //manejo si se produce el tipo de error integer_format_error  
    }  
}
```

De esta forma, cuando se produce alguno de los tipos de error tratados en el bloque anterior, el flujo de control cambia al bloque de sentencias encerrado entre los corchetes que identifican el error en cuestión. Lo que lleva a la necesidad de definir bloques catch con manipuladores de excepción, uno por cada tipo de excepción que se desea manejar.

Si, por el contrario, cuando se produce un error, este lanza una excepción que no está recogida en algún bloque catch, el programa muestra un mensaje de error y se detiene.



## Aplicación práctica

Se está desarrollando un sistema y le encargan hacer más robusta cierta parte del código que es crítica. En esa parte del código pueden aparecer hasta tres tipos de excepciones y se deben controlar correctamente en cada caso. Los tipos de excepciones son `IntegerException`, `FloatException` y `DoubleException`. Codifique la estructura necesaria para el control de excepciones en la parte de código crítica.

### SOLUCIÓN

```
try {  
    //código crítico  
} catch (IntegerException) {  
  
} catch (FloatException) {  
  
} catch (DoubleException) {
```

## Lenguajes estructurados de uso común

Como se ha mencionado anteriormente, la programación estructurada es un paradigma de programación basado en el uso de tres estructuras: secuencia, selección e iteración, y que conduce a mejorar la claridad, la calidad y el tiempo de desarrollo de un programa.

Este paradigma ha sido utilizado durante décadas. Algunos de los lenguajes que más se utilizan son lenguajes basados en este paradigma, como C, Visual Basic, etc.



### Sabía que...

Aunque Java se considera un lenguaje orientado a objetos, también se pueden realizar desarrollos basados en programación estructurada.

**Java** deriva en gran medida de C y C++, pero es un lenguaje de más alto nivel que estos dos. Las aplicaciones de Java son generalmente compiladas a bytecodes que pueden ejecutarse en cualquier máquina virtual Java (JVM) sin importar la arquitectura de la computadora subyacente.

**Visual Basic** es el lenguaje predilecto de Microsoft para desarrollar aplicaciones orientadas a sus sistemas operativos. Debido a la naturaleza de las aplicaciones Windows, el desarrollo está basado y dirigido por eventos. Su principal objetivo es simplificar y reducir la programación utilizando un entorno de desarrollo que proporciona los mecanismos necesarios para el desarrollo de aplicaciones interactivas basadas en ventanas.



### Sabía que...

Visual Basic es un lenguaje que forma un dialecto de Basic.

También permite el uso de acceso a datos utilizando Data Access Objects, Remote Data Objects o ActiveX Data Objects.

**C** es el lenguaje de programación por excelencia para el desarrollo de software de sistemas, concretamente Unix. Fue creado en 1972 por Dennis M. Ritchie en los Laboratorios Bell como evolución del anterior lenguaje B. La característica principal de C es la eficiencia del código que produce.

Entre sus cualidades destaca como lenguaje de tipos de datos estáticos, débilmente tipificado, de medio nivel, aunque muy cercano a las características de un lenguaje de bajo nivel. Los compiladores permiten la posibilidad de mezclar código en el ensamblador o acceder directamente a memoria y/o dispositivos periféricos.

**C++** es un lenguaje de programación de propósito general que fue diseñado a mediados de los años 80 para dotar el poderoso lenguaje C de los mecanismos que permiten la creación y manipulación de clases, es decir, permitir el desarrollo orientado a objetos en C.

Más tarde, se añadieron mecanismos para la programación genérica. Por todo esto se suele decir que C++ es un lenguaje de programación multiparadigma.

### 3.2. Programación orientada a objetos

La programación orientada a objetos es otra forma de programar. Es un nuevo modelo de programación, con una fuerte dependencia teórica y una metodología que potencia

nuevos procesos de desarrollo. Un lenguaje orientado a objetos es un lenguaje de programación que hace uso de este paradigma para desarrollar programas.

Para usar este paradigma de programación, lo normal es que el programador aprenda primero la filosofía (o adquiera la forma de pensar) que introduce. Más tarde, será el momento de estudiar el lenguaje para aplicar esos conocimientos en el desarrollo. A partir de ahora, se va a explicar brevemente la filosofía que encierra esta manera de programar que se desvía completamente de la forma tradicional, intentando dar una visión lo más general posible sin hacer mención de mecanismos propios de ningún lenguaje específico.

La programación orientada a objetos es una evolución natural de los paradigmas de programación. Esta evolución se produce principalmente para reducir la complejidad que, de forma innata, posee el software. La programación estructurada basaba el desarrollo en descomponer el problema en subproblemas más pequeños hasta llegar a acciones (verbos) simples y fáciles de codificar.

La programación orientada a objetos apuesta por descomponer problemas, pero el método de descomposición se basa en la identificación y definición de clases de objetos. El elemento básico no será ya la función, sino unos entes que se definirán más adelante a los que se denomina clases y objetos.



#### Importante

En la programación estructurada, son los datos los que envuelven a las funciones, mientras que, en la programación orientada a objetos, son los métodos o acciones los que operan sobre los datos o atributos.

## Clases y objetos

Un objeto en un programa es la representación de un concepto y/o de una idea abstracta en el mundo real. Esa representación contendrá toda la información necesaria para abstraer esa idea o concepto: atributos y operaciones que los manejan.

Cualquier lenguaje de programación proporciona ciertos tipos de datos predefinidos (enteros, reales, booleanos, caracteres) que pueden ser particularizados a valores concretos. La noción de objeto es una reinterpretación del concepto de dato y cualquier valor de los predefinidos de ese lenguaje puede ser implementado como un tipo (simple) de objeto. Un objeto es una instancia de una clase. La clase representa el tipo y el objeto el dato.

Además, una clase especifica un conjunto de variables, que representan el estado del objeto, y un conjunto de operaciones, que determinan el comportamiento de ese objeto. Cada objeto posee sus propias variables de estado, pero comparte las operaciones con el resto de objetos de la clase.



### Nota

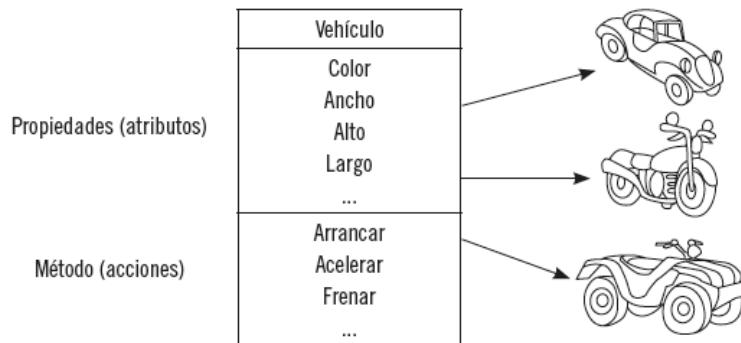
El objeto es a la clase lo que el valor es al tipo.

De este modo, la noción de clase determina un tipo de objeto y, por lo tanto, se puede ver como un tipo de dato.

Desde un punto de vista de alto nivel, la clase es la abstracción de un grupo de objetos que poseen un comportamiento común.

En el siguiente ejemplo, vehículo será la clase.

### Distinción entre el concepto de clase y objeto



Cada lenguaje proporciona una sintaxis distinta para la creación de objetos a partir de una clase definida. En C++, una clase que represente a los puntos de un plano puede ser definida como sigue:

```

class Punto {
    // Coordenadas del punto
    private double x , y ;
    // Constructor para inicializar a 0 ambas coordenadas
    public Punto ( ) {
        x = y = 0;
    }
    // Constructor para inicializar ambas coordenadas
    public Punto ( double a , double b ) {
        x = a ;
        y = b ;
    }
    // Incrementa las coordenadas del punto
    public void trasladar ( double a , double b ) {
        x = x + a ;
        y = y + b ;
    }
}

```

Ejemplo de clase representada mediante el elemento de clase de los diagramas estáticos

Punto
- x: double
- y: double
+ Punto ()
+ Punto (double, double)
+ Trasladar (double, double)
+ Distancia (Punto): double
+ Abscisa (): double
+ Ordenada(): double



### Actividades

11. Elabore un resumen con las diferencias fundamentales entre la programación estructurada y la programación orientada a objetos.

### Atributos

Los atributos de una clase permiten que los objetos puedan ser identificados en función de los valores de esas variables en un momento determinado de la ejecución del programa;

es lo que se conoce como estado del objeto. Cada atributo está compuesto de un nombre y un valor, y pertenece a un tipo de dato que puede ser simple, estructurado e incluso otra clase.

Permiten definir la información que permanecerá oculta dentro de un objeto aparte de la que se pone a disposición del programador. Esa información será manipulada por los métodos definidos sobre dicho objeto.

En C++, el acceso a los atributos desde otras clases o módulos se modifica mediante modos que se especifican a la hora de definir cada atributo. Los modos de acceso son:

- **Público:** los atributos que se definan como públicos pueden ser accesibles desde fuera de la clase y pueden ser consultados o modificados por las otras clases que no estén relacionadas con la clase principal. Suele ser representada con el símbolo `+`.
- **Privado:** los atributos privados solo pueden ser accesibles desde dentro de la propia clase en la que están definidos. Y, por lo tanto, solo pueden ser manipulados y visualizados por las operaciones propias de la clase. Se representan con el símbolo `-`.
- **Protegido:** estos atributos son accesibles desde la propia clase y sus clases hijas (aquellas que heredan de la propia clase). Se representan con el símbolo `#`.

## Métodos

Para modificar el estado de los objetos y permitir la interacción entre ellos se definen operaciones conocidas como métodos; es lo que forma el comportamiento de un objeto. Estos métodos posibilitan, además, la creación y/o consulta del estado del objeto.

Los métodos se definen a la hora de crear la clase, utilizando la sintaxis propia de cada lenguaje. En C++, sería algo así:

<Modo de Acceso> Función

<Nombre> [(Lista Parámetros)]: <Descripción del Tipo de datos>

Por ejemplo: la definición de una clase rectángulo, con dos atributos, largo y ancho, permite la posibilidad de implementar dos métodos para los objetos: uno para calcular el área y otro para calcular su perímetro.

Para cada método se define un nombre y unos parámetros que serán pasados al método. Si el método es susceptible de devolver algún valor, también habrá que especificar el tipo del dato que devuelve.

Existe un método en toda clase que resulta fundamental, ya que de él depende que se instancien los objetos de la clase. Este método se denomina constructor.

El método constructor suele tener el mismo nombre de la clase. Es un método que recibe cero o más parámetros y lo más normal es que se utilicen esos parámetros para inicializar el estado del objeto que se crea.



Sabía que...

Se puede definir más de un método constructor, diferenciándolos entre sí por el número de parámetros que reciben.

Todos los métodos que no son constructores se ejecutan cuando el objeto recibe un mensaje. Ese método es enviado por un objeto de otra clase, o bien, por un objeto de la misma clase.

Los objetos que ya no son utilizados en un programa son liberados en tiempo de ejecución para desocupar el espacio de memoria. Existen lenguajes de programación en los que el programador no necesita explícitamente definir y usar el destructor porque existe un mecanismo que permite ir liberando objetos conforme no se usan.

## Mensaje

El mensaje es la petición de un objeto sobre otro para que se ejecute un método. Todo mensaje consta de 3 partes:

- Identidad del receptor: nombre del objeto que contiene el método a ejecutar.
- Nombre del método a ejecutar: solo los métodos declarados públicos.
- Lista de parámetros que recibe el método (cero o más parámetros).

Su sintaxis algorítmica es:

```
<Variable_Objeto>.<Nombre_Método> ( [<Lista de Parámetros> ] );
```

Cuando el objeto receptor recibe el mensaje, se ejecuta el método correspondiente. Tras la ejecución, el método devolverá el resultado (si procede) para que el objeto que mandó el mensaje pueda utilizarlo.



## Actividades

- 
12. Explique las diferencias entre la invocación a un procedimiento en la programación estructurada y el paso de mensaje en la programación orientada a objetos.
- 



## Aplicación práctica

Como diseñador de software, le piden hacer el diseño de un tipo de estructura de clases que controle la barrera de un parque en lenguaje C. En este caso, la barrera solo podrá elevarse y bajar. También es posible comprobar el tiempo que la barrera lleva elevada. Identifique las propiedades y los métodos necesarios para el diseño.

## SOLUCIÓN

```
class barrera {  
    int tiempo;  
    void bajar();  
    void subir();  
    int getTiempo();  
}
```

En la teoría, la herencia se corresponde con una relación de especialización entre clases de objetos, de forma que los objetos de la subclase también se consideran instancias de la superclase o clase padre. Se va a ver el siguiente ejemplo para aclarar este concepto:

### Herencia, polimorfismo y sobrecarga dinámica de métodos

La herencia es una de las claves que hacen poderoso el paradigma orientado a objetos, permitiendo reutilizar el comportamiento de una clase para definir nuevas clases hijas o subclases.

Cada subclase de una clase hereda las operaciones y variables de esta, pero permite también añadir nuevas operaciones y nuevas variables de instancia. La herencia se puede ver como una relación entre clases, y no entre objetos.

La herencia captura una forma de abstracción de nivel superior a la abstracción de datos. Esa es la clave de que sea tan importante en el paradigma.

La herencia es capaz de expresar distintas relaciones: clasificación, especialización, generalización y aproximación.

```
class Particula extends Punto {  
    // Constante de gravitación universal  
    public final static double G = 6.67e-11;  
    // Masa de la partícula  
    private double masa ;  
    // Constructor para inicializar a 0 coordenadas y masa  
    public Particula () {  
        masa = 0;  
    }  
    // Constructor para inicializar las coordenadas y la masa  
    public Particula ( double a , double b , double m ) {  
        super ( a , b );  
        masa = m;  
    }  
    // Calcula la fuerza de atracción entre esta partícula ( this )  
    // y la partícula que se pasa como argumento  
    public double atraccion ( Particula part ) {  
        return G * masa * part.masa / Math . pow( this.distancia ( part ) , 2 );  
    }  
    // Devuelve el valor de la masa  
    public double masa ( ) {  
        return masa ;  
    }  
}
```



### Nota

La herencia es siempre transitiva: una clase hija hereda todos los métodos y atributos de todas sus clases que están más arriba en la jerarquía de herencia (clases antecesoras).

Por ejemplo, toda partícula es también un punto, y hereda su comportamiento; es decir, una partícula posee coordenadas y es posible calcular la distancia entre dos partículas.

La diferencia radica en que, además, poseen masa y es posible calcular la fuerza de atracción entre dos partículas.

El **polimorfismo** es otra pieza clave en el paradigma orientado a objetos. Según este concepto, los datos pueden presentar un comportamiento polimórfico.

En efecto, un identificador puede hacer referencia a objetos de distintas clases dependiendo del contexto.



### Nota

Cuando el lenguaje permite que un identificador de variable haga referencia a objetos de distintas clases, se habla de polimorfismo sobre los datos.

Por lo general, en un programa desarrollado bajo un lenguaje estrictamente tipificado, toda variable tiene asociado un tipo en tiempo de compilación. La herencia restringiría ese tipo a objetos de la clase o de cualquiera de las clases descendientes. Esto implica que un identificador puede cambiar su tipo en tiempo de ejecución, siempre y cuando esté sujeto a la restricción de herencia.

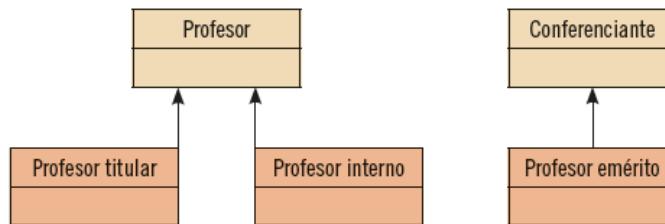
Por ejemplo, volviendo al uso de las clases punto y partícula:

```
Punto pto;
```

La declaración anterior no restringe la capacidad para almacenar solo objetos de esa clase, sino que permite cualquiera de las situaciones siguientes:

```
pto = new Punto (1 , -1) ;
pto = new Particula (1 , -1, 100) ;
pto = new PuntoAcotado ( ) ;
```

Un diseñador proporciona el siguiente diagrama entidad-relación y pide que defina todas las estructuras necesarias para la fase de implementación en lenguaje C++.



SOLUCIÓN



Aplicación práctica

```
Class Profesor {  
}  
Class Conferenciente {  
}  
Class ProfesorTitular : Profesor {  
}  
Class ProfesorInterino : Profesor {  
}  
Class ProfesorEmerito : Conferenciente {  
}
```

### Propiedades: selectores (get), modificadores (set) y referencias (let)

Es común en una clase definir métodos que consulten o modifiquen atributos que forman parte del estado de los objetos. Existen dos grandes grupos de métodos: los métodos **get**, que recuperan los valores almacenados en los atributos de los objetos, sin poder modificarlos, y los métodos **set**, que modifican esos atributos.

Un ejemplo sería una clase persona y un atributo edad. Se puede definir un método get para obtener la edad de la persona representada por objeto instancia y el método set para definir la nueva edad de esa persona si fuera necesario modificarla.



### Actividades

13. Busque ejemplos en la red de polimorfismo y comente los resultados obtenidos.

```
class persona {  
    string dni;  
    int edad;  
    ...  
    int getEdad() {  
        return edad;  
    }  
    Void setEdad(int e) {  
        edad = e;  
    }  
    ...  
}
```

Por el contrario, existen métodos que permiten la consulta y modificación a la vez. Se trata de métodos que devuelven una referencia a un objeto o estructura, de forma que, a partir de esa referencia, se puede realizar la consulta y también la modificación. Como ejemplo, se puede poner el mismo de antes, usando un puntero o referencia al atributo edad.

```
class persona {  
    int* edad;  
    int* edad() {  
        return &edad;  
    }  
}
```

## Lenguajes

Cualquier lenguaje que proporcione la forma de implementar los conceptos anteriores se denomina lenguaje orientado a objetos. Algunos autores piensan que los conceptos definidos en los apartados anteriores son una condición necesaria para que un lenguaje sea considerado orientado a objetos. Otros autores, por el contrario, piensan que un lenguaje es orientado a objetos cuando es capaz de crear estructuras de datos complejas, con operaciones sobre esas estructuras, y con la posibilidad de crear instancias de esas estructuras.

En la actualidad existen multitud de lenguajes orientados a objetos: C++, Objective C, Java, Smalltalk, Eiffel, Ruby, Python, PHP, etc.

## 4. Técnica de programación de software de gestión de sistemas

Algunas técnicas para el desarrollo de proyectos informáticos se pueden utilizar constantemente sin importar la naturaleza del producto final. Un ejemplo puede ser la reutilización de código que forma parte de otros desarrollos y que ha sido validado y verificado en las fases de pruebas de esos proyectos. En otros casos, el uso de API proporcionadas por software de terceros ayuda para acortar los tiempos en los desarrollos y a que estos sean más robustos.



### Recuerde

Una API es una interfaz de programación de aplicaciones que se usa para desarrollar programas.

### 4.1. Reutilización de código

El desarrollador con experiencia habrá llegado a muchas conclusiones durante del desarrollo de sus sistemas. Pero sin duda, alguna vez habrá pensado en utilizar código de otros proyectos en el actual. Esto es porque en informática muchos de los problemas que surgen son repetitivos y se pueden abordar con soluciones ya alcanzadas anteriormente. Por ejemplo, se pueden tener varios programas que utilizan números complejos y las funciones de suma, resta, etc., son comunes. También es posible que se esté programando un control de usuario que ya se ha usado en otros proyectos.

La reutilización de todo este código podría significar una gran disminución de los tiempos de desarrollo del proyecto, ya que no se tendría que volver a escribir el código.

Hay muchas formas de reutilizar código. Una de ellas es usar librerías precompiladas. En estos casos, el código ya compilado estará probado y será fiable.

#### Uso de bibliotecas del sistema

Las bibliotecas son una forma sencilla y versátil de formar módulos y reutilizar código. Una biblioteca es un conjunto de funciones. Es como un programa, solo que no tiene función main(). Son un conjunto de funciones que pueden ser llamadas desde otro programa o biblioteca.

La biblioteca estática es un único fichero donde se empaquetan uno o varios ficheros objetos generados por un compilador. Para su uso, en el momento de la compilación se debe especificar qué bibliotecas estáticas se han utilizado.

Por el contrario, las bibliotecas dinámicas permiten que no haga falta enlazarlas en el momento de la compilación. El enlazado se realizará en el momento de la llamada a alguna función de la biblioteca durante la ejecución del código.

#### Llamadas a utilidades y aplicaciones del sistema

Los sistemas operativos están compuestos de muchos programas diferentes para realizar procesos básicos para el ordenador. Algunos pueden ser reemplazados o eliminados por el usuario. Otros son vitales para que el sistema operativo proporcione los servicios necesarios al usuario.

Estas aplicaciones pueden ser invocadas desde los programas haciendo uso de llamadas al sistema. Con esto, se consigue que, si existe una utilidad que realiza una función necesaria para el desarrollo, reutilizarla sin necesidad de escribir código. Linux permite este tipo de comportamiento de varias formas, pero una de las más utilizadas y simples es usando tuberías. Para ello, la salida de un programa puede convertirse en la entrada de otro programa. Imagínese que el programa *main* necesita manejar el listado de ficheros de un directorio para algún fin. Se puede codificar el programa suponiendo que se pueda suministrar ese listado como lo proporciona el comando **ls** de Linux. De esta forma, se puede utilizar una tubería para que la salida de **ls -l** se convierta en la entrada del programa. La siguiente sentencia realizaría esta acción:

```
$ ls -l | programa
```

De esta forma, se pueden concatenar todas las herramientas y comandos que se quiera y componer la tarea principal del programa en función de las entradas y las salidas de otros programas.



### Actividades

14. Si se sabe que grep [cad] realiza una búsqueda de la cadena cad sobre la entrada estándar, ¿qué podría significar la sentencia ls | grep .jpg?

## 4.2. Técnicas específicas aplicables a los servicios básicos del sistema

Las características básicas de un software de sistema hacen necesario el uso de determinadas técnicas avanzadas de programación que no suelen encontrarse en otros tipos de desarrollos de forma explícita. Un ejemplo de estas técnicas es el uso de la programación concurrente. En un software de sistemas, es bastante común hacer una gestión de procesos multitarea, de forma que es necesaria la sincronización de todos estos procesos sobre los recursos que el sistema mantiene. También es frecuente realizar una gestión eficiente de la memoria que utilizan los procesos, que no es más que otro recurso con unas características particulares. De ahí surgen las técnicas de gestión de memoria para controlar la asignación y liberación de la memoria que usan los procesos. Otros ejemplos que se verán a continuación son el uso de ficheros, de dispositivos periféricos de entrada/salida y algunas técnicas que sirven para optimizar los programas, ya que, en un software de sistemas, la velocidad de ejecución suele ser un factor y crítico.

### Programación de la gestión de los procesos: multitarea, control de bloqueos (deadlock) y comunicación entre procesos

Todos los sistemas operativos actuales se basan en permitir la ejecución de varios procesos al mismo tiempo, compartiendo los recursos del sistema. A este tipo de ejecución simultánea se le denomina **multitarea**. Sin embargo, en la mayoría de las ocasiones, el sistema operativo se encuentra instalado en nodos computacionales compuestos por un solo procesador, el cual no puede realizar dos actividades a la vez. Para conseguir la capacidad de multitarea en este tipo de *hardware*, los sistemas operativos reparten el tiempo entre dos (o más) actividades, o bien utilizan los tiempos muertos de una actividad (por ejemplo operaciones de lectura/escritura de datos desde el teclado) para trabajar en la otra.

Para sistemas operativos sobre nodos con más de un procesador, la multitarea es real. Cada procesador puede llegar a ejecutar un hilo o *thread* diferente. Se está hablando de programación paralela.



### Recuerde

Un proceso es un programa que se ejecuta de forma independiente y que posee un espacio propio de memoria.

Un *thread* o hilo es un flujo secuencial simple dentro de un proceso. Es un elemento menos pesado que un proceso. La reserva de memoria es menor y el procesador puede manejarlo con más rapidez. Un proceso puede tener varios hilos ejecutándose a la misma vez.



### Ejemplo

El programa *Microsoft Edge* sería un proceso, mientras que cada una de las ventanas donde se visualiza una página web estaría formada o controlada por al menos un hilo.

En un sistema multitarea da la sensación de que realmente se está llevando a cabo la ejecución de varios procesos simultáneamente, lo cual es una gran ventaja para el usuario. Los *threads* o hilos de ejecución permiten organizar y controlar los recursos del sistema de forma que se pueda acceder a dichos recursos paralelamente. Un hilo de ejecución es como un proceso, en el sentido de que puede realizar cualquier tarea. Todos los *threads* de un

proceso comparten el espacio de memoria de ese proceso. Los *threads* pueden estar siendo ejecutados por el usuario o bien por el sistema operativo en segundo plano, los que se conoce como *daemon*. Estos *daemons* formarían parte de tareas esenciales del sistema operativo.

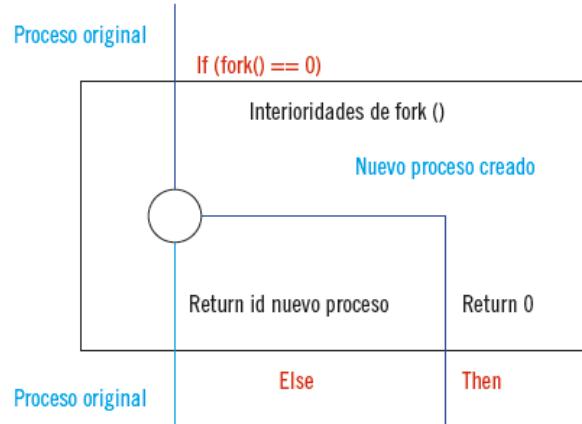
Un ejemplo de *thread daemon* podría ser aquél que comprueba permanentemente si el usuario hace un clic de ratón.

### Ejemplo de programación de procesos

En *Linux*, se puede crear un nuevo proceso durante la ejecución de otro para conseguir aprovechar la capacidad de multitarea del sistema operativo. A continuación, se va a ver un ejemplo simple.

La función en la que C crea un nuevo proceso es `fork()`. Cuando desde un proceso se ejecuta esta función, en algún lugar dentro de ella se duplican los procesos y comienza la ejecución por separado. Cuando la función retorna, ya se tienen los dos procesos creados y esta devuelve al proceso original un identificador del proceso recién creado. Sin embargo, como el proceso se ha duplicado, la función devolverá al proceso recién creado un valor 0. La imagen siguiente intenta aclarar lo que ocurre.

Representación de lo que ocurre cuando se realiza una llamada a fork()



De esta forma, cada proceso sabe si es el original o si ha sido creado, y así poder hacer cosas distintas, separando la ejecución por un **if** donde se comprueba el identificador devuelto.

A partir de este momento, se debe tener en cuenta que se ha duplicado todo el espacio de memoria. Por ello, los dos procesos tendrán todas las variables repetidas y con el mismo valor, aunque serán distintas. Cada proceso puede ahora manejar cada variable de forma independiente.

La función fork() también puede devolver -1 si se produce un error durante su ejecución. En este caso, no se creará el nuevo proceso.

Tras crear el nuevo proceso, que se conocerá como proceso hijo, el proceso original o proceso padre puede quedar a la espera de que termine la ejecución del proceso hijo. Esto se consigue usando la función **wait()**. La función **wait()** deja al proceso en espera hasta que alguno de los procesos hijo termina. Cuando se produce la finalización del proceso hijo, la función wait "devuelve" en el parámetro que se le proporcionó cómo ha sido la salida del proceso hijo, y continuará la ejecución del proceso padre.

Interioridades de fork ()

```
int estadoHijo;  
...  
wait (&estadoHijo);
```

... el proceso original (padre) se queda dormido hasta que el nuevo proceso (hijo) termina.



### Actividades

15. ¿Por qué debe ser más natural usar dos hilos para sincronizar el acceso a un recurso compartido que usar dos procesos?

## Interbloqueo (deadlock)

El interbloqueo (**deadlock**) es una anomalía que sucede cuando varios procesos intentan acceder de forma simultánea a un recurso del sistema y ninguno consigue adquirir el permiso para acceder en el mismo momento. Es decir, cuando un hilo está en estado de interbloqueo, está esperando un evento que no se producirá nunca.

Para que exista interbloqueo, se deben cumplir 4 condiciones:

- Los hilos deben reclamar derechos exclusivos a los recursos.
- Los hilos deben contener algún recurso mientras esperan otros; es decir, adquieren los recursos poco a poco, en vez de todos a la vez.
- No se pueden sacar recursos de hebras que están a la espera (no hay derecho preferente).
- Existe una cadena circular de hebras en la que cada hebra contiene uno o más recursos necesarios para la siguiente hebra de la cadena.

La solución a los interbloqueos es aplicar un mecanismo de **control de bloqueos** para eliminar el error en la lógica del programa y que los procesos implicados no se queden esperando el uno por el otro indefinidamente. Para ello, la única manera es utilizar mecanismos de sincronización de alto nivel que aseguren la realización de programas con múltiples procesos libres de estos errores.



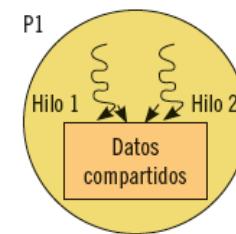
## Actividades

16. Busque un caso real donde se puedan producir este tipo de anomalías.

## Exclusión mutua

Cuando dos hilos intentan acceder a un mismo recurso no compatible, es conveniente, como se ha visto más arriba, que cada proceso acceda al mismo en un orden determinado para que no se produzca el interbloqueo o el mal funcionamiento del propio recurso. Para ello, el programador debe establecer dicho acceso por parte del hilo o proceso como una ejecución en exclusión mutua. De esta forma, un proceso podrá acceder a este recurso mientras que los demás permanecen a la espera de que finalice su uso. Cuando el proceso termine, el recurso será asignado a otro de los procesos en espera. Este mecanismo se identifica como **exclusión mutua** y al trozo de código que solo puede ser ejecutado por un proceso en cada momento se le denomina **sección crítica**.

Los hilos comparten memoria



Algunos mecanismos para conseguir este tipo de sincronización se verán a continuación

Los **semáforos** son una buena solución al problema de la exclusión mutua. Su desarrollador fue Dijkstra, que introduce el concepto de semáforo binario. Esta técnica permite resolver casi todos los problemas de sincronización entre procesos. Un semáforo

binario es un tipo abstracto de datos, identificado por un indicador (*S*) de condición que registra la disponibilidad de un recurso. Solo puede tomar dos valores: 0 y 1. Si, para un semáforo binario, *S* = 1, entonces el recurso está disponible y la tarea lo puede utilizar; si *S* = 0, el recurso no está disponible y el proceso debe esperar. Los semáforos solo permiten tres operaciones:

- Inicializar.
- Espera (wait).
- Señal (signal).

*S*: semáforo := 1;

```
task body P1 is
begin
loop
    Resto_de_código_1:
        Wait (S):
        Sección_crítica_1:
            Signal (s):
    end loop:
end P1;
```

```
task body P2 is
begin
loop
    Resto_de_código_2:
        Wait (S):
        Sección_crítica_2:
            Signal (S):
    end loop:
end P2:
```

El semáforo posee una cola de tareas asociada y solo estas tres operaciones para actuar sobre esa cola para evitar que un recurso pueda ser accedido por dos o más hilos de ejecución.



## Actividades

17. ¿Es posible que dos hilos intenten acceder al mismo recurso y que queden en espera los dos a la vez al utilizar un semáforo binario? Razone la respuesta.

Un **monitor** es una estructura de más alto nivel que el semáforo. Es un módulo que encapsula un recurso y que está compuesto de un conjunto de procedimientos. De esta forma, cualquier proceso que desee acceder al recurso tiene que pedir permiso al monitor y, cuando acabe de usarlo, devolver ese permiso. El monitor posee una interfaz, que especifica las operaciones sobre el recurso.

El uso de la programación concurrente en el desarrollo de aplicaciones hace necesaria, en muchas ocasiones, **a comunicación entre procesos**. Existen varios mecanismos de comunicación entre procesos (IPC). Los procesos que se comunican pueden ser ejecutados en un mismo equipo informático o bien en diferentes máquinas.

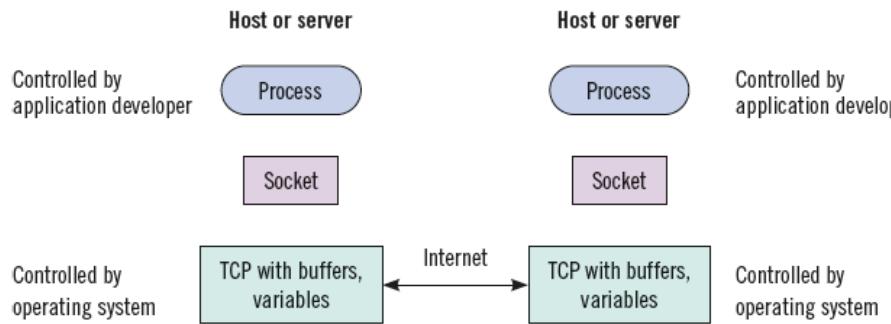
Los **pipes** o **tuberías** permiten la comunicación entre procesos utilizando la entrada y salida estándar de cada uno de los procesos. En Linux, es tan fácil como utilizar el operador | entre dos comandos.

Por ejemplo:

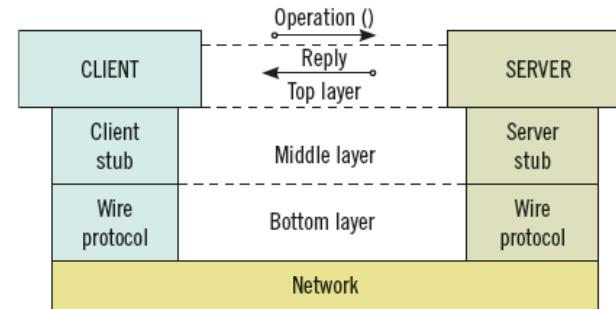
```
<$ ls | wc -l
```

En esta sentencia, el comando **ls** produce como resultado un listado de directorios y ese resultado se envía al siguiente comando **wc** a través de la entrada estándar para obtener el número de líneas y, por lo tanto, el número de elementos del directorio de trabajo.

Los **sockets** son otro mecanismo de comunicación que implica el uso de la familia de protocolos TCP/IP. La comunicación se establece entre procesos como si estuvieran en una red de ordenadores.



Las **llamadas a procedimientos remotos RPC** permiten realizar la comunicación como si el proceso estuviera llamando a una función simple. El protocolo usado para RPC hace posible que un programa realice llamadas a procedimientos ubicados en otras máquinas (procedimientos remotos).



## Programación de la gestión de memoria: jerarquías de memoria, paginación de memoria, segmentación de memoria, intercambio (swapping), compartición de memoria, seguridad y memoria virtual

La memoria es un elemento muy importante para todo programa, y aún más para un sistema operativo. Es el elemento que permite almacenar y recuperar la información, y donde la unidad de almacenamiento es el bit (*binary element*). En un sistema, nos encontramos que el acceso a cierta información de esa memoria requiere un tiempo de operación. Este tiempo, que transcurre desde que el sistema identifica la dirección que contiene la información hasta que esa información se pone a disposición del programa se conoce como **tiempo de acceso**. Introduce el texto aquí

Actualmente, se puede clasificar la memoria dependiendo del tiempo de acceso a la misma. Es lo que se conoce como una **jerarquía de memoria**.



Desde la cima de la pirámide hasta la base, se pueden encontrar memorias que comienzan siendo muy rápidas, pero de poca capacidad, hasta memorias más lentas, pero que permiten almacenar grandes cantidades de información.

Cuando un programa o proceso realiza una acción de acceso a memoria, el sistema operativo realiza una serie de operaciones que permiten extraer de la memoria esa información, la secuencia de operaciones suele pasar por varios niveles de esta pirámide hasta que el programa dispone de la información.

Cuando un proceso inicia su ejecución, el sistema operativo le asigna parte de la memoria a dicho proceso, al igual que cuando un proceso realiza una petición de memoria. La memoria que ha sido asignada al proceso es liberada al finalizar la ejecución de este.

Es, por lo tanto, primordial que exista una política correcta de **compartición de memoria** entre los procesos que se encuentran en ejecución en cada momento. Algunos de estos modelos de gestión de memoria se analizan a continuación.

Un proceso posee varios tipos de memoria que utiliza para la ejecución. Se puede encontrar la memoria que almacena el código del programa en ejecución; la memoria donde se guardan los datos estáticos, variables globales, etc.; la memoria de la pila, utilizada en las llamadas a procedimientos y/o funciones, y la memoria *heap* o de montículo, que permite la asignación dinámica de memoria dentro del proceso.

El intercambio de memoria (*swapping*) es el mecanismo a través del cual todo modelo de gestión de memoria es capaz de intercambiar la información ubicada en la memoria principal por la información almacenada en la memoria secundaria y viceversa. Es una técnica que produce lentitud en el sistema, ya que es necesario el intercambio total del proceso, aunque solo se vaya a ejecutar una parte muy pequeña del mismo.

Como se ha visto en capítulos anteriores, los procesos e hilos necesitan compartir información. Los sistemas operativos realizan un esquema de memoria compartida en el mismo espacio de memoria para los hilos de un mismo proceso. Sin embargo, no es posible compartir memoria entre procesos si no se lleva a cabo a través de algún mecanismo de comunicación entre ellos. Esto es necesario para garantizar la estabilidad de ejecución del sistema operativo. Los sistemas operativos modernos se basan en la idea de proteger y establecer **seguridad** en el espacio de direcciones de modificaciones provocadas por otros procesos.

La paginación es una técnica que permite manejar la memoria dividiéndola en secciones fijas del mismo tamaño (marcos de página). Los programas se dividen en unidades lógicas (páginas), que de forma premeditada poseen el mismo tamaño.

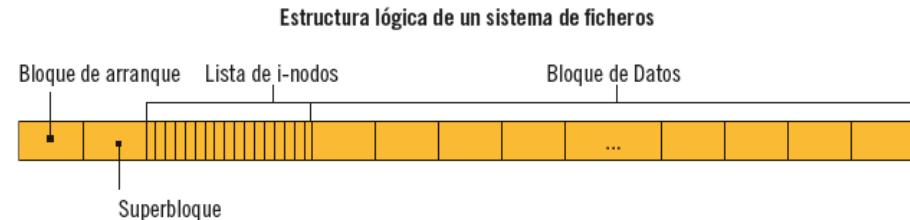
Cada proceso mantiene una tabla (PMT) denominada tabla de mapas de página. De esta forma, cada proceso tendrá en un momento dado un conjunto de páginas residentes en la memoria principal (páginas activas) y el resto en la memoria secundaria.

La segmentación de memoria consiste en agrupar la estructura del programa en bloques lógicos de tamaño variable denominados segmentos. Cada uno posee información del programa: rutinas, variables, pila, etc. Las direcciones del programa consisten en una colección de segmentos

La **memoria virtual** surge con la idea de que el tamaño de un programa, datos y pila, pueda superar la cantidad de memoria física instalada. La forma de conseguirlo es mantener el resto, que no se puede almacenar en memoria principal, en memoria secundaria (en disco). De esta forma, cuando el procesador necesite una parte que no se encuentra en la memoria RAM la recuperará del disco y continuará su ejecución normal. Este mecanismo se encuentra implementado en casi todos los sistemas operativos modernos.

### Programación de los sistemas de archivos: acceso a archivos y directorios, atributos y mecanismos de protección

En Unix, un sistema de ficheros es una secuencia de bloques lógicos con la siguiente estructura:



El bloque de arranque corresponde al primer sector y contiene el código de arranque del sistema operativo. El superbloque contiene el estado del sistema de ficheros:

- Tamaño del sistema de ficheros.
- Número de bloques libres en el sistema de ficheros.
- Lista de bloques libres disponibles y un índice al siguiente bloque libre de la lista de bloques libres.
- Tamaño de la tabla de i-nodos.
- Número de i-nodos libres en el sistema
- Lista de i-nodos libres y un índice al siguiente libre.
- La lista de i-nodos se crea cuando se crea el sistema de archivos; su tamaño y ubicación en el disco son fijos.

Un i-nodo representa a un fichero o directorio del sistema de ficheros y mantiene la información relevante de este:

- Identificador del propietario del fichero: UID, GID.
- Tipo de fichero (regular, directorio, dispositivo, flujo).
- Permisos de acceso.
- Tiempos de acceso: última modificación de fichero, acceso y modificación del i-nodo.

- Contador de enlaces.
- Tabla con las direcciones de los datos en disco del fichero, enlaces a bloques físicos dentro del disco.
- Tamaño.

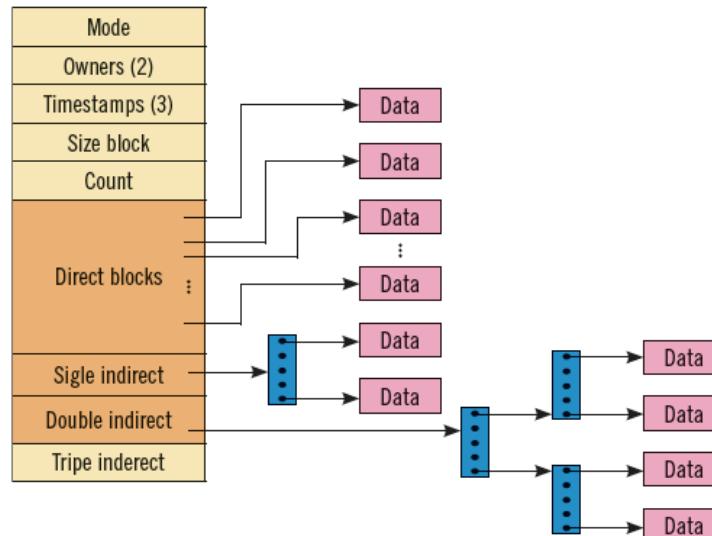


## Actividades

18. ¿Para qué podría servir almacenar el identificador UID de un fichero en un i-nodo?

El núcleo del sistema operativo también mantiene un campo con el estado de cada i-nodo, indicando si está bloqueado, si un proceso está esperando a que el i-nodo se desbloquee, si el fichero es un punto de montaje o si la copia del i-nodo difiere del fichero en memoria masiva o principal.

**Estructura lógica de los i-nodos en un sistema de ficheros**



El i-nodo mantiene una estructura de referencias hacia los bloques que componen el fichero. Estos bloques se asignan de forma dinámica, a medida que se necesita y no tienen por qué estar contiguos.

### Estructuras lógicas que mantiene cada proceso para la gestión de los ficheros abiertos

Tablas de descriptores de archivos de usuario

	(Proc. A)
0	
1	
2	
3	
4	
5	
.	

Tabla de archivos

	(Proc. B)
0	
1	
2	
3	
4	
5	
.	

Tabla de Inodos

Cuenta 1	leer-escribir
.	
Cuenta 1	leer
.	
Cuenta 1	escribir
.	
Cuenta 1	escribir

puntero de lectura/escritura, los permisos de acceso, el modo de apertura de fichero y un contador de las entradas en la tabla de descriptores que referencian al fichero.



### Sabía que...

Cuando se crea un proceso, el sistema operativo abre tres entradas en esta tabla, correspondiendo a stdin, stdout y stderr, descriptores que se utilizan para la entrada de datos desde el teclado (stdin) y la salida de datos por pantalla (stdout y stderr).

Además, el proceso mantiene una tabla de descriptores, donde cada entrada se compone de un puntero a la tabla de ficheros. El valor del identificador de esta entrada es usado por el programa para manejar el fichero.

El sistema de ficheros garantiza el acceso a ficheros y directorios. Sin embargo, también mantiene cierta información referente a cada fichero y directorio conocida como metadatos. Se puede decir que los atributos de los ficheros y directorios se encuentran incluidos en los metadatos. Los atributos más típicos de un fichero son:

- Nombre: identificador del fichero a vista del usuario.
- Tipo: tipo de la información que almacena el fichero.
- Localización: identificador que referencia al i-nodo en el sistema de ficheros, para conocer la ubicación exacta del fichero.
- Tamaño: cantidad de información que es almacenada por el fichero.
- Protección: atributos que especifican el usuario que tiene autorización de lectura, escritura, ejecución, etc., del fichero.
- Día y hora: contiene la fecha y hora del último acceso, creación, etc.
- Usuario y grupos: identificador del usuario y grupo al que pertenece el fichero.

Cada proceso, cuando se abre un fichero, añade una entrada en la tabla de ficheros asociada. El contenido de cada entrada posee el puntero al i-nodo correspondiente, un

Ya se ha visto que un proceso puede acceder al contenido de un fichero a través de las llamadas al sistema. En resumen, se tienen las siguientes llamadas como principales:

- **Open**, que devuelve un descriptor de fichero para su manejo.
- **Close**, que cierra el manejo del fichero y libera los recursos.
- **Read**, que permite leer un conjunto de datos del fichero.
- **Write**, que permite escribir un conjunto de datos en el fichero.
- **Lseek**, que permite desplazar el cursor del fichero para leer desde otra ubicación del propio fichero.
- **lctl**, que permite, entre otras cosas, acceder a los atributos del fichero.

El conjunto de estas funciones es lo que se conoce como interfaz de acceso al fichero.

El directorio, como fichero de carácter especial, también posee información a través de los metadatos y una interfaz de acceso. En concreto, los atributos son:

- **Nombre**: identificador del directorio para el usuario.
- **Tamaño**: cantidad de ficheros que posee o almacena.
- **Protección**: información sobre el control de lectura, escritura y ejecución.
- **Día y hora**: instante de tiempo que registra el último acceso, creación, etc.
- **Identificación de usuario**: nombre del usuario que es propietario del directorio.

En cuanto a funciones de acceso a nivel de programación, se puede hacer una recopilación genérica de las más importantes:

- **Mkdir**, que permite crear un directorio.
- **Rmdir**, que permite eliminar un directorio.
- **Chdir**, que cambia el directorio de trabajo.
- **Opendir**, que devuelve un descriptor de directorio para manejarlo.
- **Closedir**, que cierra el descriptor y libera los recursos.

- **Readdir**, que permite leer el contenido de un directorio.

Los principales mecanismos de **protección** entre ficheros y directorio provienen de la política de seguridad que implemente el sistema operativo, en su defecto, el sistema de ficheros que use el sistema operativo. En concreto, *Linux* permite una protección basada en permisos para tres grupos de usuarios: el usuario dueño del fichero o directorio, los usuarios que pertenecen al mismo grupo del dueño y el resto de usuarios. Para cada grupo, se establecen tres tipos de permisos: lectura, escritura y ejecución. De esta manera, el usuario dueño puede proteger fácilmente la información de cualquier otro usuario.



## Actividades

19. ¿Sería posible acceder a un fichero almacenado en el disco duro si no existiera su ícono asociado? Razone la respuesta.

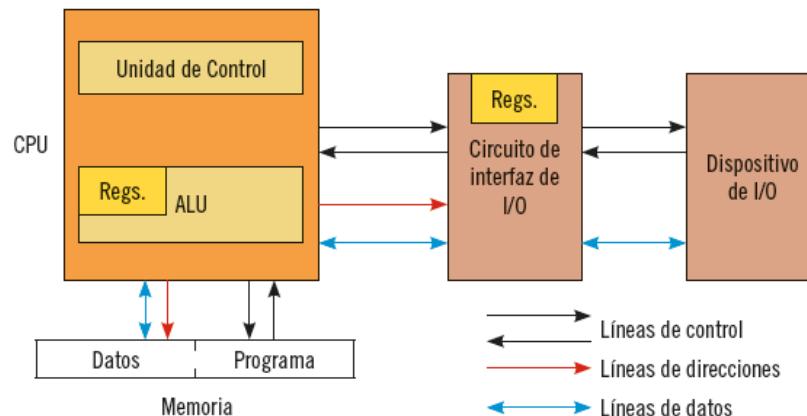
**Programación de los sistemas de entrada y salida: gestión de interrupciones, acceso directo a memoria (DMA), puertos de entrada/salida y asignación de memoria**

Los dispositivos de entrada/salida, además de ser una forma de interacción entre la máquina y los humanos, permiten descargar a la CPU de trabajo adicional. El problema radica en que estos dispositivos tienen velocidades muy variadas y, por lo tanto, no pueden aceptar datos enviados a diferentes velocidades. Por ello, tiene que existir alguna forma de coordinar el envío de datos entre la CPU y los periféricos.

Existen dos formas básicas de asegurar esta coordinación, dependiendo del tipo de procesador en el que se base la arquitectura. Ambos tipos de acceso requieren que la CPU realice el movimiento de los datos entre el periférico y la memoria principal:

1. Conexión mapeada en memoria (*memory-mapped I/O*), que usa "direcciones especiales" en el espacio normal de direcciones. En este caso, la memoria del dispositivo se refleja en la memoria principal.
2. Conexión mediante puertos especiales de entrada/salida (*I/O-mapped I/O*), que usa "instrucciones especiales" de entrada/salida y un espacio de direcciones específico para el dispositivo.

Comunicación entre los dispositivos E/S y la CPU



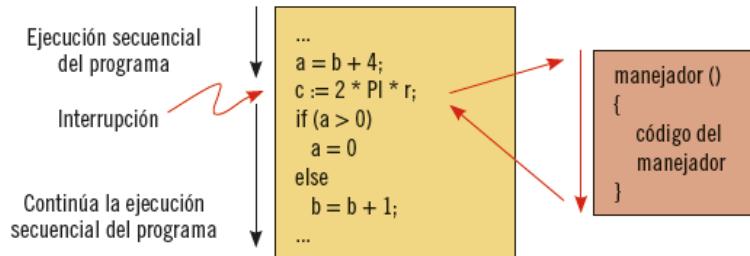
Los puertos I/O aparecen como celdas de memoria para la CPU, pero tienen conexiones hacia el dispositivo real. Se clasifican habitualmente en tres tipos diferentes:

- Puertos de solo lectura (*read-only* o *input port*).
- Puertos de solo escritura (*write-only* o *output port*).
- Puertos de lectura-escritura (*read-write* o *input-output*).

La **interrupción** es el mecanismo mediante el cual es posible interrumpir la ejecución del programa ejecutado por la CPU. El dispositivo es el que avisa a la CPU de que ha ocurrido un evento e inicia la transferencia de los datos: nuevo dato disponible, posibilidad de enviar nuevo dato, cambio en una línea de estado, error, etc.

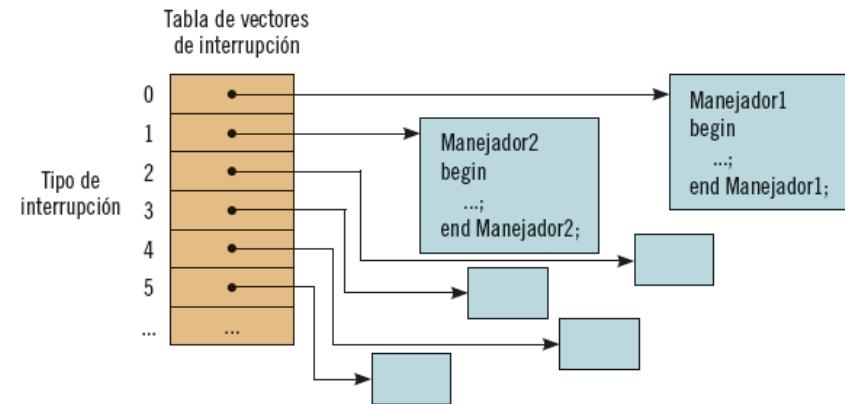
Tras la interrupción, se ejecuta la rutina de servicio de interrupción (ISR), quedando el programa suspendido hasta que retorna. Las interrupciones que se pueden producir se identifican mediante un valor numérico.

Momento en que se produce la interrupción



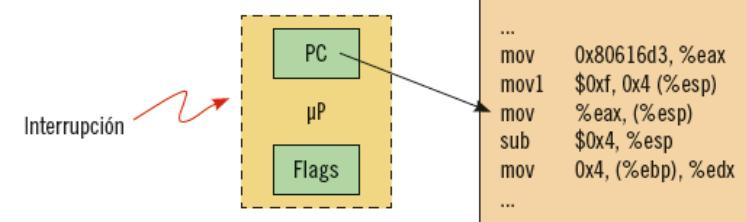
El sistema mantiene una **tabla de vectores** de interrupción que establece el enlace entre cada tipo de interrupción y su ISR asociada.

#### Estructura lógica que mantiene el manejador de interrupciones

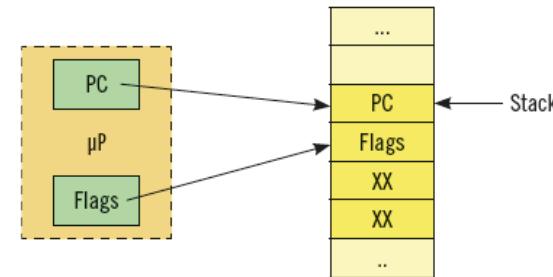


La secuencia de pasos para la gestión de las interrupciones es la siguiente:

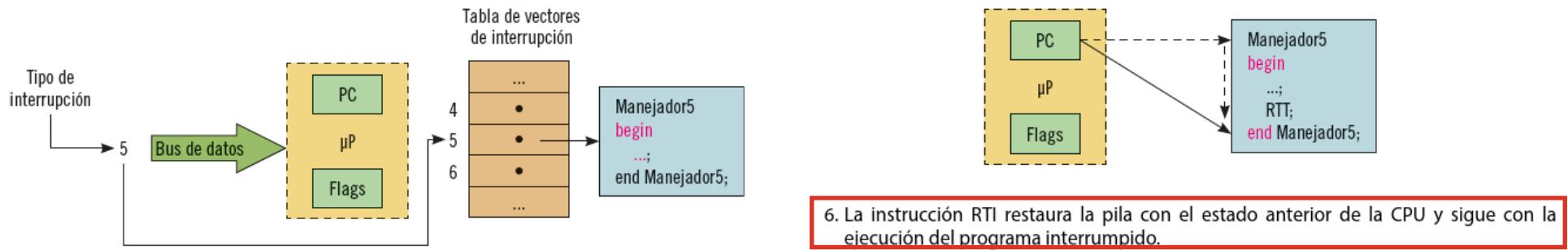
1. Se genera una interrupción. El procesador termina de ejecutar la instrucción del ensamblador en proceso.



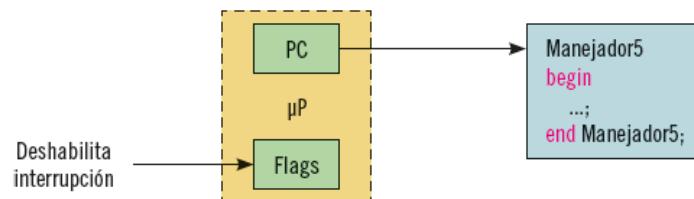
2. Se guarda en la pila el estado del procesador (todos los registros).



3. La CPU lee el tipo de interrupción y busca la referencia a la rutina ISR de esa interrupción en la tabla de vectores de interrupción.

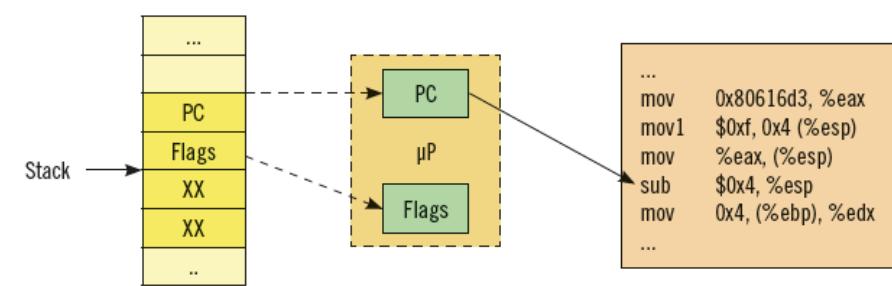


4. La CPU carga la dirección de comienzo de la ISR y deshabilita la gestión de interrupciones para que no se solapen.



5. Se ejecuta la ISR hasta llegar a la instrucción de retorno de interrupción (RTI).

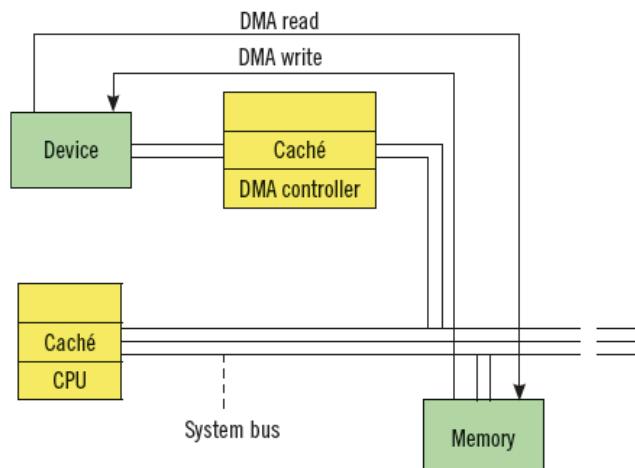
6. La instrucción RTI restaura la pila con el estado anterior de la CPU y sigue con la ejecución del programa interrumpido.



DMA (*Direct Memory Access*) es la capacidad que permite a los dispositivos de E/S acceder directamente a la memoria del sistema sin necesidad de que dicho acceso sea realizado por la CPU, liberándola así de la carga de procesamiento.

Las transferencias DMA, llevadas a cabo por el controlador DMA, consisten en copiar bloques de memoria entre dispositivos.

### Accesos de los dispositivos a memoria usando DMA



Para la conexión física de los distintos dispositivos de entrada/salida, los fabricantes de ordenadores diseñaron lo que se conoce como **puertos de entrada/salida**. Estos puertos o ranuras físicas permiten realizar las transferencias de información entre el procesador y el periférico que se conecta.

Antiguamente, los dos puertos más importantes y comunes a casi todos equipos informáticos eran:

- El **puerto paralelo**, que daba la posibilidad de transferir varios bits simultáneamente a través de un mismo enlace físico.

- El **puerto serie**, que se basaba en una única línea, a través de la cual se transmitían en secuencia los bits que formaban la información.

Hoy en día, el puerto USB (*Universal Serial Bus*) define una interfaz a nivel físico y lógico que se ha consolidado como estándar de facto y que permite distintas velocidades de transferencia a través de un mismo medio.

Sin embargo, se ha hablado de los puertos a nivel de *hardware*. En una arquitectura de PC, cada dispositivo de entrada/salida cuenta con una serie de direcciones que permiten la comunicación. Esas direcciones (a nivel lógico) se denominan **puertos de E/S**. Se cuenta con hasta 65.536 puertos de 8 bits en una arquitectura PC. La CPU establece la comunicación con el periférico a través de esas direcciones, de dos formas distintas: de forma directa, mediante operaciones de ensamblador, o bien mapeando los puertos en memoria y usando instrucciones de acceso de memoria sobre esas direcciones mapeadas.

El segundo método requiere una **asignación de memoria** que permita el mapeo adecuado de las direcciones de comunicación.



### Actividades

20. ¿Por qué cree que es necesaria una comunicación directa entre un periférico de entrada/salida y la CPU?

## Programación de la seguridad: control de variables, control de desbordamiento de búferes, aserciones, precondiciones y postcondiciones

El **desbordamiento de búfer** es una técnica de *software* que hace que se ejecute un código arbitrario de forma no autorizada. Puede afectar tanto a aplicaciones de usuario como al propio núcleo del sistema operativo.

Este problema se da con más frecuencia en determinados lenguajes de programación (C, C++, etc.) que no comprueban los límites en el uso de variables. Por lo general, se basan en la manipulación de la pila del programa para conseguir ejecutar código que no pertenece a ese programa y obtener permisos de ejecución del sistema completo.

Algunas de las posibilidades para evitar este tipo de ataques son:

- Evitar las funciones de vulnerabilidad que los programadores mal intencionados pueden explotar para sobrellevar los búferes. En C, minimizar el uso de **strcpy()**, **strcat()**, **sprintf()** y **vsprintf()**, que no realiza la comprobación de límites. Evitar **gets()**, que no especifica cuántos caracteres se van a leer. Y, si se utiliza **scanf()**, asegurarse de especificar un ancho para el formato %. Esto es lo que se conoce como **control de variables**.
- Evitar que el contenido de la pila se pueda ejecutar. Algunos compiladores, como GCC, usan funciones trampolín (pequeñas porciones de código generadas en tiempo de ejecución que se almacenan en la pila desde donde se ejecutan).
- Implementar herramientas de compilación para proporcionar advertencias cuando se usa código demasiado permisivo. Algunas de estas herramientas prohíben que las personas ajenas tengan acceso a direcciones ilegales.

- Instalar herramientas como *libsane*, que comprueba que la dirección más cercana al retorno en la pila no se sobreesciba y se produzca el desbordamiento.

Las **aserciones** son instrucciones que se usan durante la ejecución del programa y que ayudan a comprobar condiciones de este a medida que se ejecuta y evitar que se produzcan situaciones que no deberían estar permitidas. Sin embargo, no pueden ser un mecanismo único de control de errores. Los errores que se pueden esperar deben ser gestionados mediante un mecanismo de más alto nivel como el manejo de excepciones, y las aserciones para los errores, que nunca deberían de ocurrir.

Por ejemplo: si se ha desarrollado un programa de facturación en el que se ha limitado el número de facturaciones a un máximo diario, asegurar que ese límite no supere en ningún punto del programa. Aquí es donde es útil una aserción. Si el límite es de 10.000 facturaciones diarias, se puede usar la siguiente sentencia:

```
...
assert(nFacturaciones <= 10000);
...
```

Esta instrucción detendrá el programa cuando sea falsa la condición de que el número de facturaciones es inferior a 10.000.

Otro mecanismo que ayuda a asegurar los programas es el uso de **precondiciones** y **postcondiciones**.

La precondición es un conjunto de condiciones que se deben cumplir antes de la ejecución de un trozo de código. En general, se utilizan para indicar las condiciones que deben satisfacer los datos de entrada para que el programa pueda cumplir su tarea.

Por ejemplo: si se va a utilizar una división entre enteros, se debe evitar que el denominador sea 0. El problema del uso de precondiciones es que a veces se puede llegar a ser demasiado restrictivo, lo que provoca cierta inflexibilidad en el programa porque se están dejando de tratar algunos casos posibles.

Por el contrario, la postcondición es un aserto que indica el estado en el que debe quedar el sistema cuando se ejecuta cierto código. Por lo general, se refiere a los valores de salida de la función que se asegura.



### Ejemplo

Se está desarrollando un sistema de control de un cajero automático. En concreto, a la hora de retirar fondos. Para ello, se utiliza la función `retirarFondos(Cliente, Cantidad)`.

Analizando el programa, se llega a las siguientes precondiciones y postcondiciones en la ejecución de esta función.

Precondiciones:

- I El titular de la cuenta bancaria (Cliente) es el cliente que ha introducido el pin correspondiente a su tarjeta de débito.
- I La cantidad solicitada para retirar está disponible en el interior del cajero y además cumple la restricción de cantidad disponible diaria para el cliente según el servicio contratado.
- I La cantidad es positiva, mayor que el mínimo posible.

Postcondiciones:

- I Tras la ejecución, la cantidad total de la cuenta bancaria es menor que la cantidad total antes de la ejecución de la función.
- I La cuenta bancaria no queda en números rojos.



### Actividades

21. ¿Podría un buen uso de precondiciones y postcondiciones evitar el problema del desbordamiento de búfer? Razone la respuesta.

### 4.3. Técnicas de optimización

La compilación es un proceso al que le afecta directamente el tipo de *hardware* y el sistema operativo que se está utilizando. El compilador tiene capacidad para mejorar el resultado de los programas en función del *hardware* y el *software*. Sin embargo, en muchas ocasiones no consigue producir código tan bueno como el que puede llegar a escribir un buen programador que piensa en la optimización. Esto es porque el programador escribe código en un lenguaje de alto nivel, que por su naturaleza puede generar código ineficiente. A pesar de ello, es posible realizar transformaciones sobre el código para mejorar su rendimiento; a estas transformaciones se les denomina **optimizaciones**.



### Sabía que...

Algunos compiladores permiten especificar opciones de optimización respecto a la plataforma donde se ejecutan los programas.

Los tipos de optimizaciones pueden clasificarse según se hable de técnicas que dependen o no del tipo de máquina que ejecutará el programa. También se pueden clasificar en técnicas locales, aquellas que analizarán pequeños trozos de código, y técnicas globales, que tienen en cuenta todo el código.

El análisis de un programa comienza dividiéndolo en bloques básicos o secuencias de sentencias, en las cuales el flujo comienza y termina sin que se produzcan saltos o paradas. Las optimizaciones se pueden dividir en:

- Optimizaciones que no alteran la estructura del código o flujo de control de programa:
  - Eliminar expresiones que son comunes.
  - Eliminar código que no se ejecuta nunca.
  - Renombrar variables temporales.
  - Intercambiar sentencias dependientes adyacentes.
- Transformaciones algebraicas. Se basan en la simplificación de expresiones y el reemplazo de operaciones costosas por otras menos costosas en tiempo de cómputo.

Existen otras optimizaciones locales que permiten reducir a nivel de instrucción. Se les denomina optimizaciones **peephole**. Algunas de las más usuales son:

- Eliminar instrucciones redundantes.
- Optimizar el flujo de control.
- Simplificar expresiones algebraicas.
- Usar instrucciones de máquina específicas.

Otras técnicas se centran en la optimización a nivel de **bucles**:

- Movimiento de código.
- Eliminación de variables que son inducidas.
- Sustitución de variables costosas por otras menos costosas.
- Expansión de código (*loop unrolling*).
- Unión de bucles (*loop jamming*).



### Actividades

22. Elabore un diagrama con los principales tipos de optimizaciones vistos hasta ahora.

## 5. Control de calidad del desarrollo del software de gestión de sistemas

El control de calidad en el *software*, denominado SQA (*Software Quality Assurance*), está compuesto de una gran variedad de actividades que se aplican a todas las fases del desarrollo del proyecto: se utilizan métodos y herramientas durante el análisis, el diseño, la codificación y la prueba, se aplican revisiones técnicas formales en cada paso y durante la fase de pruebas se utilizan estrategias de prueba multiescalada. Además, se evalúan los cambios realizados y el control de documentación. Al mismo tiempo que se asegura que el sistema se ajusta a los estándares de desarrollo y, lo más importante, se aplican métricas para la medida de la calidad.

Existen muchas definiciones de calidad del *software*, pero, sin duda, todas están relacionadas directa o indirectamente con el cumplimiento de los requerimientos de la fase de análisis.



### Importante

El criterio de cumplimiento de los requerimientos es un factor importante, pero no es el único, ya que existen condiciones implícitas que el *software* debe cumplir, como son eficiencia, seguridad, integridad, consistencia, etc.

Sin embargo, no se puede afirmar que un *software* es de alta calidad cuando cumple con los requerimientos del usuario, ya que podría darse alguno de estos casos:

- No es eficiente al utilizar los recursos de la máquina (programas muy lentos).
- No es confiable; los resultados que entrega varían, no son siempre iguales al procesar los mismos datos.
- No es fácil de utilizar.
- No es seguro.
- No es fácil hacerle el mantenimiento.

Por lo tanto, la calidad en el *software* se puede ver como una mezcla compleja de factores que dependen en gran medida del tipo de *software* que se está desarrollando.

Los tres puntos claves a tener en cuenta cuando se habla de la calidad en el *software* son:

1. Si se produce falta de concordancia entre los requisitos del usuario y la funcionalidad final del sistema, se puede decir que existe una disminución en la calidad final del producto.

2. El uso de estándares que guían la manera en la que deben aplicarse los mecanismos de ingeniería sobre el *software* también es un factor determinante en la cantidad de calidad del *software*.
3. Por último, los requisitos implícitos que a menudo no se mencionan (eficiencia, facilidad de uso, facilidad de mantenimiento) también determinan la calidad final del sistema

En cuanto a los **factores y criterios que determinan la calidad** en el *software*, los elementos básicos empleados para medir la calidad en el *software* se denominan factores. Se pueden encontrar dos grandes categorías: aquellos que se pueden medir de forma directa y aquellos que deben medirse en función de otros valores subjetivos.

Los factores de calidad también se pueden clasificar según se relacionen con las características operacionales, la capacidad de soportar los cambios o la adaptabilidad a nuevos entornos. Pero, en la mayoría de los casos, son difíciles de medir. Una forma de facilitar esa medición es dividir los factores en sus características independientes. De esta forma, los criterios ofrecen una definición más concreta y completa de los factores, ayudan a identificar la interrelación entre ellos y pueden ser medibles y verificables a través de métricas (valor numérico de la medida de calidad).

### 5.1. Métricas aplicables

Una métrica es una magnitud que permite su revisión o evaluación y que corresponde a un posible atributo o requerimiento del *software*. Por ejemplo: un criterio del factor de calidad "Eficiencia" es "Ejecución eficiente" y un atributo de este sería "datos agrupados para procesamiento eficiente". En una revisión para evaluar esta métrica, se podría formular la siguiente pregunta: "¿Están los datos agrupados para permitir un procesamiento eficiente?" La respuesta será 1 si es afirmativa y 0 si es negativa. De esta forma, el valor de la métrica

para este factor de calidad analizado será la suma de todos los valores obtenidos por criterios/subcriterios divididos por el número de preguntas aplicadas.

Existen conjuntos de métricas que han sido elaboradas para multitud de proyectos *software* y que se convierten en grandes plantillas que pueden ser útiles para los proyectos que empiezan a desarrollarse.

A continuación, se muestran algunas de estas métricas.

Métricas	
Factor calidad	Definición
Cumplimiento	El grado en que un programa satisface sus especificaciones y consigue los objetivos de la misión encomendada por el cliente
Fiabilidad	Grado en que se puede esperar que un programa lleve a cabo sus funciones esperadas con la precisión requerida
Eficiencia	Cantidad de recursos de <i>hardware</i> y de código requerido por un programa para realizar su función
Integridad	Grado en que puede controlarse el acceso al <i>software</i> o a los datos por personas no autorizadas

Facilidad de uso	Esfuerzo requerido para aprender, trabajar, preparar la entrada e interpretar la salida de un programa
Facilidad de mantenimiento	Esfuerzo requerido para localizar y arreglar un error en un programa
Facilidad de prueba	Esfuerzo requerido para probar un programa de forma que se asegure que realiza la función requerida
Portabilidad	Esfuerzo requerido para transferir el programa desde una configuración de <i>hardware</i> o sistema operativo a otro
Reusabilidad	Grado en que un programa (o partes de él) se pueden reutilizar en otras aplicaciones
Facilidad de interoperación	Esfuerzo requerido para acoplar un sistema a otro
Facilidad de auditoría	Facilidad con que se puede comprobar la conformidad con los estándares
Exactitud	Precisión en los cálculos y el control
Normalización de las comunicaciones	Grado en que se usan el ancho de banda, los protocolos y las interfaces estándar

Compleitud	Grado en que se ha conseguido la total implementación de las funciones requeridas
Concisión	Lo compacto que es el programa en términos de líneas de código
Consistencia	Uso de un diseño uniforme y de técnicas de documentación
Estandarización datos	Uso de estructuras de datos y de tipos de datos estándar
Tolerancia de errores	Daño que se produce cuando el programa encuentra un error
Eficiencia en la ejecución	Rendimiento en tiempo de ejecución de un programa
Facilidad de expansión	Grado en que se puede ampliar el diseño arquitectónico de datos
Generalidad	Amplitud de aplicación potencial de los componentes del programa
Independencia del hardware	Grado en que el <i>software</i> es independiente del hardware que usa

Instrumentación	Grado en que el programa muestra su propio funcionamiento e identifica errores que aparecen
Modularidad	Independencia funcional de los componentes del programa
Facilidad de operación	Grado de facilidad de operación
Seguridad	Disponibilidad de mecanismos que controlen o protejan los programas o los datos
Autodocumentación	Grado en que el código fuente proporciona documentación significativa
Simplicidad	Grado en que un programa puede ser entendido sin dificultad
Facilidad de trazo	Posibilidad de seguir la pista de la representación del diseño de los componentes reales del programa hacia atrás
Formación	Grado en que el <i>software</i> ayuda a permitir que nuevos usuarios apliquen el sistema



## Actividades

23. Explique qué quiere decir que un programa sea fiable, eficiente y portable.

### 5.2. Verificación de requisitos

Como se ha visto, la captura de los requerimientos tiene como principal objetivo llegar a un entendimiento profundo de lo que debe y no debe hacer el sistema que se está desarrollando.



## Recuerde

Un requerimiento especifica qué es lo que el sistema debe hacer, funciones y propiedades esenciales y deseables.

El análisis de requisitos es una de las fases más importantes para que los objetivos del proyecto se cumplan. De ella depende en gran medida que no se cometan errores en fases posteriores, lo que provoca un incremento importante del coste final.

La validación de requerimientos permite comprobar que estos definen el sistema que el cliente busca, asegurando que son completos, exactos y consistentes. Sirve como garantía y firma de que lo descrito es lo que el cliente necesita. Es importante porque la detección de errores en esta fase reduce drásticamente el coste, como se ha explicado anteriormente.

Cuando se produce un cambio en los requisitos durante el desarrollo del sistema, el coste se dispara porque es necesario cambiar el diseño, modificar la implementación y probarlo de nuevo. La etapa de validación plantea las siguientes **Tareas de verificación**:

- Verificación de validez. Este conjunto de técnicas ayuda a identificar la necesidad de incluir funciones adicionales o diferentes de las que pidieron los *stakeholders*.
- Verificación de consistencia. Ayuda a determinar si algún requerimiento entra en conflicto con otros.
- Verificación de completitud. Comprueba que el documento de requerimientos define todas las funciones y restricciones de los *stakeholders*.
- Verificación de realismo. La tecnología que se usará, el presupuesto y el tiempo estimado aseguran que los requisitos pueden cumplirse.
- Verificabilidad. La redacción de los requisitos tiene que ser la adecuada para permitir que se pueda escribir el conjunto de pruebas para demostrar que se cumplen o no.

Las revisiones de requerimientos, la construcción de prototipos y la generación de casos de prueba son algunas de las técnicas de validación más utilizadas.

En una **revisión de requerimientos**, el cliente o cualquier *stakeholder*, involucrado de manera formal o informal, debe poder verificar que el documento de requerimientos no presente anomalías ni omisiones. En una revisión de este tipo, los revisores deben prestar especial atención a los siguientes puntos:

- Que el requerimiento sea verificable de forma realista.
- Que todos los usuarios finales comprendan correctamente cada requerimiento.
- ¿Qué dependencia existe entre cada requerimiento y los demás? Es decir, ¿un cambio en un requerimiento puede causar efectos de gran escala en los otros requerimientos del sistema?



## Acciones

23. Explique qué queremos decir que un programa sea fiable, eficiente y portable.

## 5.2. Verificación de requisitos

Como se ha visto, la captura de los requerimientos tiene como principal objetivo llegar a un entendimiento profundo de lo que debe y no debe hacer el sistema que se está desarrollando.



## Recuerde

Un requerimiento especifica qué es lo que el sistema debe hacer, funciones, propiedades esenciales y deseables.

El análisis de requisitos es una de las fases más importantes para que los objetivos del proyecto se cumplan. De ella depende en gran medida que no se comentan errores en fases posteriores, lo que provoca un incremento importante del coste final.

La validación de requerimientos permite comprobar que estos definen el sistema que el cliente busca, asegurando que son completos, exactos y consistentes. Sirve como garantía y firma de que lo escrito es lo que el cliente necesita. Es importante porque la detección de errores en esta fase reduce drásticamente el coste, como se ha explicado anteriormente.

Cuando se produce un cambio en los requisitos durante el desarrollo del sistema, el coste se dispara porque es necesario cambiar el diseño, modificar la implementación y probarlo de nuevo. La etapa de validación plantea las siguientes fases de verificación:

- Verificación de validez. Este conjunto de técnicas ayuda a identificar la necesidad de incluir funciones adicionales o diferentes de las que pidieron los stakeholders.
- Verificación de consistencia. Ayuda a determinar si un requerimiento entra en conflicto con otros.
- Verificación de completitud. Comprueba si el documento de requerimientos define todas las funciones y restricciones para los stakeholders.
- Verificación de realismo. La tecnología que se usará, el presupuesto y el tiempo estimado aseguran que los requisitos pueden cumplirse.
- Verificabilidad. La redacción de los requisitos tiene que ser la adecuada para permitir que se pueda escribir el conjunto de pruebas para demostrar que se cumplen o no.

Las revisiones de requerimientos, la construcción de prototipos y la generación de casos de prueba son algunas de las técnicas de validación más utilizadas.

En la revisión de requerimientos, el cliente o cualquier stakeholder, involucrado de manera formal o informal, debe poder verificar que el documento de requerimientos no presente anomalías ni omisiones. En una revisión de este tipo, los revisores deben prestar especial atención a los siguientes puntos:

- Que el requerimiento sea aplicable de forma realista.
- Que todos los usuarios finales comprendan correctamente cada requerimiento.
- ¿Qué dependencia existe entre cada requerimiento y los demás? Es decir, ¿un cambio en un requerimiento puede causar efectos en una escala en los otros requerimientos del sistema?



## Actividades

23. Explique qué quiere decir que un programa sea fiable, eficiente y portable.

## 5.2. Verificación de requisitos

Como se ha visto, la captura de los requerimientos tiene como principal objetivo llegar a un entendimiento profundo de lo que debe y no debe hacer el sistema que se está desarrollando.



## Recuerde

Un requerimiento especifica qué es lo que el sistema debe hacer, funciones y piedades esenciales y deseables.

El análisis de requisitos es una de las fases más importantes para que los objetivos del proyecto se cumplan. De ella depende en gran medida que no se comentan errores en fases posteriores, lo que provoca un incremento importante del coste final.

La validación de requerimientos permite comprobar que estos definen el sistema que el cliente busca, asegurando que son completos, exactos y consistentes. Sirve como garantía y firma de que lo que se hace es lo que el cliente necesita. Es importante porque la detección de errores en esta fase reduce drásticamente el coste, como se ha explicado anteriormente.

Cuando se produce un cambio en los requisitos durante el desarrollo del sistema, el coste se dispara porque es necesario cambiar el diseño, modificar la implementación y probar de nuevo. La etapa de validación plantea las siguientes fases de verificación:

- Verificación de validez. Este conjunto de técnicas ayuda a identificar la necesidad de incluir funciones adicionales o diferentes de las que pidieron los stakeholders.
- Verificación de consistencia. Ayuda a determinar si algún requerimiento entra en conflicto con otros.
- Verificación de completitud. Comprueba si el documento de requerimientos define todas las funciones y restricciones que los stakeholders.
- Verificación de realismo. Valora la tecnología que se usará, el presupuesto y el tiempo estimado asegurando que los requisitos pueden cumplirse.
- Verificabilidad. La redacción de los requisitos tiene que ser la adecuada para permitir que se pueda escribir el conjunto de pruebas para demostrar que se cumplen o no.

Revisar y revisar, la construcción de prototipos y la generación de casos de prueba son algunas de las técnicas de validación más utilizadas.

En una **revisión de requerimientos**, el cliente o cualquier stakeholder, involucrado de manera formal o informal, debe poder verificar que el documento de requerimientos no presente anomalías ni omisiones. En una revisión de este tipo, los revisores deben prestar especial atención a los siguientes puntos:

- Que el requerimiento sea verificable de forma realista.
- Que todos los usuarios finales comprendan correctamente cada requerimiento.
- ¿Qué dependencia existe entre cada requerimiento y los demás? Es decir, ¿un cambio en un requerimiento puede causar efectos de gran escala en los otros requerimientos del sistema?



## Recuerde

Un **stakeholder** es toda parte interesada en el desarrollo del proyecto, ya sea persona u organización.

La **construcción de prototipos** es otra forma de verificación. Mostrar un sistema de prueba con los requisitos que se quieren verificar y probar es un medio para que los usuarios finales y/o clientes puedan experimentar y comprobar cómo el modelo puede cumplir con las necesidades reales.

Debido a la naturaleza de los requerimientos, estos deben poder probarse. Por ello, es posible realizar una generación de **casos de prueba**. Si, en algún momento, una prueba es difícil o imposible de diseñar, lo más normal es que signifique que los requerimientos no están bien definidos.

## Técnicas de verificación automática para especificación de requisitos

Las técnicas de verificación automática proponen herramientas para verificar los documentos de requisitos y encontrar posibles defectos. Es importante que antes de continuar en una verificación automática, se considere el lenguaje de especificación a utilizar en la definición de los requisitos. Se podrá utilizar un lenguaje natural, aunque también un lenguaje formal. Las técnicas que se explican a continuación suponen el uso del lenguaje natural como punto de partida.

## Verificación automática de especificaciones de requisitos en lenguaje natural

Gervas propone un método de dos fases para la identificación automática de defectos en las especificaciones. En una primera se define el estilo, la estructura y el lenguaje para el documento de requisitos. Se eligen las propiedades a evaluar y los modelos que sirven para evaluar dichas propiedades. En una segunda fase, o fase de producción, se aplica un analizador sobre el documento de requisitos y se obtiene una representación analizable del contenido semántico. De esta forma, se podrán aplicar los modelos anteriores para evaluar las propiedades. Tras evaluar todos los requisitos a partir de estos modelos, se obtendrá el resultado de la verificación.

Otros métodos dentro de este grupo son los llevados a cabo por Fabbrini y Rosemberg. Frabriní presenta una herramienta CASE para evaluar la calidad de los requisitos. Analizando y buscando ciertas palabras o expresiones lingüísticas que lleven a incoherencias en las especificaciones.

Rosemberg propone buscar palabras ambiguas, imprecisas, imperativas o que proponen opcionalidad.

## Técnicas de verificación manual para la especificación de requisitos

Se basan en técnicas que consisten en realizar una lectura exhaustiva de las especificaciones para encontrar incoherencias y ambigüedades que es necesario eliminar.

Con respecto a otras técnicas, varía el número de participantes, mantener el objetivo o no en un tipo específico de defectos o de participantes, el tipo y número de reuniones si las hay y el tipo de formalismo utilizado en la documentación.

Para que este tipo de técnicas sean eficientes, deben cumplir las siguientes propiedades:

- Estar asociada a un tipo de documento en particular y tener en cuenta la notación en que este está redactado.
- Ser adaptable a un proyecto y entorno característico.
- Estar lo suficientemente detallada como para que un lector pueda aplicarla con soltura, es decir, que sea usable.
- Ser específica, de tal forma que el procedimiento establecido se adapte perfectamente al propósito u objetivo particular del lector.
- Estar constituida por varias técnicas si es necesario para cubrir las distintas partes del documento. La unión de todas estas técnicas debe cubrir la verificación del documento completo.
- Estar estudiada empíricamente para determinar si, y cuándo, es más efectiva que otras.

A continuación, se definirán brevemente las principales técnicas de lectura propuestas para detectar defectos en los requisitos.



### Actividades

24. ¿Son mejores las técnicas automáticas que las técnicas manuales de verificación? Razone la respuesta.

### Revisión

Es el proceso o reunión durante la cual se presenta el producto, o conjunto de productos, al cliente, los gestores, los usuarios y otras partes interesadas para su aprobación. Puede incluir revisión de código, de diseño, de cualificación formal, de requisitos y de disponibilidad para pruebas.

### Inspección

Se trata de técnicas de análisis estático que se basan en una evaluación visual de los productos de desarrollo para detectar errores, violaciones de recomendaciones estándar y otros problemas.

### Errors-Abstraction

Esta técnica, descrita en Lanubile et al. (1998), exige la redefinición de los siguientes conceptos:

- **Error:** equivocación en la comprensión de las necesidades de un cliente o usuario.
- **Falta:** manifestación concreta de un error en el software. Un error puede causar varias faltas y varios de ellos pueden producir la misma falta.
- **Fallo:** comportamiento erróneo del software operacional. Un fallo particular puede tener su causa en varias faltas y es posible que algunas faltas nunca produzcan un fallo.
- **Defecto:** término genérico para referirse a cualquiera de los tres anteriores.

La técnica consiste en abstraer las faltas para identificar los errores. Se van identificando faltas que se deben a la misma causa y se agrupan para crear una nueva categoría de error.

Al final, se identificarán nuevas faltas que serán ocurrencias del error anterior si se deben a la misma causa.

Lo que se pretende es identificar una jerarquía de errores en la especificación de requisitos según el grado de abstracción del error. Esto supone que facilitará la identificación de errores.

### 5.3. Proceso de mejora continua

Todo proceso requiere el uso de recursos en una organización. De esta forma, ningún producto puede fabricarse sin un proceso y ningún proceso puede existir sin un producto o servicio.

Una posible definición sería que un proceso es un conjunto de actividades que se coordinan y relacionan entre sí para obtener un resultado a partir de entradas de información y recursos. En las organizaciones que se encargan de producir *software*, la calidad del producto está muy relacionada con los mecanismos internos que componen el proceso. De ahí la necesidad de trazar estrategias que permitan controlar el proceso y mejorarlo para posteriores trabajos.

Muchas organizaciones se han dado cuenta de esto y han comenzado a invertir para reforzar y mejorar los procesos, y así lograr un aumento de la calidad y una disminución del fracaso del desarrollo de *software*. Surge entonces la mejora del proceso *software* como un mecanismo de mejora continua de la calidad, utilizado para elevar la capacidad de los contratistas, para auditar desarrollos de *software* interno y para planificar la estrategia de ingeniería del *software* de la empresa.

Para ello, es primordial comprender el estado actual de las prácticas de gestión y de ingeniería de *software* en la empresa, seleccionar las áreas de mejora que son susceptibles de cambios que produzcan resultados óptimos a medio y largo plazo; centrarse en añadir el valor al negocio, sin pensar en que se puede alcanzar la "utopía del mejor proceso posible" y combinar procesos eficaces con personas con habilidades, motivadas y creativas.

## 6. Herramientas de uso común para el desarrollo de *software* de sistemas

El tiempo de desarrollo de cualquier sistema de *software* depende del uso de las herramientas adecuadas que proporcionen al programador el conjunto de capacidades para desenvolverse con facilidad y flexibilidad con todas las actividades que deben llevarse a cabo en las etapas de programación y despliegue del sistema final.

El tipo de herramientas que utilizará el programador también depende del tipo de *software* que se va a desarrollar. Sin embargo, se puede dar un repaso por aquellas herramientas que se suelen utilizar en la mayoría de los casos cuando se habla de un desarrollo a nivel de sistema.

### 6.1. Editores orientados a lenguajes de programación

Si se quiere empezar a programar, es completamente necesario utilizar una herramienta que permita escribir el código, al mismo tiempo que proporcione facilidad para mantenerlo,

visualizarlo y gestionarlo. Es decir, se necesita un **editor de texto** que esté orientado al uso del lenguaje de programación que se ha elegido para el desarrollo. Esta es, sin duda, la principal herramienta de trabajo de todo programador.

- **Emacs** es un editor de texto famoso en el mundo *Linux*. Creado por GNU y programado en C y Lisp. Funciona como un editor simple, pero permite a los programadores utilizar bibliotecas para compilar y depurar el código.
- **Kate** es un editor de texto para la plataforma *KDE* donde destaca su simplicidad. Programada en C++ y qt, permite coloreado de sintaxis extensible mediante XML, soporte de sesiones y seguimiento de código para C, C++, Java y otros lenguajes.
- **Vim** es una versión mejorada de *Vi* y es el editor de texto más avanzado de *Linux*. Se trata de un editor altamente configurable. A veces, es denominado "editor para programadores". Sin embargo, permite todo tipo de edición de texto, desde escribir un e-mail hasta editar archivos de configuración.

Es bastante complejo de usar y requiere un buen dominio para sacarle partido.



#### Importante

La elección del editor adecuado marcará la diferencia en el nivel de eficiencia y versatilidad que el desarrollador alcanzará en la actividad de codificación del sistema.



#### Actividades

25. Busque en internet alguna herramienta alternativa a los editores que se han nombrado anteriormente y compare sus características.

## 5.2. Compiladores y enlazadores

Se sabe que un compilador traduce las instrucciones de un lenguaje de programación a instrucciones que la máquina, o, mejor dicho, el procesador, pueda interpretar y ejecutar. Cada lenguaje requiere un compilador propio y traduce el programa antes de que pueda ser ejecutado.

Uno de los compiladores más importantes para el lenguaje de programación C es el que forma parte de *GCC (GNU Compiler Collection)*, conjunto de compiladores del proyecto GNU y que son estándares y derivados de sistemas *Unix*. Un ejemplo de uso de este compilador es el siguiente:

1. Imagínese el siguiente trozo de código que se quiere compilar con *GCC*:

```
#include <stdio.h>
int main(){
    int x;

    printf("Escribe el número (sin decimales");
    scanf("%i",&x);

    if(x%2==0) printf("Es par");
    else printf("Es impar");
}
```

Para ello, primero se usa un editor de texto sin formato y se guarda el código anterior en un fichero, por ejemplo llamándolo "main.c", en el disco duro.

2. Después, desde la linea de comandos de *Linux*, se puede usar gcc ejecutando la siguiente sentencia:

```
gcc -c main.c
```

Tras esta sentencia, el compilador habrá generado un fichero objeto que contiene el código en lenguaje máquina que puede ser entendido por el procesador para realizar la tarea que se ha escrito en el editor.

3. Pero aún no acaba el proceso de generación del ejecutable. Hace falta enlazar al fichero objeto todo el código externo que se ha utilizado en el programa. Para ello, se usa la sentencia:

```
gcc -o main main.o
```

Tras esta sentencia, el enlazador habrá resuelto todas las referencias de código y el resultado será un fichero ejecutable del programa.

4. También es posible realizar el proceso de compilación y enlazado de una sola vez, usando la siguiente sentencia:

```
gcc -o main -c main.c
```

Se puede encontrar GCC en la mayoría de los sistemas *Linux*, pero en el entorno de *Unix* existe un compilador estándar que aparece con más frecuencia, el denominado *UNIX Compiler Command (cc)*.



## Sabía que...

La mayoría de los sistemas *Unix* usan cc como compilador principal de C. Su uso es muy parecido a gcc.

El compilador cc usa opciones que son comunes a gcc.

### Opción -c

Esta opción es, probablemente, la más estandarizada a nivel global. Indica que el compilador debe generar un fichero objeto ("fichero.o"), pero sin enlazar para obtener un ejecutable. Se usa en la compilación por separado de múltiples módulos fuentes, para que sean enlazados en el siguiente paso usando el enlazador. Por ejemplo, si se tiene un fichero fuente llamado "mates.c", la compilación con el comando cc y la opción -c:

```
$cc -c mates.c
```

El resultado que se obtiene será un listado de los errores sintácticos del programa. O bien, si no hay errores, se habrá generado el fichero con el código objeto "mates.o". A continuación, se necesitará enlazar y generar el fichero ejecutable:

```
$cc mates.o
```

Pero se pueden dar los dos pasos en uno. Para ello, se usa la sentencia de la siguiente forma:

```
$cc hello.c
```

El resultado de esta compilación, si no hay errores, es el fichero ejecutable "a.out". Se puede cambiar el nombre del fichero en el proceso de compilación, pero, si no se especifica, se generará por defecto el fichero con el nombre "a.out".

Un programa C se puede descomponer en varios módulos, cada uno en un fichero distinto. Para compilar el programa se puede hacer compilando cada uno por separado y finalmente enlazarlos todos a la vez para formar el fichero ejecutable. Por ejemplo: una aplicación que está formada por tres módulos: "nomina1.c", "nomina2.c" y "nomina3.c". La compilación de cada archivo fuente es:

```
$cc -c nomina1.c  
$cc -c nomina2.c  
$cc -c nomina3.c
```

Después, se enlazan los ficheros objetos generados:

```
$cc nomina1.o nomina2.o nomina3.o
```

El resultado vuelve a ser un fichero ejecutable de nombre "a.out". Con la opción -c se han generado los tres ficheros objetos: "nomina1.o", "nomina2.o" y "nomina3.o". Después, la orden cc sobre los tres ficheros objetos produce el archivo ejecutable final.

La opción -o ayuda a cambiar el nombre del fichero de salida generado en la compilación. Por ejemplo, en la sentencia siguiente, el fichero de salida generado será un fichero objeto de nombre "mat\_ecuacion.o":

```
$ cc -c ecuacion.c -o mat_ecuacion.o
```



### Aplicación práctica

Se ha terminado el desarrollo del sistema y se comprueba que se han obtenido tres ficheros con los módulos necesarios implementados. Sin embargo, ahora se quiere obtener el ejecutable de la aplicación. ¿Cómo se debe obtener dicho ejecutable?

#### SOLUCIÓN

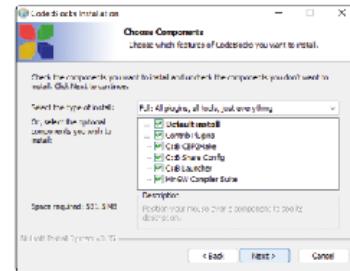
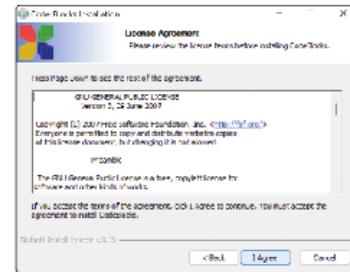
- I. Se abre un terminal en *Linux* o bien un intérprete de comandos en *Windows*.
- II. Se ejecutan las siguientes sentencias:

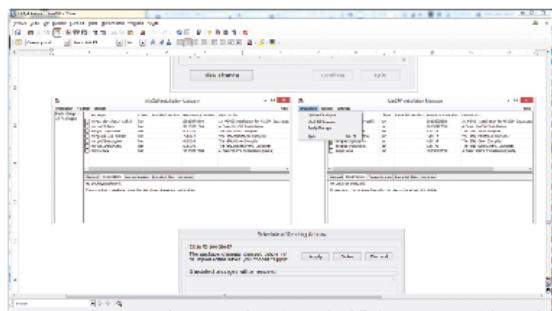
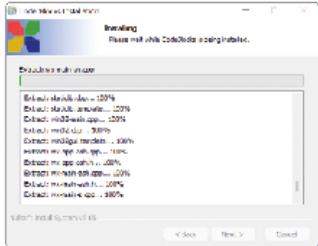
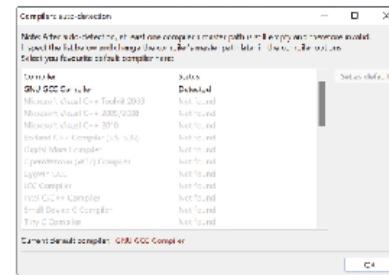
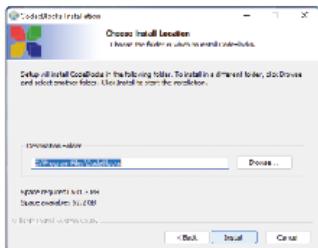
```
gcc -c fichero1.c  
gcc -c fichero2.c  
gcc -c fichero3.c  
gcc -o main fichero1.o fichero2.o fichero3.o
```

**Code::Blocks** es una implementación de los compiladores GCC para los sistemas operativos de Microsoft. Además, el compilador incluye un conjunto de la biblioteca de funciones de Win32 para poder desarrollar aplicaciones nativas.



La descarga de *MinGW* se puede hacer a través de su página <<https://www.codeblocks.org>>. Tras la descarga, se procede a la instalación.



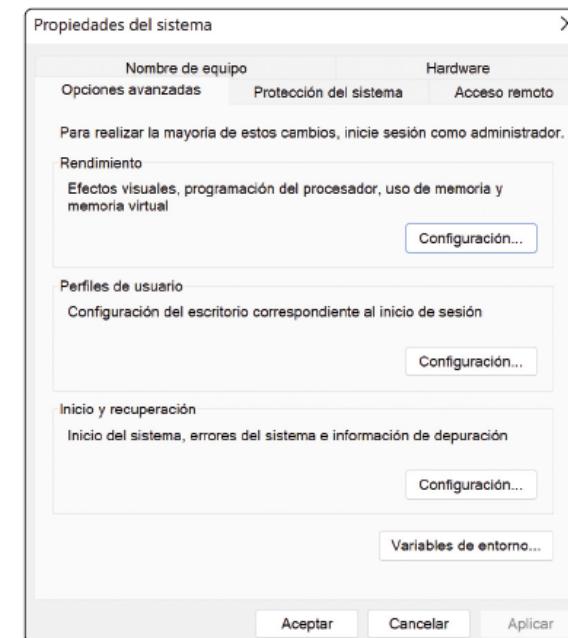


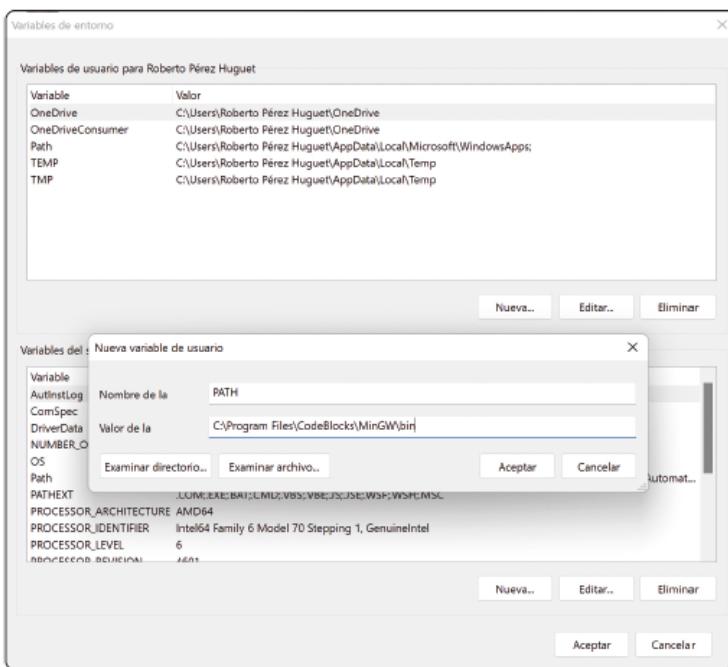
The screenshot shows the Eclipse IDE interface with the 'Java Build Path' dialog open. The 'Libraries' tab is selected, displaying the following entries:

- Project build path
- JavaSE-1.8
- JavaSE-1.8 API
- JavaSE-1.8 Javadoc

Below the tabs, the 'Order and Export' section is visible, showing the order of the libraries: Project build path, JavaSE-1.8, JavaSE-1.8 API, and JavaSE-1.8 Javadoc.

At the bottom of the dialog, there are several buttons: 'OK', 'Cancel', 'Apply', 'Close', and 'Help'. The 'OK' button is highlighted with a blue border.





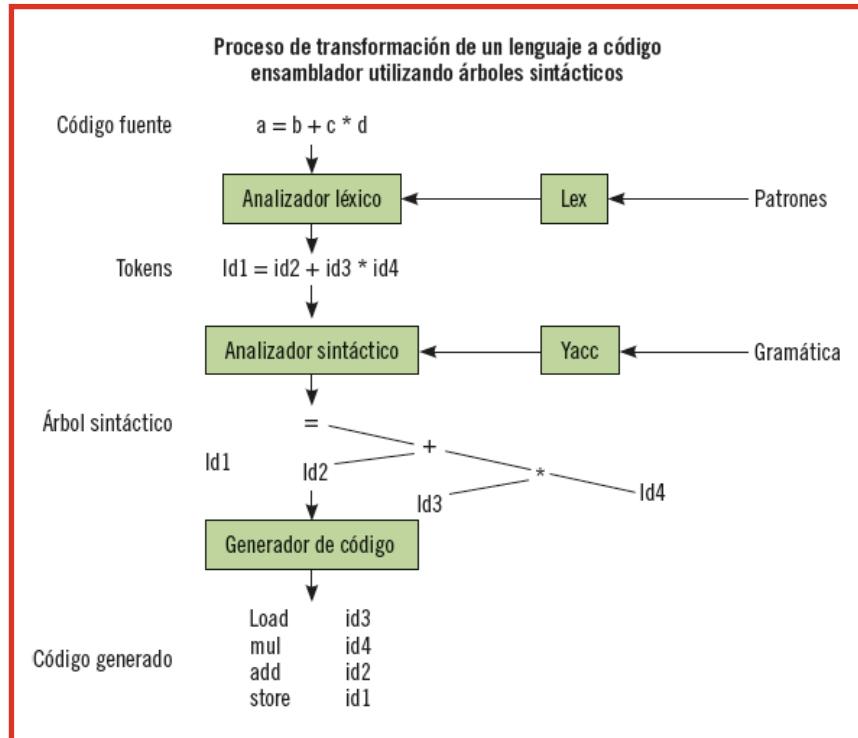
*Modificación de las variables del entorno para facilitar la llamada al compilador*

Tras esto, se podrá usar el compilador *Code::Blocks* de forma similar a como se utiliza *gcc* en *Linux*. Se debe abrir previamente una línea de comandos en *Windows*. Si se tiene un programa en "main.c", se podrá compilar usando las sentencias siguientes:

*Compilación y ejecución de un programa básico*

### 6.3. Generadores de programas

El concepto de generador de programa está relacionado con el de compilación. En este tipo de sistemas, la entrada es una gramática que describe un lenguaje a analizar, siendo la salida el propio analizador del lenguaje. Existen algunas herramientas de bajo nivel que permiten simplificar estos pasos. ***Yacc (Yet another compiler-compiler)*** es un programa que se usa para generar analizadores sintácticos (*parsers*), es decir, programas que convierten la entrada de texto (que representa un trozo de código del lenguaje que se intenta analizar) en estructuras en forma de árbol, que son necesarias para realizar un análisis posterior para determinar si el texto representa un programa válido escrito en ese lenguaje. *Yacc* se usa, con frecuencia, junto con analizadores lexicográficos como *lex* (*scanners* o *lexers*). Un analizador lexicográfico forma parte de la primera fase de un compilador. Su función es recibir como entrada un programa escrito en un lenguaje y dar como resultado un conjunto de *tokens* o símbolos que están contenidos en ese código y que forman el programa. Estos *tokens* se utilizan en etapas posteriores para el análisis sintáctico.



ejecución y que, en la mayoría de los casos, no son detectados por el programa son los peores errores de todos, porque son inesperados y, a veces, son difíciles de encontrar.

En un modelo de desarrollo, los esfuerzos por diagnosticar y corregir estos errores suponen una cantidad importante de recursos en coste y tiempo. Se suelen denominar *bugs* (bichos).

Los programadores poseen diferentes métodos para “depurar” los errores de un programa. Algunos con más eficacia que otros. Un primer método consiste en incluir llamadas a **printf** que escriben en pantalla mensajes para monitorizar aspectos como variables, control de flujo, etc. Es el método menos eficiente, pero el más rápido. El método más recomendable sería aquel que permitiera suspender la ejecución en cualquier línea de código para comprobar el estado de todo el proceso, variables, etc. Este método es posible aplicarlo usando lo que se conoce como depurador o en, su término inglés, *debugger*.



#### Importante

Un depurador se comporta como un envoltorio donde un programa se ejecuta y permite asumir el control del flujo del programa, así como monitorizar todo el estado de su ejecución en cualquier instante.

Entre las características que un buen depurador debe proporcionar, se encuentran las siguientes:

- Ejecución del programa línea a línea.
- Detención de la ejecución en una línea de código concreta.
- Detención de la ejecución cuando se cumplen determinadas condiciones específicas.

## 6.4. Depuradores

Podría parecer que es suficiente con generar el ejecutable del programa para considerar que está libre de errores. Sin embargo, esto no es verdad. Los errores que se producen en

- Visualizar los contenidos de las variables en cualquier momento de la ejecución.
- Poder cambiar el valor del entorno de ejecución para comprobar el efecto sobre el comportamiento del programa.

En *Unix*, *gdb* es el depurador proporcionado por GNU. Es gratuito y se ha convertido en un estándar. Hay que aclarar que el uso de un depurador es totalmente independiente del lenguaje de programación en el que se desarrolle el programa. Aun así, cuando se compila un programa en C, es posible añadir información útil, que facilita una posterior depuración mediante *gdb*.

Para poder depurar el programa en C, se debe haber compilado con la opción `-g`. De lo contrario, tan solo se tendrá información en código máquina para el programa. A continuación, se usa la siguiente sentencia desde la línea de comandos de *Linux*:

```
$ gdb fichero_ejecutable
```

Además, utilizando esta forma de depurar, se puede también invocar una depuración cuando un programa ha abortado su ejecución. Cuando un programa aborta, vuelve la información de su estado en un fichero denominado **core**. De esta forma, se puede usar la información que ha volcado debido a la salida repentina para encontrar el posible error que la provocó. Se usa la sentencia siguiente:

```
$ gdb ejecutable fichero_core
```

Incluso se puede interrumpir un proceso para depurarlo. La siguiente sentencia interrumpe el proceso con PID NNNN y entra en modo depuración. El proceso debe pertenecer al usuario que ejecuta el depurador:

```
$ gdb ejecutable NNNN
```



### Actividades

26. ¿Cuál es la diferencia entre un depurador y un compilador?

### 6.5. De prueba y validación de software

Existen herramientas que permiten programar casos de prueba para los requisitos del sistema y así hacer un seguimiento de dichas pruebas y proporcionar consistencia a la validación del *software*. *Postman* es una de estas herramientas, genérica, muy completa y libre. Permite realizar pruebas de *software* e integrar informes de defectos encontrados, así como el resultado de los casos de prueba.

Postman herramienta para pruebas de software

## 6.6. Optimizadores de código

En la mayoría de las ocasiones, el compilador produce código eficiente, pero no alcanza el punto óptimo de eficiencia. La optimización de *software* es el proceso de modificación de

un *software* para volverlo más eficiente y reducir así los recursos que usa (mayor rendimiento). En general, la optimización provocará que el programa se ejecute más rápido y que use menos memoria u otros recursos.

Este proceso puede ser automatizado por compiladores o realizado de forma manual por los programadores. Un compilador optimizador trata de reducir los atributos de un programa para aumentar la eficiencia y el rendimiento.

Las optimizaciones del compilador son una secuencia de transformaciones que convierten un programa en otro semánticamente equivalente pero optimizado.

Generalmente, hay cuatro grupos de transformaciones:

- Reducir el tiempo de ejecución del programa.
- Reducir la cantidad de espacio en memoria que ocupa el programa en ejecución.
- Reducir el tamaño del programa.
- Minimizar la potencia de cálculo del programa.

## 6.7. Empaquetadores

Los paquetes son la forma de distribuir *software*. Agrupan los diferentes archivos necesarios para que un programa sea instalable y, por lo tanto, funcione. En Windows, el fichero que compone el paquete suele ser un fichero ejecutable que descomprime, instala y ejecuta la aplicación.



Sabía que...

**En Linux, los paquetes no son ejecutables, son gestionados por tercera aplicaciones.**

Los paquetes suelen ser muchísimo más compactos y reducidos porque no contienen las librerías compartidas (dependencias). El gestor de paquetes sería el encargado de avisar de la falta de esas dependencias y dar la posibilidad de su descarga e instalación.

Existen tres formatos o tipos de paquetes en una distribución *Linux*:

- **RPM:** se utilizan en distribuciones basadas en *Red Hat*, *Fedora*, *Mandriva*, etc. Pueden tener tanto binarios como código fuente.
- **DEB:** son distribuciones basadas en *Debian* (*Linex*, *Guadalinex*, etc.). Pueden contener tanto binarios como código fuente.
- **Tar.gz:** código fuente empaquetado y comprimido para ser instalado directamente.

En casi todos los casos, las aplicaciones se proporcionan en varios de estos tipos de paquetes.

El empaquetado RPM se está convirtiendo en un estándar de hecho en el mundo *Linux*, porque tiene ventajas importantes con respecto a los otros tipos de empaquetados. Mantiene una base de datos de los paquetes instalados y de sus archivos y permite realizar consultas y verificaciones del sistema.

Otra ventaja es que, en las actualizaciones, los ficheros de configuración se respetan y no es necesario volver a realizar los ajustes específicos.

En RPM se suele llevar una numeración de versión propia. Por ejemplo, supóngase el programa *yasr* empaquetado en RPM de la siguiente forma: *yasr-0.7.6.i386.rpm*.

Su nombre está compuesto del propio nombre de la aplicación empaquetada (*yasr*), la versión (0.7), el desarrollo (6) y la arquitectura (i386).

La sentencia desde la línea de comandos para instalar este paquete sería:

```
$ rpm -ivh yasr-0.7.6.i386.rpm
```

El empaquetado DEB, al igual que el RPM, permite instalar y desinstalar, buscar dependencias y actualizar el software instalado. Una de las ventajas con respecto a RPM es que suele exigir menos dependencias para la instalación de los programas y es algo más rápido por este motivo.

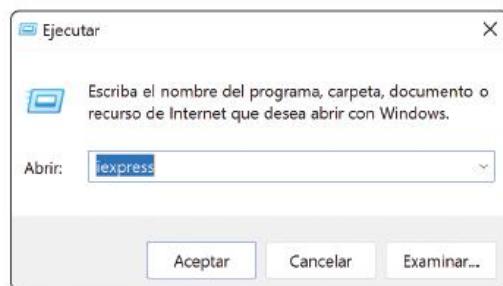
El formato TAR es una forma de agrupar ficheros y directorios en uno solo, de forma que sea más fácil su manipulación. Se puede ver como algo similar al formato ZIP en *Windows*. Su origen fue la creación de copias de seguridad sobre unidades de cinta externa.

Se suele usar con frecuencia en combinación con GZIP (herramienta de compresión) de forma que no solo se agrupa, sino que también se comprime. Los dos procesos se pueden realizar a la misma vez o por separado. Usando la opción z, se indica a tar que se quiere la compresión del fichero.

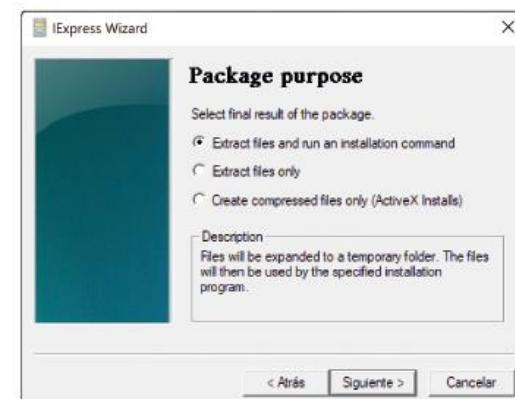
Por ejemplo: tar cvfz fichero /directorioacomprimir.

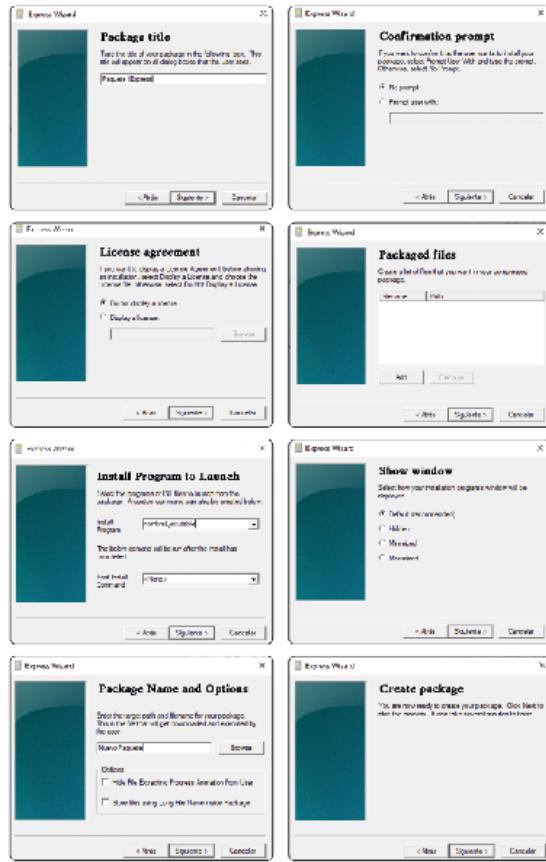
*Windows* también proporciona un sistema de empaquetado propio. Se trata de una herramienta que permite construir un asistente de instalación que empaqueta y facilita la distribución de los programas.

La herramienta se puede invocar desde la línea de comandos de la siguiente forma:



Tras ejecutar *IExpress*, un asistente guiará la realización el empaquetado de la aplicación.





#### Ejecución del asistente de empaquetamiento IExpress de Windows

Tras terminar el asistente, se habrá generado un fichero ejecutable que contendrá todo lo necesario para instalar y desempaquetar el programa en un sistema *Windows*.



#### Actividades

27. Haga un resumen de los principales tipos de empaquetadores de los que se han nombrando sus características principales.



#### Aplicación práctica

Se está ante el despliegue del proyecto en un sistema *Linux*, pero se necesita trasladar el código compilado a cada uno de los sistemas que lo usarán. El jefe de proyecto le pide empaquetar todo el proyecto con algún sistema de empaquetamiento de *Linux*. Los ficheros que componen el proyecto son: "cliente.f" y "servidor.f". Ambos están ubicados en la ruta /usr/root/proyecto. ¿Qué sentencia o sentencias ejecutaría en un terminal *Linux* para realizar el empaquetado?

#### SOLUCIÓN

```
$ tar -cvfz proyecto.tar /usr/root/proyecto
```

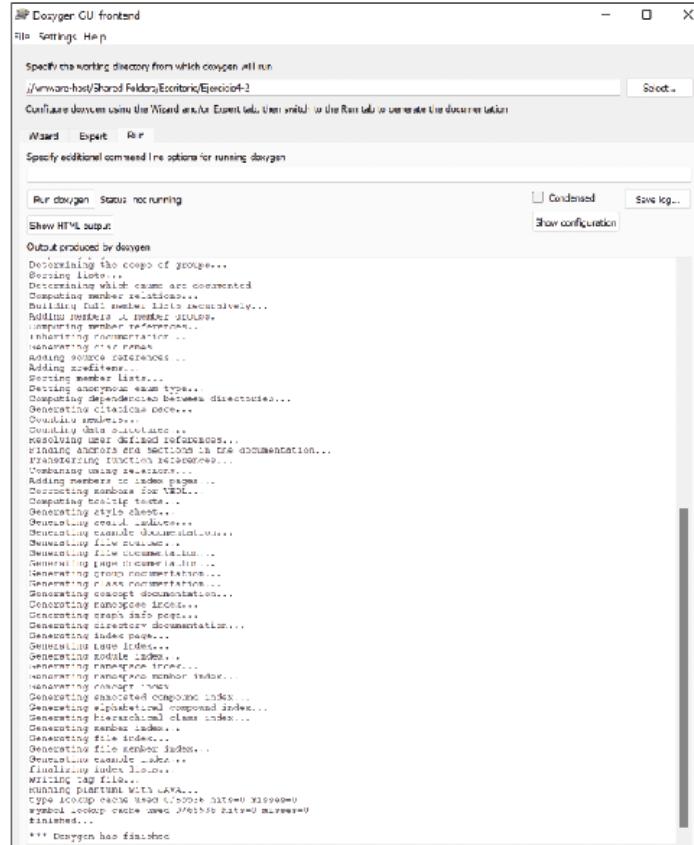
## 6.8. Generadores de documentación de software

El proceso de documentación del *software* es muy importante, desde el punto de vista del usuario final para entender el funcionamiento del programa, y desde el punto de vista del programador como ayuda en el proceso de desarrollo. Un generador de documentación es una herramienta de programación que genera documentación a partir del código fuente, si este se encuentra documentado en la fase de codificación.

Existen muchísimos generadores de documentación, algunos libres y gratuitos y otros de pago.

*Doxygen* es uno de los más famosos debido a que es muy adaptable, es multiplataforma y permite la salida de la documentación en diversos formatos.

Varios proyectos como *KDE* usan *Doxygen* para generar la documentación de su API. *KDevelop* incluye soporte para *Doxygen*.



## 6.9. Despliegue de software

Tras el proceso de desarrollo, el siguiente paso suele ser la instalación del *software* en los equipos. A este proceso se le denomina despliegue del *software*. A veces, el despliegue se realiza sobre un único equipo, mientras que en otras ocasiones se realiza sobre varios equipos al mismo tiempo. En concreto, si se está en el segundo caso, cada vez que una nueva actualización está disponible, el proceso vuelve a comenzar y, si no se usan las herramientas adecuadas, la pérdida de tiempo puede ser importante.

### Gestores y repositorios de paquetes. Versionado y control de dependencias

Un sistema de gestión de paquetes o gestor de paquetes tiene como finalidad permitir la instalación, actualización, configuración y eliminación de *software* de forma automática. Existen sistemas operativos, como es el caso de los sistemas *Linux*, que se apoyan en este tipo de sistemas para mantener todo el *software* instalado.

En estos casos, todo el *software* se distribuye en forma de paquetes.



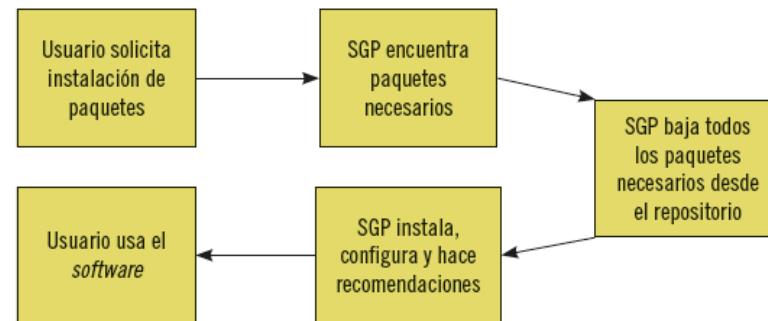
#### Recuerde

Un paquete es una forma de distribuir *software*. Agrupa diferentes archivos necesarios para que un programa sea instalable y, por lo tanto, funcione.

Además de incluir el programa, a veces se incluye información adicional como una descripción de la funcionalidad del *software* que está empaquetado, el número de versión, el distribuidor del *software*, la suma de verificación e incluso la lista de dependencias de otros paquetes. Toda esta información es mantenida por el sistema una vez se haya instalado el programa para el control de versiones.

Un gestor de paquetes lleva a cabo un flujo de acciones que se puede ilustrar de la siguiente forma:

Diagrama de flujo de información típico del sistema de gestión de paquetes



Windows mantiene una filosofía distinta, acostumbra a distribuir el *software* junto con sus propios instaladores. Usa para ello una única base de datos para la instalación, guardando información de dicha instalación en el registro del sistema operativo. En *Linux*, la ventaja principal es que la gestión es centralizada.

Un buen sistema de gestión de paquetes debe organizar todos los paquetes instalados en el sistema y mantener su usabilidad. Para ello, se deben realizar comprobaciones para evitar que haya diferencias entre la versión local de un paquete y la versión oficial y comprobaciones de firma digital, para asegurar la legitimidad del *software*.

Otras funciones a destacar del gestor son la instalación, actualización y eliminación simple de paquetes y la resolución de dependencias para garantizar que el *software* funcione correctamente.

*Apt-get* y *Yum* son las herramientas de gestión de paquetes para DEB y RPM respectivamente. Son muy potentes y fáciles de manejar. Permiten desinstalar y manejar todo lo referente a paquetes desde la línea de comandos. Lo mejor de todo es que tienen configurados repositorios en la web donde buscar los programas que se intentan instalar, de forma que simplemente con una sentencia somos capaces de descargar e instalar cualquier programa. Si el paquete que se está instalando necesita algún otro para su funcionamiento, el gestor avisará descargará e instalará todos ellos si se le indica.

### Distribución de software

La distribución de *software*, también conocida como *software distro*, es un conjunto de *software* o colección que se encuentra compilada y configurada.

El *software* es el producto final de un trabajo de una o varias personas, por lo que su uso y distribución deben estar regulados. Para ello, el producto final de *software* debe incluir un documento que determine el tipo y las propiedades exigibles a la hora de su distribución. Existen diferentes tipos de documentos que permiten distribuir el *software* de múltiples formas posibles. Cada documento de este tipo se denomina licencia. En el mundo *Linux*, la

licencia más utilizada para la distribución es la licencia GPL u *open source*, pero existen otras muchas. Se van a describir algunas de las más utilizadas.

### Licencias OEM (Original Equipment Manufacturer)

Las licencias OEM son adquiridas en la compra de un equipo informático. El *software* bajo esta licencia solo puede ser utilizado y reinstalado en el equipo donde fue preinstalado. Por lo general, se trata de *software* en versión económica y es una forma de distribución dirigida a fabricantes o ensambladores de equipos informáticos.

### Licencias FPP o Retail

Se trata de la licencia que permite al comprador ceder o vender el *software*, a menos que la licencia propia del producto indique lo contrario. Es el método de distribución que se aplica cuando se compra el producto en la tienda.

### Licencias académicas

Las Licencias académicas permiten beneficiar a instituciones educativas para el uso del *software* a precios competitivos. Pueden encontrarse en forma OEM o Retail.

## Licencias por volumen

Las licencias de *software* están destinadas a empresas donde el número de usuarios suele ser grande. Se basan en licencias únicas que pueden ser utilizadas por diferentes equipos. Normalmente, se venden en paquetes de un número de licencias determinado.

## Licencias de software libre

Las licencias de *software* libre se basan en cuatro propiedades clave:

- La libertad de usar el programa, con cualquier propósito.
- La libertad de estudiar el funcionamiento del programa y adaptarlo a las necesidades.
- La libertad de distribuir copias, con lo que se puede ayudar a otros.
- La libertad de mejorar el programa y hacer públicas las mejoras, de modo que toda la comunidad se beneficie.

Este tipo de licencia no implica que el programa deba ser gratuito, pero sí que es necesario y obligatorio distribuir el código del programa junto con su versión ejecutable.

El *software* libre se distribuye mediante diversas licencias entre las que se pueden encontrar las siguientes:

### Licencia GPL

La licencia GPL (Licencia Pública General GNU o simplemente GNU) indica que el autor conserva los derechos (*copyright*), pero se permite la redistribución y modifi-

cación, controlando que todas las versiones modificadas del *software* permanecen bajo los términos más restrictivos de la propia licencia GNU.

### Licencia BSD

La Licencia BSD sí permite utilizar código fuente en *software* que no es libre. Es compatible con las licencias GPL.

El autor puede mantener la protección de *copyright*, pero permite la libre redistribución y modificación. El usuario tiene libertad ilimitada e incluso puede decidir redistribuirlo como no libre.

### Licencia MPL (Mozilla Public License)

Es parecida a la licencia BSD, pero menos permisiva, aunque sin llegar a los extremos de las licencias GPL. Cumple las libertades propias del *software* libre, pero permite que el *software* sea reutilizado, aunque no libremente, por el usuario que lo desee. Es decir, se puede volver a utilizar el *software* sin necesidad de volver a licenciarlo bajo la misma licencia.

### Licencias Copyleft

*Copyleft* fue creado como alternativa a *copyright* (derechos reservados de copia). En *copyleft*, el autor permite la redistribución y modificación del *software*, y añadir restricciones adicionales.



## Actividades

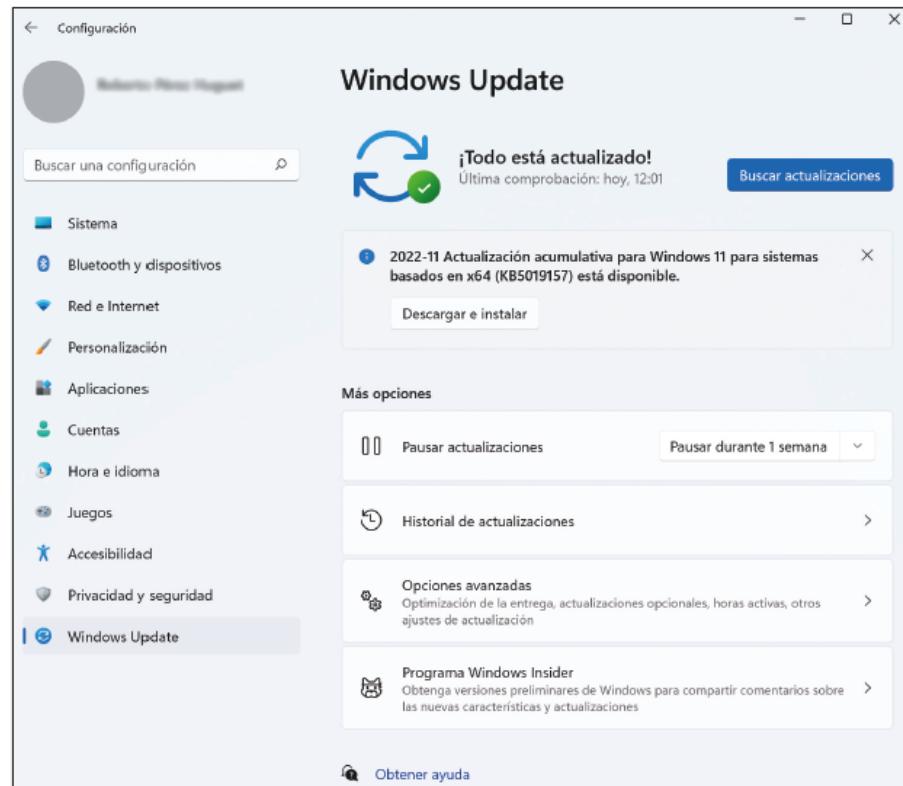
28. ¿Es posible utilizar una licencia Copyleft junto a una licencia GPL? Razona la respuesta.

### Gestores de actualización de software

En los sistemas operativos actuales, existen utilidades que permiten la actualización del software del sistema. Utilizar estos gestores cada cierto tiempo puede reducir drásticamente los problemas de seguridad en entornos donde el acceso a internet es habitual.

En *Linux*, son los gestores de paquetes los que se encargan de la actualización de los componentes del sistema, como ya se ha mencionado.

En *Windows*, la herramienta *Windows Update* permite descargar e instalar las últimas actualizaciones que Microsoft va subiendo a la web.



## 6.10. Control de versiones

En los proyectos de desarrollo de *software* abordados por equipos de programadores, es muy común y necesario el uso de algún sistema de control de versiones y control de cambios en los ficheros que componen el código del proyecto. Teniendo en cuenta que una versión, revisión o edición de un producto, es el estado en el que se encuentra el mismo en un momento dado de su desarrollo o modificación, resulta imprescindible que exista una sincronización entre las versiones y los desarrolladores que crean estas versiones para que el código siga siendo coherente.

En el desarrollo de *software*, se encuentran dos grandes grupos de herramientas de control de versión: herramientas basadas en una estructura cliente/servidor y herramientas basadas en una estructura distribuida.

Entre las del primer grupo, destacan *CVS* y *Subversion* por ser libres y de código abierto.

*CVS* es un sistema de control de versiones centralizado, donde un servidor guarda las versiones actuales del proyecto y su historial. Los desarrolladores se conectan al servidor para obtener una copia completa del proyecto, trabajan con esa copia y más tarde suben los cambios de nuevo al servidor. Por lo general, el desarrollador que usa el *software* cliente se conecta al servidor utilizando internet, aunque tanto cliente como servidor pueden residir en la misma máquina. El sistema *CVS* es capaz de mantener el registro del historial de versiones del proyecto y todos los cambios producidos.

Los desarrolladores pueden obtener copias del proyecto al mismo tiempo. Posteriormente, cuando se actualizan sus modificaciones, el servidor intenta acoplar las

diferentes versiones. Si se produce algún conflicto, debido a que dos clientes han modificado la misma línea en un archivo en particular, entonces el servidor deniega la actualización e informa al cliente. Tras esto, el desarrollador debe resolver manualmente el conflicto.

Cuando no se producen conflictos, la copia sube al servidor como una nueva versión del *software*. Por lo tanto, los números de versión de todos los archivos implicados se incrementan automáticamente y el servidor *CVS* almacena información sobre la actualización: descripción suministrada por el usuario, fecha y nombre del autor, así como sus archivos de registro (*log*).

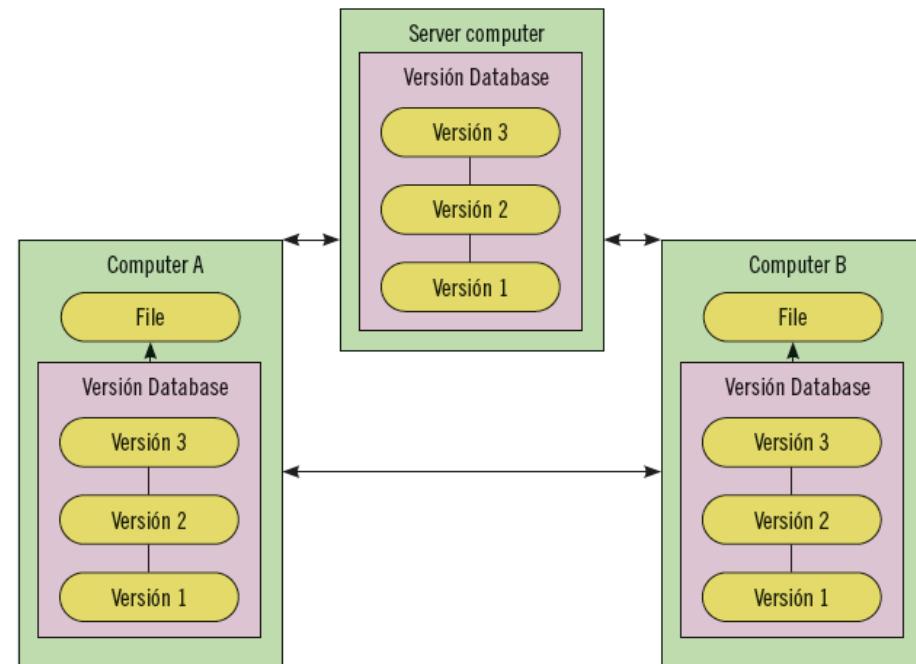
*Subversion* es la evolución natural de *CVS* e intenta mejorar algunas de las carencias de este último. Se ve en esta comparación:

CVS	Subversion (SVN)
Licencia gratuita.	Licencia gratuita.
Multiplataforma.	Multiplataforma.
Orientado a ficheros. Cada modificación de un fichero hace variar la versión del ese fichero.	Orientado a proyectos. Cualquier modificación de fichero registra una nueva versión del proyecto.
Se suben al repositorio ficheros completos, no solo los cambios.	Solo se suben los cambios que se producen por la modificación del fichero.

Soporte Unicode limitado. Puede traer problemas con caracteres "raros".	Soporte Unicode.
Renombrar/eliminar. Hay que hacerlo manualmente: copiar con otro nombre y borrar el antiguo.	Renombrar/eliminar. Internamente realiza las dos cosas mediante una sola transacción de forma transparente para el usuario.

En un DVCS, o control de versiones distribuido (como *Git* o *Mercurial*), los clientes al descargar el proyecto hacen una copia de seguridad o réplica exacta del repositorio. De esta forma, si un servidor cae, se puede recuperar y restaurar el proyecto completamente. Esta es la principal ventaja que existe entre los sistemas de control de versiones distribuidos con respecto a los no distribuidos.

Diagrama de un sistema de control de versiones distribuido



Incluso, aunque existan varios repositorios con los que trabajar, el sistema permite la colaboración con distintos grupos de gente simultáneamente dentro del mismo proyecto. Esto permite establecer varios flujos de trabajo, algo impensable para sistemas centralizados.

## 6.11. Entornos integrados de desarrollo (IDE) de uso común

Un entorno de desarrollo integrado es un conjunto de herramientas que trabajan de forma cooperativa para mejorar y facilitar el trabajo de codificación cuando se desarrolla un proyecto informático. La mayoría de los IDE consisten en un editor de código, un compilador, un depurador y un constructor de interfaz gráfica (GUI).

Los IDE proporcionan el marco de trabajo apropiado para que el programador se sienta cómodo a la hora de llevar a cabo la fase de implementación. Existen multitud de IDE para casi cualquier lenguaje de programación existente, aunque, por razones obvias, los más conocidos son aquellos que permiten programar en C++, PHP, Python, Java, C#, Delphi, Visual Basic, etc.

Los IDE pueden funcionar mediante la gestión de ficheros de texto o bien de forma interactiva en tiempo de ejecución, permitiendo una integración total entre el entorno y el propio proyecto. A veces, la funcionalidad de los IDE puede ser extendida mediante *plugins* o complementos que suministra el propio desarrollador o bien otros desarrolladores.

Las características más destacadas y comunes que se pueden encontrar entre los múltiples IDE más reconocidos y famosos que existen pueden ser las siguientes:

- Multiplataforma.
- Soporte para diversos lenguajes de programación.
- Integración con Sistemas de Control de Versiones.
- Reconocimiento de sintaxis.
- Extensiones y componentes para el IDE.
- Integración con frameworks populares.
- Depurador.
- Importar y exportar proyectos.

- Múltiples idiomas.
- Manual de usuario y ayuda.



### Actividades

29. Busque información sobre lo que quiere decir que un IDE sea capaz de "reconocer la sintaxis".

### Específicos de sistemas Windows

Entre los específicos para una plataforma basada en *Windows* y teniendo en cuenta que se buscan IDE basados en el lenguaje C o C++, se encuentran las siguientes opciones:

#### BloodShed Dev-C++

Es un IDE para C/C++ que usa *Mingw* como compilador, aunque también se puede combinar con *Cygwin* o cualquier otro compilador basado en *GCC*. Entre algunas de sus características, destacan:

- Soporte para compiladores basados en *GCC*.
- Depuración integrada (usando *gdb*).
- Gestor de proyectos.
- Permite resaltar la sintaxis.
- Explorador de clases.

- Autocompletado de código.
  - Listado de funciones.
  - Soporte de perfiles.
  - Creación de aplicaciones Windows, en consola, librerías estáticas y dinámicas (dlls).
  - Soporte de plantillas.
  - Creación de ficheros *makefile*.
  - Edición y compilación de ficheros de recursos.
  - Gestor de herramientas.
  - Soporte de impresión.
  - Facilidades de búsqueda y reemplazo de código.
  - Soporte para CVS.

## *Captura de pantalla de Dev-C++*

## **Microsoft Visual Studio 2022**

Quizás uno de los mejores IDE para C y C++. Proporcionado por Microsoft para su línea de desarrollo basado en .NET. Entre sus múltiples características, destacan:

- Alta personalización.
- Herramientas de optimización de código.
- Depurador integrado.
- Control de versiones propio.
- Uso de extensiones.
- El editor de código admite lenguajes como C#, VB.NET, HTML, JavaScript, XAML, SQL, etc.
- Resaltado de sintaxis y autocompletado de código.

The screenshot shows the Microsoft Visual Studio 2022 interface. The main window displays a C++ file named 'main.c' with the following code:

```
#include <stdio.h>
int main () {
    printf("Hola mundo");
    return 0;
}
```

The 'PROBLEMAS' (Problems) pane at the bottom shows two errors:

- Se han detectado errores de #include. Actualice el valor de includePath. El subray... C/C++(1696) [Lín. 1, col. 1]
- no se puede abrir el archivo origen (código de error "stdio.h"). C/C++(1696) [Lín. 1, col. 1]

The 'PUNTOS DE INTERRUPCIÓN' (Breakpoints) pane at the bottom shows a single breakpoint set on line 1 of the code.

*Captura de pantalla de Microsoft Visual Studio*

## Borland C++

*Borland C++* fue uno de los primeros IDE que existieron para C y C++. Actualmente, es propiedad de la empresa Embarcadero. Combina la biblioteca *Visual Component Library* y un IDE escrito en Delphi. Otras características que lo distinguen son:

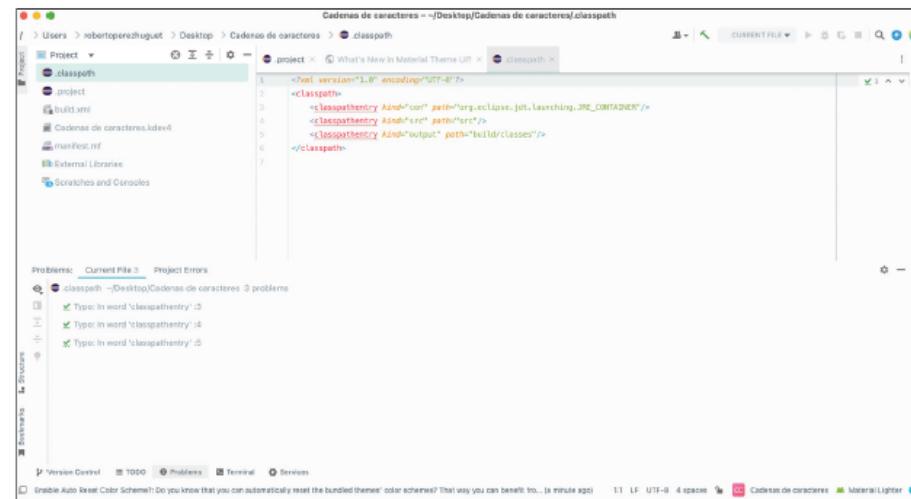
- Permite crear aplicaciones *Windows, Mac y Mobile*.
- Desarrollar aplicaciones para *Android e iOS* en C++.
- Desarrollar aplicaciones 64-bit para *Windows*.
- Acceso a bases de datos con *FireDAC*.

## Específicos de sistemas Unix

### IntelliJ IDEA

*IntelliJ IDEA* permite el desarrollo de aplicaciones en Java, Kotlin, Scala o Groovy, además de incorporar otros como Python o Ruby.

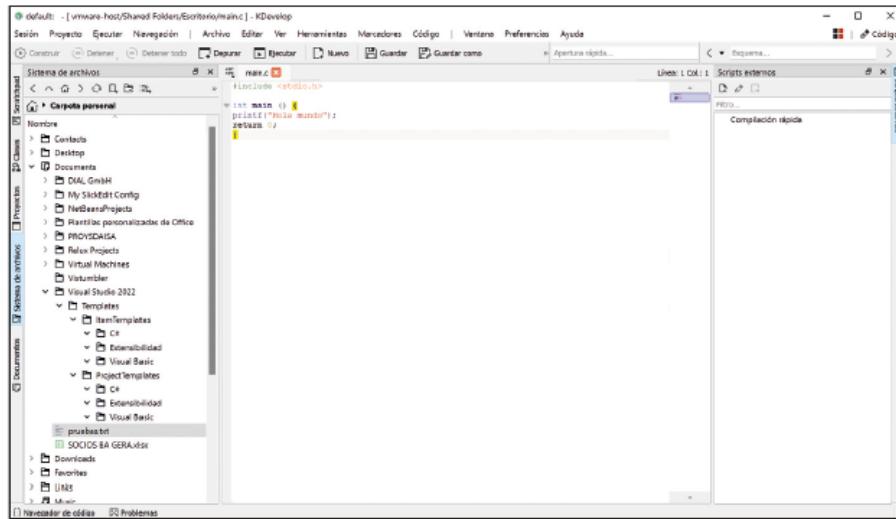
Permite realizar las tareas rutinarias y repetitivas como análisis del código, refactorizaciones, de forma que mientras se desarrolla el IDE analiza el código marcando el que sea incorrecto o no cumpla alguno de los estándares.



Interface de IntelliJ IDEA

### KDevelop

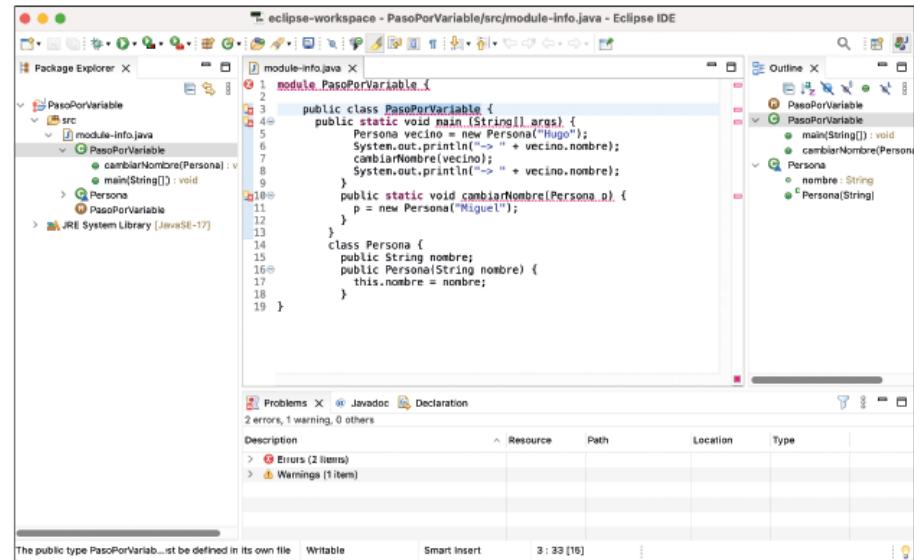
*KDevelop* no cuenta con un compilador propio, por lo que depende de gcc para producir código binario.



Captura de pantalla de KDevelop

## Eclipse

Eclipse es un IDE que va un peldaño más allá que el resto y mantiene una comunidad de usuarios bastante amplia. Fue desarrollado originariamente por IBM. Es libre y de código abierto.



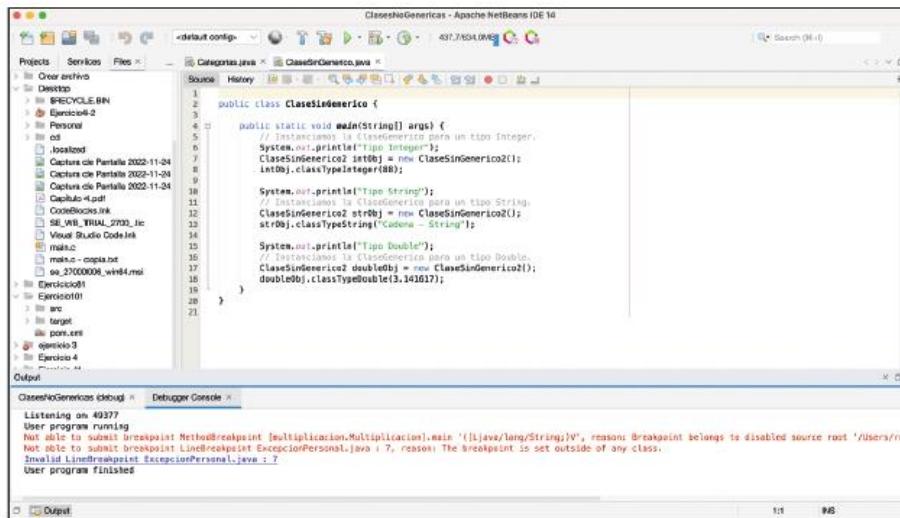
Captura de pantalla de Eclipse

## Multiplataforma

Otros muchos IDE, debido a su popularidad, se encuentran en versiones para diferentes plataformas. Es el caso de los conocidos *Eclipse* y *Netbeans*.

## NetBeans

NetBeans es un IDE muy famoso, cuyo origen es el desarrollo de aplicaciones para Java. Sin embargo, multitud de extensiones han permitido que se convierta en un IDE para otros lenguajes como C/C++. Es código abierto y tiene una gran comunidad detrás.



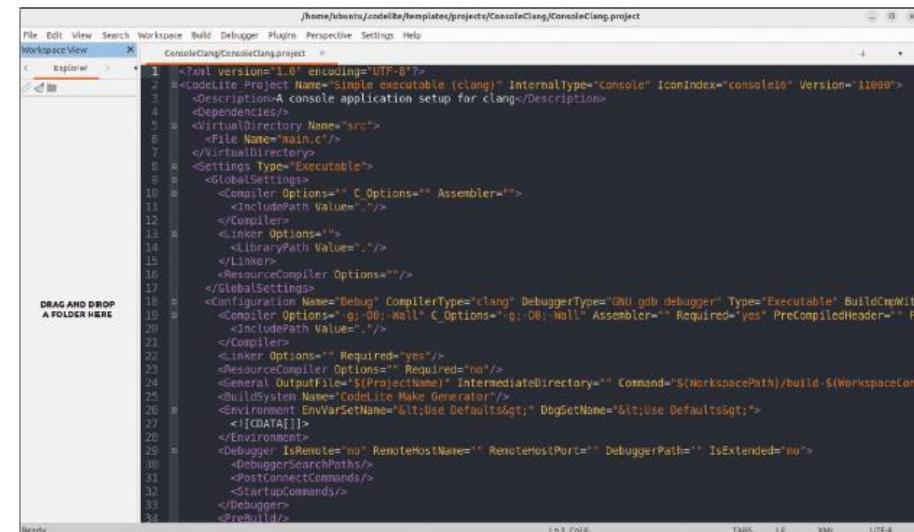
```
public class ClaseIndemericoo {
    public static void main(String[] args) {
        // Instanciar la ClaseGenerica para un tipo String.
        ClaseSinGenerico2 strObj = new ClaseSinGenerico2();
        strObj.classTypeString("Cadena - String");
        System.out.println("Tipo String");

        // Instanciar la ClaseGenerica para un tipo Double.
        ClaseSinGenerico2 doubleObj = new ClaseSinGenerico2();
        doubleObj.classTypeDouble(3.141617);
    }
}
```

Captura de pantalla de NetBeans

## Code::Blocks y CodeLite

CodeLite y Code::Blocks son IDE de código abierto exclusivo para programar en lenguaje C/C++. Usan wxWidgets para la interfaz y se basan en el uso de herramientas libres (MinGW y GDB).



```
<?xml version="1.0" encoding="UTF-8"?>
<CodeLite Project Name="simple_executable (clang)" InternalType="Console" IconIndex="console10" Version="11099">
<Dependencies/>
<VirtualDirectory Name="src">
<File Name="main.c"/>
</VirtualDirectory>
<Settings Type="Executable">
<GlobalSettings>
<Compiler Options="" C_Options="" Assembler="">
<IncludePath Value="" />
</Compiler>
<Linker Options="">
<LibraryPath Value="" />
</Linker>
<ResourceCompiler Options="" />
</GlobalSettings>
<Configuration Name="Debug" CompilerType="Clang" DebuggerType="GNU_gdb_debugger" Type="Executable" BuildCmakeIn="0" IncludePath Value="" />
<Compiler Options="-gj -O0 -Wall" C_Options="-gj -O0 -Wall" Assembler="" Required="Yes" PreCompiledHeader="" PC=""/>
<Linker Options="" Required="yes" />
<ResourceCompiler Options="" Required="no" />
<General OutputFile="$ProjectName" IntermediateDirectory="" Command="$WorkspacePath/build-$WorkspaceConf" BuildSystem Name="CodeLite Make Generator" />
<Environment EnvVarSetName="Git;Use Defaults$gt;" DlgSetName="Git;Use Defaults$gt;">
<![CDATA[]]>
</Environment>
<Debugger IsRemote="no" RemoteHostName="" RemoteHostPort="" DebuggerPath="" IsExtended="no" />
<DebuggerSearchPaths/>
<PostConnectCommands/>
<StartupCommands/>
</Debugger>
<PreBuild/>

```

Captura de pantalla de CodeLite

The screenshot shows the Code::Blocks IDE interface. The main window displays a wxWidgets plugin script named 'menu\_test.plugin.script'. The code defines a constructor for a plugin and a function 'GetMenu' that adds six menu items to a plugin menu. The script uses the wxWidgets API to print messages to the console. The status bar at the bottom indicates the file path 'C:\Program Files\CodeBlocks\share\CodeBlocks\scripts\...\\_Squirrel' and the build configuration 'Windows (CR+LF) WINDOWS-1252 Line 1, Col 0, Pos 0'. A 'Logs & others' tab is open, showing various build logs and messages.

```
menu_test.plugin.script - Code::Blocks 2003
File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plugins DocBlocks Settings Help
Management Projects Files FSymbols Workspace Console application
menu_test.plugin.script
16 | // mandatory to setup the plugin's info
17 | constructor()
18 |
19 | {
20 |     info = PluginInfo();
21 |     info.name = _T("Menu_test_plugin");
22 |     info.title = _T("test menu in the scripting engine of C::B");
23 |     info.version = _T("0.1a");
24 |     info.license = _T("GPL");
25 |
26 |
27 | // optional to create menubar items
28 | Function GetMenu()
29 | {
30 |     local entries = :wxArrayString();
31 |     entries.Add(_T("Plugin/Test/Menu Test"), 1);
32 |     entries.Add(_T("Plugin/Test/~Menu Test2"), 1);
33 |     entries.Add(_T("Plugin/Test/Menu Test3"), 1);
34 |     entries.Add(_T("Plugin/Test/Menu/Test0"), 1);
35 |     entries.Add(_T("Plugin/Test/Menu/Test0"), 1);
36 |     entries.Add(_T("Menu Test Plugin/Test1"), 1);
37 |
38 |     :print("#####\n");
39 |     :print("Welcome to the Menu_test_plugin\n");
40 |     :print("There should have been created 6 menu:\n");
}
Logs & others
Code::Blocks Search results Cocc Build log Build messages CppCheck/Vera++ CppCheck/Vera++ messages
C:\Program Files\CodeBlocks\share\CodeBlocks\scripts\...\_Squirrel Windows (CR+LF) WINDOWS-1252 Line 1, Col 0, Pos 0 Insert Read/Write default
```

Captura de pantalla de Code::Blocks

## 7. Resumen

El modelo de desarrollo del *software* de sistemas no se aleja mucho del de otros tipos de *software*. No se debe olvidar que el *software* de sistemas más famoso de la historia fue escrito y desarrollado por un estudiante universitario que todavía no había acabado sus estudios y

que, en ese momento, los modelos de desarrollo no eran lo suficientemente maduros para acometer este tipo de proyectos. Sin embargo, las técnicas computacionales incluidas en su desarrollo están fuera de toda discusión.

Aun así, es necesario seguir un proceso de desarrollo, ya que permite otorgar rigurosidad y seriedad al producto final, que es algo que los clientes van a exigir continuamente. Se ha visto, entre muchos conceptos y técnicas, que la fase de análisis es muy importante para marcar un camino en el desarrollo y sentar las bases del producto final. Por otro lado, nada sería del análisis sin los conocimientos y técnicas necesarias para materializar todas las necesidades de los clientes y usuarios potenciales del sistema.

Estos conocimientos, técnicos y avanzados, son la clave para dar forma a las ideas de los clientes y, al mismo tiempo, son un espejo donde mirar para resolver muchos problemas que se presentan de forma natural en los desarrollos.

## Bibliografía

### Monografías

- | PRESSMAN, R. S.: *Software engineering: a practitioner's approach*. McGraw-Hill Higher Education: 2014, 8<sup>th</sup> edition.
- | STALLINGS, W.: *Operating Systems. Internals and Design Principles*. Pearson Education, 2014.
- | SOMMERVILLE, I.: *Software engineering*. Pearson Education: 2021, 10<sup>th</sup> edition.
- | WINTERS,T., MANSHERECK, T., WRIGHT, H.: *Software engineering at Google*. California: Editorial O'Reilly Media, 2020.

### Textos electrónicos, bases de datos y programas informáticos

- | SANTOS, J. M., QUESADA, A., SANTANA, F., ESTEBAN, B.: Sistemas Operativos. Gestión de memoria (1998-2012), de: <<http://sopa.dis.ulpgc.es/fso/teoria/pdf/so-04-1-Memoria.pdf>>.
- | Gestión de memoria, de: <[https://w3.ual.es/~acorral/DSO/Tema\\_3.pdf](https://w3.ual.es/~acorral/DSO/Tema_3.pdf)>.
- | Gestión de procesos, de: <[https://w3.ual.es/~acorral/DSO/Tema\\_2.pdf](https://w3.ual.es/~acorral/DSO/Tema_2.pdf)>.
- | Desarrollo de software basado en componentes, de: <<http://webdelprofesor.ula.ve/ingenieria/jonas/Productos/Publicaciones/Congresos/CAC03%20Desarrollo%20de%20componentes.pdf>>.

- | Especificación de requisitos software según el estándar de IEEE 830, de: <<https://www.fdi.ucm.es/profesor/gmendez/docs/is0809/ieee830.pdf>>.
- | Pruebas de software, de: <[https://oa.upm.es/40012/1/PFC\\_JOSE\\_MANUEL\\_SANCHEZ\\_PENO\\_3.pdf](https://oa.upm.es/40012/1/PFC_JOSE_MANUEL_SANCHEZ_PENO_3.pdf)>.
- | Evaluación de metodologías para la validación de requerimientos, de: <[http://sedici.unlp.edu.ar/bitstream/handle/10915/130429/Documento\\_completo.pdf?sequence=1](http://sedici.unlp.edu.ar/bitstream/handle/10915/130429/Documento_completo.pdf?sequence=1)>.
- | La optimización. Una mejora en la ejecución de programas, de: <<http://ditec.um.es/~jmgarcia/papers/ensayos.pdf>>.
- | Una aproximación empírica a la verificación de especificaciones de requisitos para sistemas de información, de: <[http://www.lsi.us.es/docs/doctorado/memorias/Memoria\\_ProyectoInvestigador.pdf](http://www.lsi.us.es/docs/doctorado/memorias/Memoria_ProyectoInvestigador.pdf)>.
- | Calidad de componentes software, de: <<http://www.javier8a.com/itc/bd1/Norma%20EEE%201061.pdf>>.