

# ENTORNOS DE DESARROLLO

DESARROLLO DE APLICACIONES WEB

Ilerna

**ILERNA**

© Ilerna Online S.L., 2021

Maquetado e impreso por Ilerna Online S.L.

© Imágenes: Shutterstock

Impreso en España - Printed in Spain

Reservado todos los derechos. No se permite la reproducción total o parcial de esta obra, ni su incorporación a un sistema informático, ni su transmisión en cualquier forma o por cualquier medio (electrónico, mecánico, fotocopia, grabación u otros) sin autorización previa y por escrito de los titulares del copyright. La infracción de dichos derechos puede constituir un delito contra la propiedad intelectual.

**Ilerna Online S.L.** ha puesto todos los recursos necesarios para reconocer los derechos de terceros en esta obra y se excusa con antelación por posibles errores u omisiones y queda a disposición de corregirlos en posteriores ediciones.

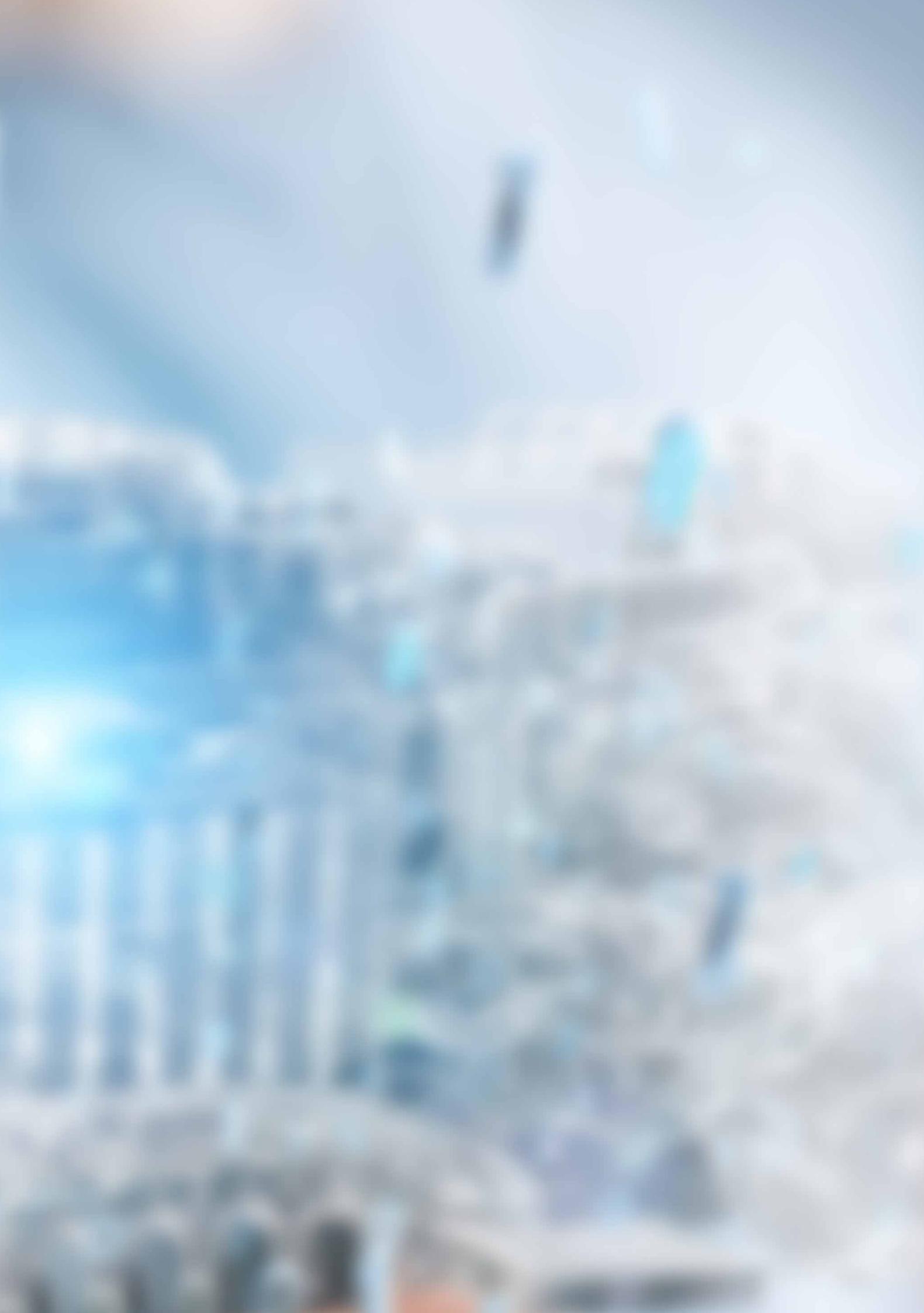
1.<sup>a</sup> edición: enero 2021

# ÍNDICE

## Entornos de desarrollo

<b>1. Desarrollo de software.....</b>	<b>6</b>
1.1. El software del ordenador .....	7
1.2. Concepto de programa informático.....	11
1.3. Código fuente, código objeto y código ejecutable. Máquinas virtuales.....	14
1.4. Tipos de lenguajes de programación. Clasificación y características de los lenguajes más difundidos.....	18
1.5. Fases del desarrollo de una aplicación: análisis, diseño, codificación, pruebas, documentación, mantenimiento y explotación.....	25
1.6. Metodologías ágiles .....	61
1.7. Proceso de obtención de código a partir del código fuente. Herramientas implicadas.....	67
<b>2. Instalación y uso de entornos de desarrollo.....</b>	<b>68</b>
2.1. Funciones de un entorno de desarrollo .....	69
2.2. Instalación de un entorno de desarrollo.....	70
<b>3. Diseño y realización de pruebas .....</b>	<b>80</b>
3.1. Planificación de pruebas .....	81
3.2. Estrategias de pruebas de software. Automatización de pruebas .....	83
3.3. Pruebas de código: cobertura, valores límite y clases de equivalencia .....	91
3.4. Pruebas funcionales, pruebas estructurales y pruebas de regresión.....	103
3.5. Herramientas de depuración de código .....	103
3.6. Calidad del software .....	104
<b>4. Documentación y optimización .....</b>	<b>108</b>
4.1. Refactorización. Concepto. Limitaciones. Patrones de refacción más usuales.....	109
4.2. Control de versiones. Estructura de las herramientas de control de versiones .....	117
4.3. Documentación. Uso de comentarios. Alternativas.....	125
<b>5. Introducción al UML .....</b>	<b>128</b>
<b>6. Elaboración de diagramas de clases .....</b>	<b>132</b>

6.1.	Objetos .....	133
6.2.	Diagramas de clases .....	134
6.3.	Herramientas para el diseño de diagramas.....	141
<b>7.</b>	<b>Elaboración de diagramas de comportamiento.....</b>	<b>148</b>
7.1.	Tipo. Campo de aplicación .....	149
7.2.	Diagramas de casos de uso. Actores, escenario, relación de comunicación .....	150
7.3.	Diagramas de interacción.....	156
7.4.	Diagramas de estados. Estados, eventos, señales, transiciones .....	165
7.5.	Diagramas de actividades. Actividades, transiciones, decisiones y combinaciones .....	167
	<b>Bibliografía / webgrafía .....</b>	<b>171</b>
	<b>Solucionario .....</b>	<b>172</b>



```
onFail: failcase
});
},
changeGroupState: function(state, btn) {
var doChangeState = function() {
if (cur.changingGroupState) return;
function participate(yes) {
if (yes) {
val('members_count', cur.count);
replaceClass(btn, 'colorize');
val(btn, cur.unsubscribe, yes);
setStyle('andis_row', 'dark');
} else {
val('members_count', cur.count);
replaceClass(btn, 'colorize');
val(btn, cur.unsubscribe, yes);
setStyle('andis_row', 'light');
}
}
cur.changingGroupState = doChangeState;
if (true) {
var group = document.querySelector('.group');
var count = document.querySelector('.count');
var row = document.querySelector('.andis_row');
var button = document.querySelector('.button');
}
}
}
}
```

# 1

## DESARROLLO DE SOFTWARE

En esta primera parte de la unidad vamos a aprender a reconocer elementos y herramientas que se usan para desarrollar un programa informático, así como las características y fases que tiene que pasar hasta su puesta en funcionamiento.

Definiremos conceptos como qué es el software y su ciclo de vida. Analizaremos los distintos tipos de lenguajes de programación con sus características y veremos las fases de desarrollo de la ingeniería del software.

## 1.1. EL SOFTWARE DEL ORDENADOR

Antes de comenzar con la definición de software, es necesario aclarar la diferencia entre hardware y software. El ordenador está compuesto por dos partes: la parte física, que llamamos **hardware** y que está compuesta por elementos físicos como el teclado, el ratón, el monitor, los discos duros o la placa base, entre otros elementos. En definitiva, lo forman todos aquellos componentes que podemos ver y tocar. Por otro lado, el ordenador posee otra parte lógica llamada **software**, encargada de dar instrucciones al hardware y hacer funcionar la computadora.

En este apartado veremos conceptos básicos para el desarrollo del software.

Además de dar instrucciones al hardware, el software también almacenará los datos necesarios para ejecutar los programas y contendrá los datos almacenados del ordenador.

Podemos dividir el software en dos categorías: según las **tareas que realiza** y según su **método de distribución**.

BASADAS EN EL TIPO DE TRABAJO QUE REALIZAN	
 	<ul style="list-style-type: none"> <li>· De sistema</li> <li>· De aplicación</li> <li>· De programación</li> </ul>

BASADAS EN EL MÉTODO DE DISTRIBUCIÓN	
 	<ul style="list-style-type: none"> <li>· Shareware</li> <li>· Freeware</li> <li>· Adware</li> </ul>

### 1.1.1. SOFTWARE BASADO EN EL TIPO DE TRABAJO QUE REALIZAN

Según esta clasificación, podemos distinguir tres tipos de software:

- **Software de sistema:** es el que hace que el hardware funcione. Está formado por programas que administran la parte física e interactúa entre los usuarios y el hardware. Algunos ejemplos son los sistemas operativos, los controladores de dispositivos, las herramientas de diagnóstico, etcétera.
- **Software de aplicación:** aquí tendremos los programas que realizan tareas específicas para que el ordenador sea útil al usuario, por ejemplo, los programas ofimáticos, el software médico o el de diseño asistido, entre otros.
- **Software de programación o desarrollo:** es el encargado de proporcionar al programador las herramientas necesarias para escribir los programas informáticos y para hacer uso de distintos lenguajes de programación. Entre ellos encontramos los entornos de desarrollo integrado (IDE).

### 1.1.2. SOFTWARE BASADO EN EL MÉTODO DE DISTRIBUCIÓN

Mediante esta clasificación, también distinguimos tres tipos de software:

- **Shareware:** el usuario puede evaluar de forma gratuita el producto, pero con limitaciones en algunas características. Si realiza un pago, podrá disfrutar del software sin limitaciones. Por ejemplo, Malwarebytes.
- **Freeware:** donde los usuarios de software pueden descargar el aplicativo de forma gratuita, pero que mantiene los derechos de autor. Sería el caso de Firefox.
- **Adware:** es un aplicativo donde se ofrece publicidad incrustada, incluso durante la instalación de este. Por ejemplo, CCleaner.

### 1.1.3. LICENCIAS DE SOFTWARE. SOFTWARE LIBRE Y PROPIETARIO

Una **licencia** es un contrato entre el desarrollador de un software y el usuario final. En él se especifican los derechos y deberes de ambas partes. Es el desarrollador el que especifica qué tipo de licencia distribuye.

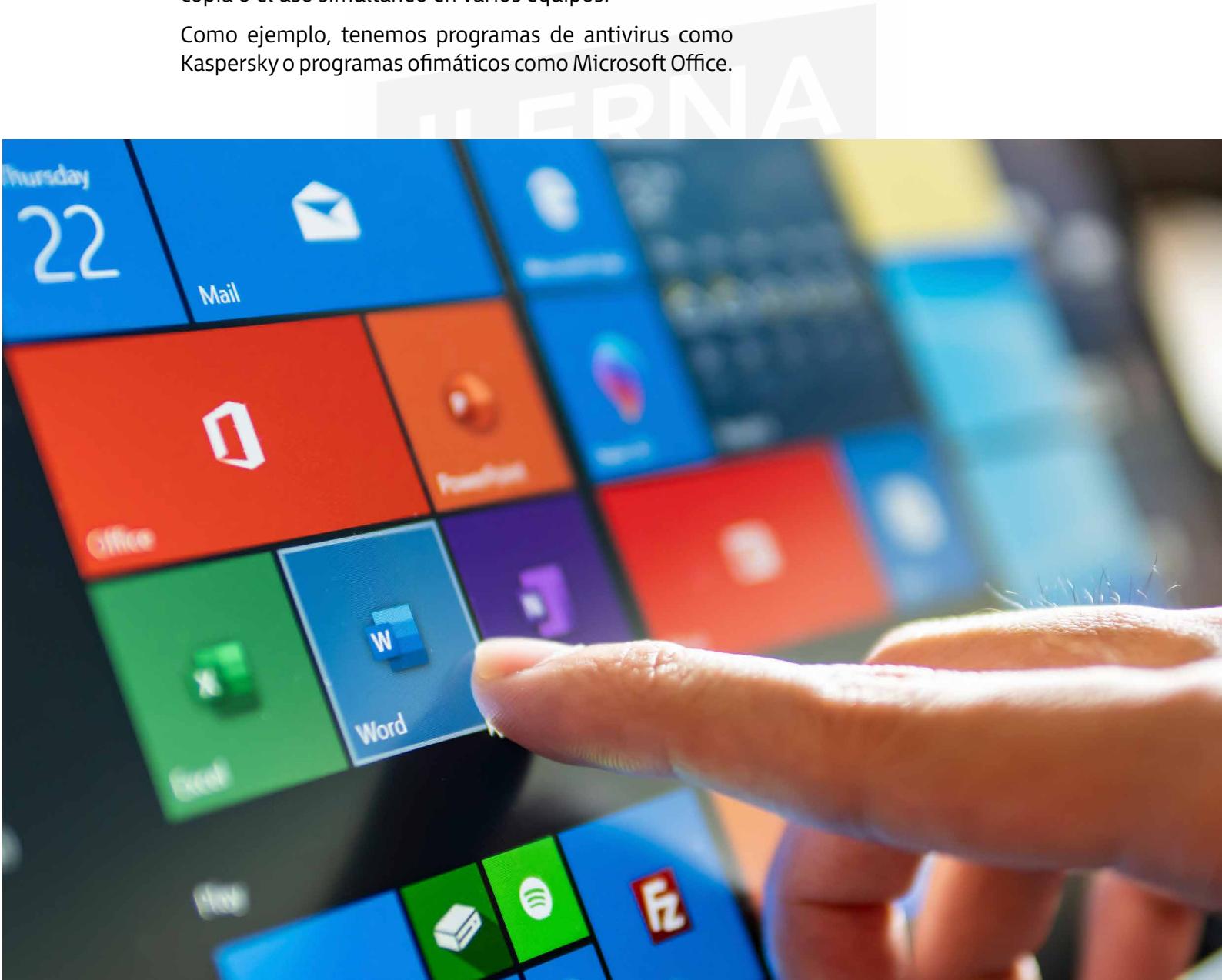
Existen dos tipos de licencias:

- **Software libre:** el autor de la licencia concede libertades al usuario, entre ellas están:
  - Libertad para usar el programa con cualquier fin.
  - Libertad para saber cómo funciona el programa y adaptar el código a nuestras propias necesidades.
  - Libertad para poder compartir copias con otros usuarios.
  - Libertad para poder mejorar el programa y publicar las modificaciones realizadas.

Hay que dejar claro que la palabra “libre” en este contexto no significa que sea gratis. El **software libre** es un concepto de libertad, no de precio. Nos referimos a la libertad de los usuarios para ejecutar, copiar, distribuir, cambiar, reutilizar y mejorar el software.

- **Software propietario:** este software no nos permitirá acceder al código fuente del programa y, de forma general, nos prohibirá la redistribución, la reprogramación, la copia o el uso simultáneo en varios equipos.

Como ejemplo, tenemos programas de antivirus como Kaspersky o programas ofimáticos como Microsoft Office.



La licencia que más se usa en el software libre es la licencia **GPL** (*general public license* o licencia pública general), que nos dejará usar y cambiar el programa, con el único requisito de que se hagan públicas las modificaciones realizadas.

**BUSCA EN LA WEB**

Para encontrar más información sobre licencia GPL, podremos visitar la siguiente dirección:

[www.gnu.org/licenses/license-list.es.html#SoftwareLicenses](http://www.gnu.org/licenses/license-list.es.html#SoftwareLicenses)



**ponte a prueba**

**¿En qué tipo de método de distribución estaría el siguiente software?**



- a) Adware
- b) Shareware
- c) Freeware
- d) Jailware

**El software libre puede ser vendido.**

- a) Verdadero
- b) Falso

## 1.2. CONCEPTO DE PROGRAMA INFORMÁTICO

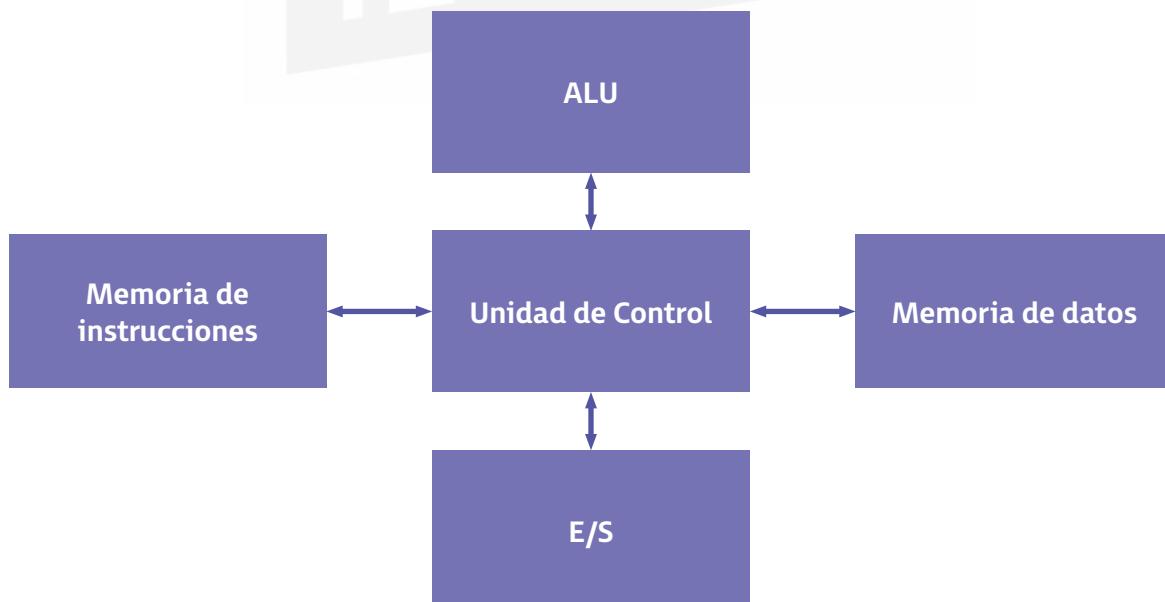
Un **programa informático** es un fragmento de software formado por una secuencia de instrucciones y procesos construido con el fin de cumplir un objetivo concreto. Estos programas informáticos están escritos utilizando lenguajes de programación como Java, C# o Python y deben ser traducidos al lenguaje máquina para que puedan ser procesados por nuestro ordenador.

### 1.2.1. PROGRAMA Y COMPONENTES DEL SISTEMA INFORMÁTICO

El diseño de la arquitectura de nuestra CPU (en inglés, *central processing unit*) está basado en la arquitectura de **Von Neumann**<sup>1</sup>.

Por el contrario, tenemos otra estructura que es la llamada **Harvard**<sup>2</sup>. La característica fundamental es que tanto la memoria RAM como la memoria de instrucciones no comparten características comunes.

La unidad de control (UC) es el centro de la arquitectura y conecta con la unidad aritmético-lógica (UAL), los dispositivos de entrada y salida y con ambas memorias mencionadas anteriormente.



1 MacTutor History of Mathematics Archive: [https://mathshistory.st-andrews.ac.uk/Biographies/Von\\_Neumann/](https://mathshistory.st-andrews.ac.uk/Biographies/Von_Neumann/)

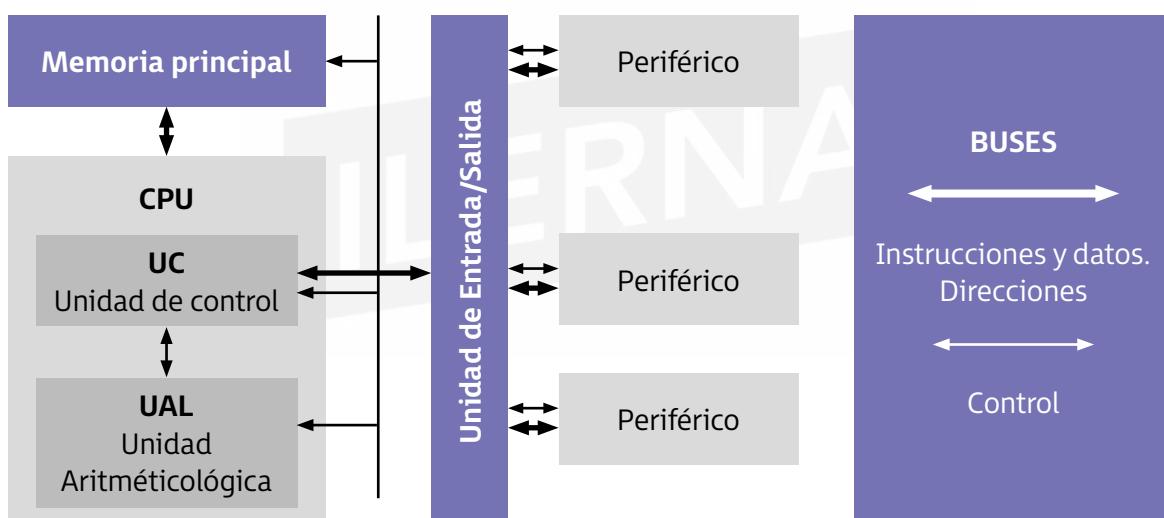
2 García Ortega, Víctor H.; Sosa Savedra, Julio C.; Ortega S., Susana; Tovar, Rubén H. (2009) *Microprocesador didáctico de arquitectura RISC implementado en un FPGA*. E-Gnosis. <https://www.redalyc.org/pdf/730/73012215014.pdf>

En la arquitectura de Von Neumann, puede estar leyendo una instrucción o bien escribir un dato en la memoria, pero ambos no pueden ocurrir a la vez, ya que utilizan el mismo sistema de buses.

En la arquitectura Harvard, la CPU puede tanto leer una instrucción como realizar un acceso de memoria de datos.

Si vemos la arquitectura Von Neumann, entenderemos cómo funcionan los componentes que conforman la CPU:

- **La unidad de control (UC):** se encarga de interpretar y ejecutar las instrucciones que se almacenan en la memoria principal y, además, genera las señales de control necesarias para ejecutarlas.
- **La unidad aritmético-lógica (UAL):** es la que recibe los datos y ejecuta operaciones de cálculo y comparaciones, además de tomar decisiones lógicas (si son verdaderas o falsas), pero siempre supervisada por la unidad de control.
- **Los registros:** son aquellos que almacenan la información temporal. Es el almacenamiento interno de la CPU.



A continuación, vamos a ver los diferentes registros que posee la UC:

- **Contador de programa (CP):** contendrá la dirección de la siguiente instrucción para realizar. Su valor será actualizado por la CPU después de capturar una instrucción.
- **Registro de instrucción (RI):** es el que contiene el código de la instrucción, se analiza dicho código. Consta de dos partes: el código de la operación y la dirección de memoria en la que opera.
- **Registro de dirección de memoria (RDM):** tiene asignada una dirección correspondiente a una posición de memoria que va a almacenar la información mediante el bus de direcciones.

- **Registro de intercambio de memoria (RIM):** recibe o envía, según si es una operación de lectura o escritura, la información o dato contenido en la posición apuntada por el RDM.
- **Decodificador de instrucción (DI):** extrae y analiza el código de la instrucción contenida en el RI.
- **El reloj:** marca el ritmo del DI y nos proporciona unos impulsos eléctricos con intervalos constantes a la vez que marca los tiempos para ejecutar las instrucciones.
- **El secuenciador:** son órdenes que se sincronizan con el reloj para que ejecuten correctamente y de forma ordenada la instrucción.

Cuando ejecutamos una instrucción podemos distinguir dos fases:

1. **Fase de búsqueda:** se localiza la instrucción en la memoria principal y se envía a la unidad de control para poder procesarla.
2. **Fase de ejecución:** se ejecutan las acciones de las instrucciones.

Para que podamos realizar operaciones de lectura y escritura en una celda de memoria, se utilizan el RDM, el RIM y el DI. El decodificador de instrucción es el encargado de conectar la celda RDM con el registro de intercambio RIM, el cual posibilita que la transferencia de datos se realice en un sentido u otro según sea de lectura o escritura.



### ponte a prueba

**¿Qué tipo de lenguaje de programación es Python?**

- a) Alto nivel
- b) Bajo nivel
- c) Ensamblador
- d) Ninguna de las respuestas es correcta

**¿Qué función realiza la ALU?**

- a) Operaciones aritméticas
- b) Decisiones lógicas
- c) Operaciones de comparación
- d) Todas las opciones son correctas



## 1.3. CÓDIGO FUENTE, CÓDIGO OBJETO Y CÓDIGO EJECUTABLE. MÁQUINAS VIRTUALES

En la etapa de diseño construimos las herramientas de software capaces de generar un código fuente en lenguaje de programación. Estas herramientas pueden ser diagramas de clase (realizados con el lenguaje de modelado **UML**), pseudocódigo o diagramas de flujo.

La etapa de codificación es la encargada de generar el código fuente, y pasa por diferentes estados.

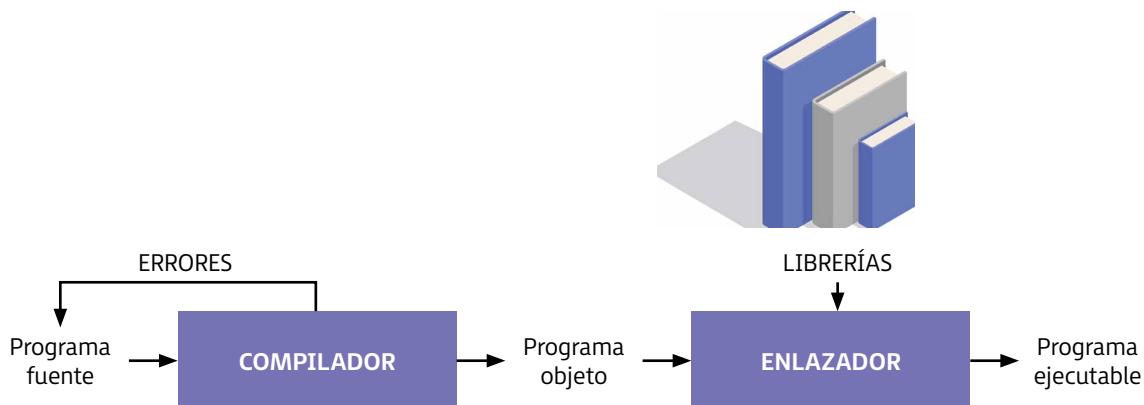
### 1.3.1. TIPOS DE CÓDIGO

Cuando escribimos un código, pasa por distintos estados hasta que se ejecuta:

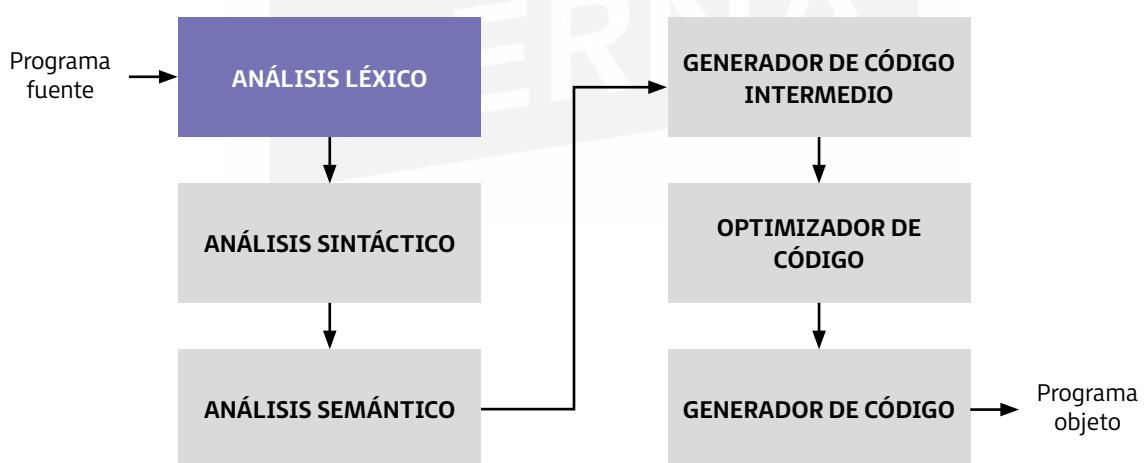
- **Código fuente:** es el código realizado por los programadores usando algún editor de texto o herramienta de programación. Posee un lenguaje de alto nivel y para escribirlo se parte de, por ejemplo, diagramas de clases. No se puede ejecutar directamente en el ordenador.
- **Código objeto:** es el código que se crea tras realizar la compilación del código fuente. Este código no es entendido ni por el ordenador ni por nosotros. Es una representación intermedia de bajo nivel.
- **Código ejecutable:** este código se obtiene tras unir el código objeto con varias librerías para que así pueda ser ejecutado por el ordenador.

### 1.3.2. COMPILACIÓN

La compilación es el proceso a través del cual se convierte un programa en lenguaje máquina a partir de otro programa de computadora escrito en otro lenguaje. La compilación se realiza a través de dos programas: el compilador y el enlazador. Si en el **compilador** se detecta algún tipo de error no se generará el código objeto y tendremos que modificar el código fuente para volver a pasarlo por el compilador.



Dentro del compilador, tendremos varias **fases** en las que se realizan distintas operaciones:



- **Análisis léxico:** se lee el código obteniendo unidades de caracteres llamados *tokens*. Ejemplo: la instrucción  $= 2 - 1$ , genera 5 *tokens*: resta, =, 2, -, 1.
- **Análisis sintáctico:** recibe el código fuente en forma de *tokens* y ejecuta el análisis para determinar la estructura del programa, se comprueba si cumplen las reglas sintácticas.
- **Análisis semántico:** revisa que las declaraciones sean correctas, los tipos de todas las expresiones, si las operaciones se pueden realizar, si los arrays son del tamaño correcto, etcétera.

- **Generación de código intermedio:** después de analizarlo todo, se crea una representación similar al código fuente para facilitar la tarea de traducir al código objeto.
- **Optimización de código:** se mejora el código intermedio anterior para que sea más fácil y rápido a la hora de interpretarlo la máquina.
- **Generación de código:** se genera el código objeto.

El **enlazador** insertará en el código objeto las librerías necesarias para que se pueda producir un programa ejecutable. Si se hace referencia a otros ficheros que contengan las librerías especificadas en el código objeto, se combina con dicho código y se crea el fichero ejecutable.

### 1.3.3. MÁQUINAS VIRTUALES

Una máquina virtual es un tipo de software capaz de ejecutar programas como si fuese una máquina real. Se clasifican en dos categorías:

- **Máquinas virtuales de sistema.** Nos permiten virtualizar máquinas con distintos sistemas operativos en cada una. Un ejemplo son los programas VMware Workstation o Virtual Box, que podremos usar para probar nuevos sistemas operativos o ejecutar programas.
- **Máquinas virtuales de proceso.** Se ejecutan como un proceso normal dentro de un sistema operativo y solo soportan un proceso. Se inician cuando lanzamos el proceso y se detienen cuando este finaliza. El objetivo es proporcionar un entorno de ejecución independiente del hardware y del sistema operativo y permitir que el programa sea ejecutado de la misma forma en cualquier plataforma. Ejemplo de ello es la máquina virtual de Java (JVM).

Las máquinas virtuales requieren de grandes recursos, por lo que hay que tener en cuenta dónde las vamos a ejecutar. Los procesadores tienen que ser capaces de soportar dichas máquinas para que no se ralentice o colapse el resto del sistema.

#### La máquina virtual de Java

Los programas que se compilan en lenguaje Java son capaces de funcionar en cualquier plataforma (UNIX, Mac, Windows, Solaris, etc.). Esto se debe a que el código no lo ejecuta el procesador del ordenador sino la propia *Máquina Virtual de Java* (JVM).



El funcionamiento básico de la máquina virtual es el siguiente:

1. El código fuente estará escrito en archivos de texto planos con la extensión .java.
2. El compilador **javac** generará uno o varios archivos, siempre que no se produzcan errores, y tendrán la extensión .class.
3. Este fichero .class contendrá un lenguaje intermedio entre el ordenador y el sistema operativo y se llamará *bytecode*.
4. La JVM coge y traduce mediante un compilador JIT (siglas en inglés de compilación en tiempo de ejecución) el *bytecode* en código binario para que el procesador de nuestro ordenador sea capaz de reconocerlo.
5. Los ficheros .class podrán ser ejecutados en múltiples plataformas.

La máquina virtual de Java contiene, entre otras, las instrucciones para las siguientes tareas:

- Carga y almacenamiento de datos.
- Excepciones (errores en tiempo de ejecución).
- Operaciones aritméticas.
- Conversiones de tipos de datos.
- Llamadas a métodos y devolución de datos.
- Creación y manejo de objetos.

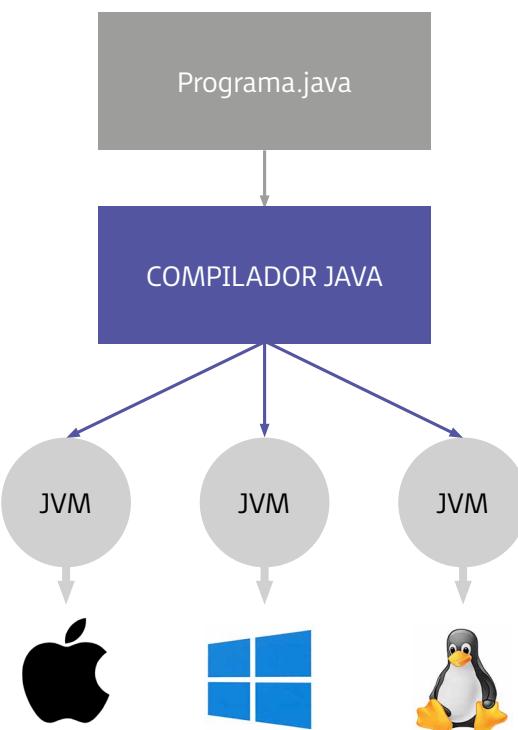
● ● ●

**BUSCA EN LA WEB**

[www.oracle.com/technetwork/java/  
javase/downloads/index.html](http://www.oracle.com/technetwork/java/javase/downloads/index.html)




Una de las desventajas de usar este tipo de lenguajes que se basan en una máquina virtual puede ser que son más lentos que los lenguajes ya compilados, debido a la capa intermedia.



## 1.4. TIPOS DE LENGUAJES DE PROGRAMACIÓN. CLASIFICACIÓN Y CARACTERÍSTICAS DE LOS LENGUAJES MÁS DIFUNDIDOS

Como hemos definido anteriormente, un programa informático es un conjunto de instrucciones escritas en un lenguaje de programación. Asimismo, lenguaje de programación hace referencia al conjunto de caracteres, reglas y acciones combinadas y consecutivas que un equipo debe ejecutar.

Constará de los siguientes elementos:

- **Alfabeto o vocabulario:** conjunto de símbolos permitidos.
- **Sintaxis:** reglas para realizar correctamente construcciones con los símbolos.
- **Semántica:** reglas que determinan el significado de construcción del lenguaje.

### 1.4.1. CLASIFICACIÓN Y CARACTERÍSTICAS

Podemos clasificar los lenguajes de programación basándonos en los siguientes criterios:

<b>Según su nivel de abstracción:</b>	Lenguajes de bajo nivel. Lenguajes de nivel medio. Lenguajes de alto nivel.
<b>Según la forma de ejecución:</b>	Lenguajes compilados. Lenguajes interpretados.
<b>Según el paradigma de programación:</b>	Lenguajes imperativos. Lenguajes funcionales. Lenguajes lógicos. Lenguajes estructurados. Lenguajes orientados a objetos.

#### Según su nivel de abstracción

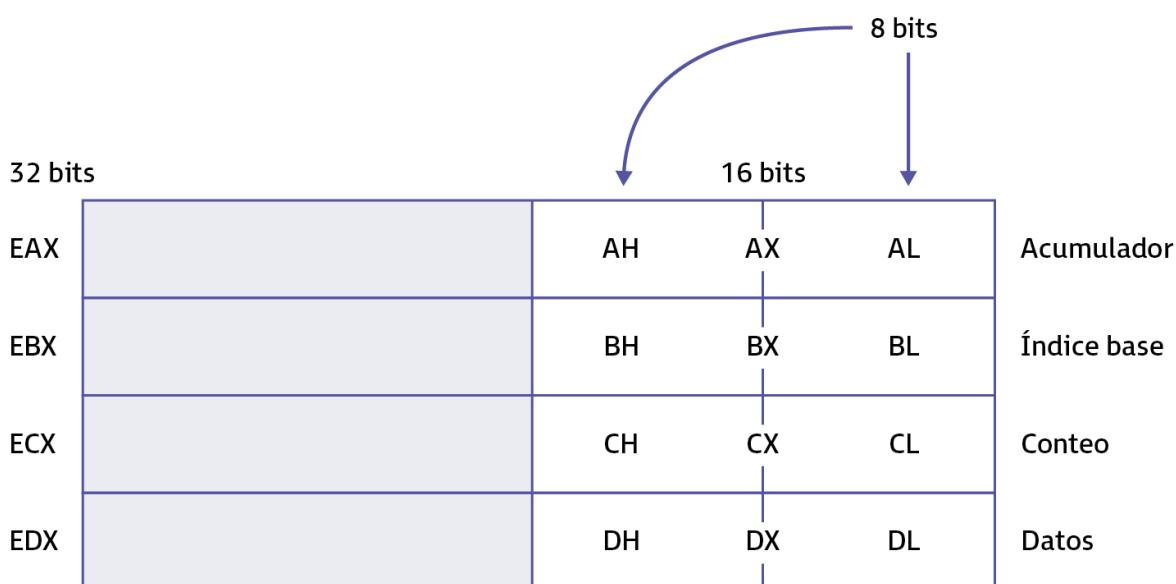
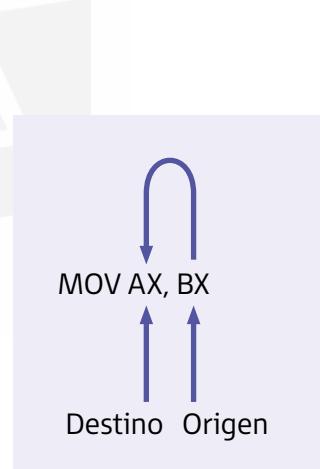
- **Lenguajes de bajo nivel:** el lenguaje de más bajo nivel por excelencia es **el lenguaje máquina**, el que entiende directamente la máquina. Utiliza el lenguaje binario (0 y 1) y los programas son específicos para cada procesador.

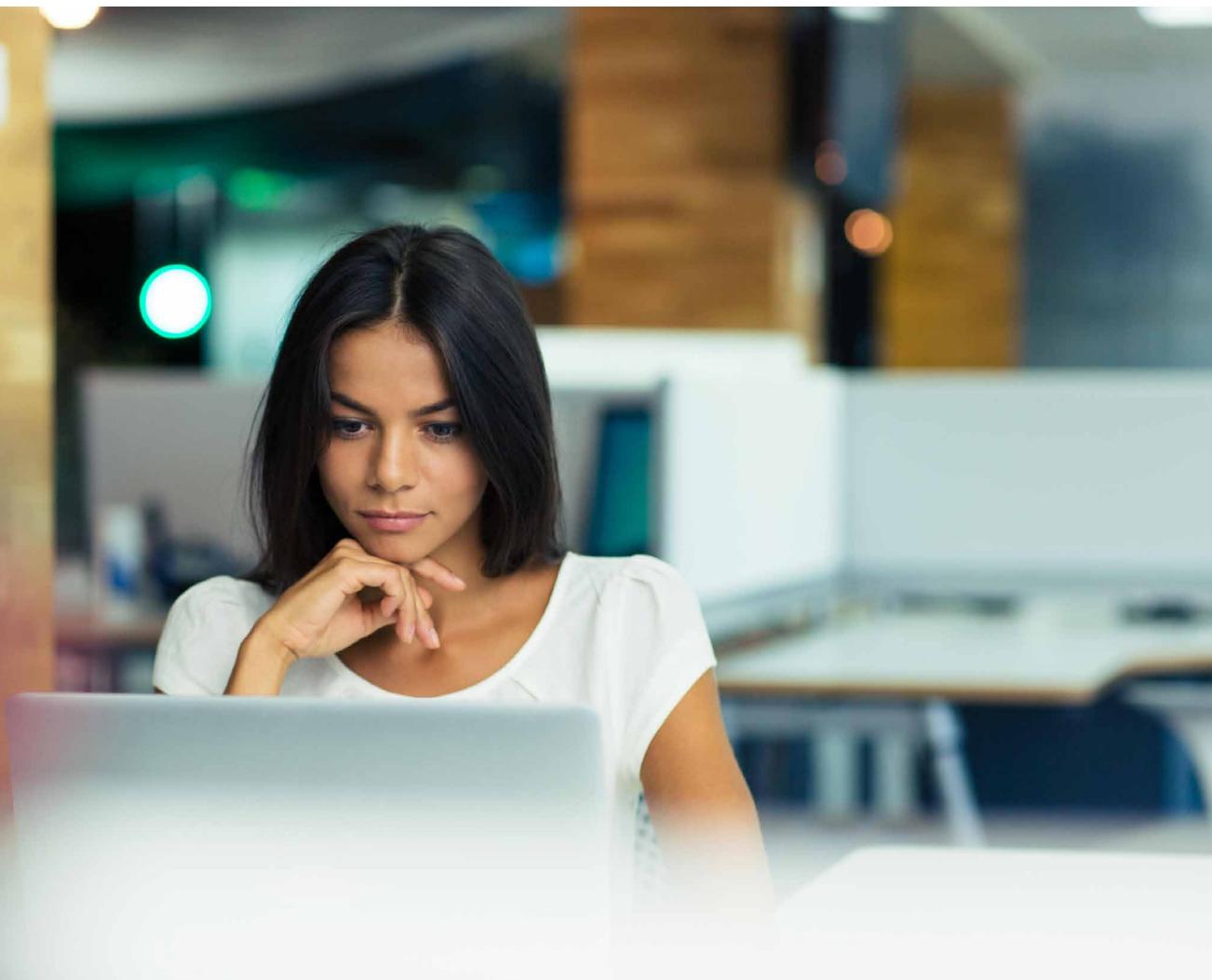
Al lenguaje máquina le sigue el **lenguaje ensamblador**. Es complicado de aprender y es específico para cada procesador. Cualquier programa escrito en este lenguaje tiene que ser traducido al lenguaje máquina para que se pueda ejecutar. Se utilizan nombres mnemotécnicos y las instrucciones trabajan directamente con registros de memoria física.

En el lenguaje ensamblador, podemos encontrar:

- Los registros extendidos de 32 bits, como EAX (registro acumulador), EBX (registro de índice base), ECX (registro de conteo) o EDX (registro de datos).
- Estos registros se dividen en registros de menor tamaño, como AX, BX, CX y DX, de 16 bits, y estos, a su vez, en otros de 8 bits, como AH, AL, BH, BL, CH, CL, DH o DL.
- **Registro EAX:** el acumulador se utiliza para instrucciones tales como multiplicación o división.
- **Registro EBX:** guarda la dirección de desplazamiento de una posición en el sistema de memoria.
- **Registro ECX:** es un registro de propósito general que guarda la cuenta de varias instrucciones. Realiza funciones de contador.
- **Registro EDX:** es un registro de propósito general que almacena datos de, por ejemplo, aplicaciones aritméticas como el divisor antes de hacer una división.

Podemos realizar operaciones con estos registros. Por ejemplo, una instrucción ADD ECX, EBX suma el contenido de 32 bits de EBX con el de ECX (solo ECX cambia debido a esta instrucción) o mover el contenido de un registro a otro MOV AX, BX (el contenido de BX se almacenaría en el registro AX).





- **Lenguajes de nivel medio:** poseen características de ambos tipos de nivel, tanto del nivel bajo como del alto, y se suele usar para la creación de sistemas operativos. Un lenguaje de nivel medio es el lenguaje C.
- **Lenguajes de alto nivel:** este tipo de lenguaje es más fácil a la hora de aprender, ya que estos lenguajes utilizan nuestro lenguaje natural. El idioma que se suele emplear es el inglés y, para poder ejecutar lo que escribamos, necesitaremos un compilador para que traduzca al lenguaje máquina las instrucciones.

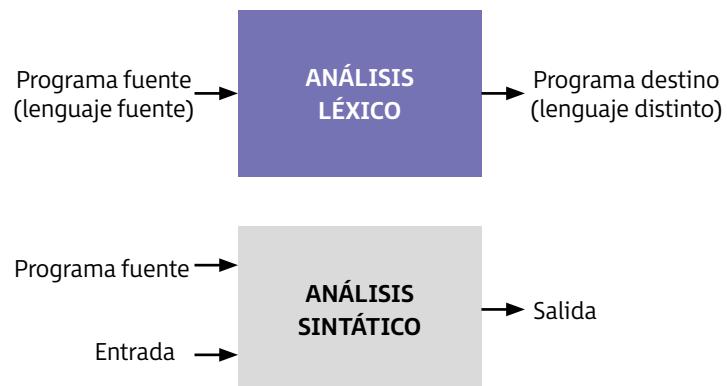
Este lenguaje es independiente de la máquina, ya que no depende del hardware del ordenador.

Algunos ejemplos de lenguajes de alto nivel son ALGOL, C++, C#, Clipper, COBOL, Fortran, Java, Logo y Pascal.

#### Según la forma de ejecución

- **Lenguajes compilados:** al programar en alto nivel, hay que traducir ese lenguaje a lenguaje máquina a través de compiladores.

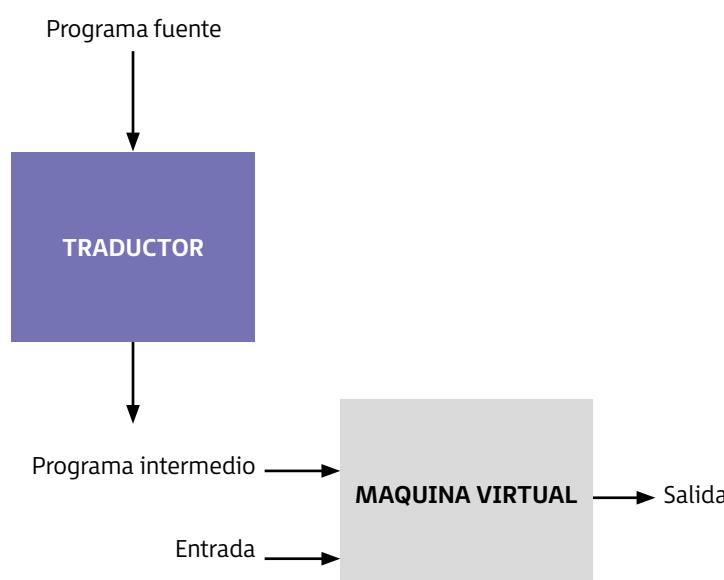
Los compiladores traducen desde un lenguaje fuente a un lenguaje destino. Devolverán errores si el lenguaje fuente está mal escrito y lo ejecutarán si el lenguaje destino es ejecutable por la máquina. Por ejemplo: C, C++, C#, Objective-C.



- **Lenguajes interpretados:** son otra variante para traducir programas de alto nivel. En este caso, nos da la apariencia de ejecutar directamente las instrucciones del programa fuente con las entradas proporcionadas por el usuario. Cuando ejecutamos una instrucción, se debe interpretar y traducir al lenguaje máquina.

El compilador es, de forma general, más rápido que un intérprete al asignar las salidas. Sin embargo, al usar el intérprete evitaremos tener que compilar cada vez que hagamos alguna modificación. Ejemplos de algunos lenguajes son: PHP, JavaScript, Python, Perl, Logo, Ruby, ASP y Basic.

El lenguaje Java usa tanto la compilación como la interpretación. Un programa fuente en Java puede compilarse primero en un formato intermedio, llamado *bytecodes*, para que luego una máquina virtual lo interprete.



## Según el paradigma de programación

El paradigma de programación nos detalla las reglas, los patrones y los estilos de programación que usan los lenguajes. Cada lenguaje puede usar más de un paradigma, el cual resultará más apropiado que otro según el tipo de problema que queramos resolver.

Existen diferentes categorías de lenguaje:

- **Lenguajes imperativos:** al principio, los primeros lenguajes imperativos que se usaron fueron el lenguaje máquina y, más tarde, el lenguaje ensamblador. Ambos lenguajes consisten en una serie de sentencias que establecen cómo debe manipularse la información digital presente en cada memoria o cómo se debe enviar o recibir la información en los dispositivos.

A través de las estructuras de control podemos establecer el orden en que se ejecutan y modificar el flujo del programa según los resultados de las acciones. Facilitan las operaciones por medio de cambios de estado, siendo este la condición de una memoria de almacenamiento.

Algunos ejemplos de estos lenguajes son: Basic, Fortran, Algol, Pascal, C, Ada, C++, Java, C#. Casi todos los **lenguajes de desarrollo de software comercial** son imperativos.

Dentro de esta categoría, podremos englobar:

- **Programación estructurada.**
- **Programación modular.**
- **Programación orientada a objetos** (usa objetos y sus interacciones para crear programas).

- **Lenguajes funcionales:** están basados en el concepto de función y estarán formados por definiciones de funciones junto con sus argumentos.

Entre sus características destaca que no existe la operación de asignación. Las variables almacenan definiciones a expresiones. El resultado de un cálculo es la entrada del siguiente, y así sucesivamente hasta que se consigue el valor deseado.

Sería el caso de tipos de lenguaje como Lisp, Scheme, ML, Miranda o Haskell. Apenas se usan para el software comercial.

- **Lenguajes lógicos:** están basados en el concepto de razonamiento, ya sea de tipo deductivo o inductivo. A partir de una base de datos consistente en un conjunto de entidades, propiedades de esas entidades o relaciones entre entidades, el sistema es capaz de hacer **razonamientos**.

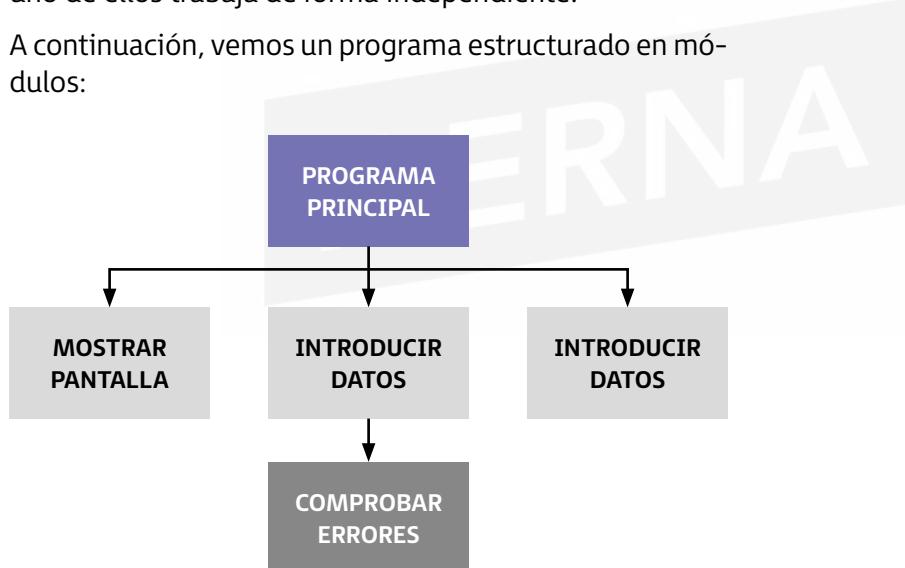
Los programas escritos en este lenguaje suelen tener forma de una **base de datos**, la cual está formada por declaraciones lógicas, es decir, que son ciertas o falsas, que podremos consultar. La ejecución será en forma de consultas hacia esa base de datos.

El lenguaje lógico más importante es Prolog, especialmente preparado para sistemas expertos, demostración de teoremas, consultas de bases de datos relacionales y procesamiento de lenguaje natural.

- **Lenguajes estructurados:** utilizan las tres construcciones lógicas nombradas anteriormente y resultan fáciles de leer. El inconveniente de estos programas estructurados es el código, que está centrado en un solo bloque, lo que dificulta el proceso de hallar el problema.

Cuando hablamos de programación estructurada nos estamos refiriendo a programas creados a través de módulos, es decir, pequeñas partes más manejables que, unidas entre sí, hacen que el programa funcione. Cada uno de los módulos poseen una entrada y una salida y deben estar perfectamente comunicados, aunque cada uno de ellos trabaja de forma independiente.

A continuación, vemos un programa estructurado en módulos:



La evolución a esta programación mediante módulos se le denomina **programación modular** y posee las siguientes ventajas:

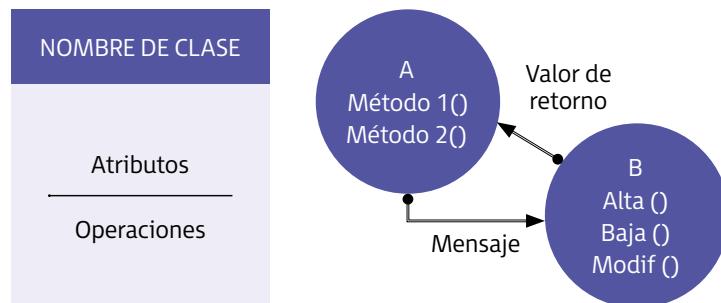
- Al dividir el programa en módulos, varios programadores podrán trabajar a la vez en cada uno de ellos.
- Estos módulos pueden usarse para otras aplicaciones.
- Si surge algún problema será más fácil y menos costoso detectarlo y abordarlo, ya que se puede resolver de forma aislada.

Algunos ejemplos de estos lenguajes son Pascal, C, Fortran y Modula-2.

- **Lenguajes orientados a objetos:** estos lenguajes están definidos por un conjunto de objetos en vez de por módulos, como hemos visto anteriormente.

Estos objetos están formados por una estructura de datos y por una colección de métodos que interpretan esos datos. Los datos que se encuentran dentro de los objetos son sus **atributos**, y las operaciones que se realizan sobre los objetos cambian el valor de uno o más atributos.

La comunicación entre objetos se realiza a través de mensajes, como se plasma en la siguiente figura:



Una clase es una plantilla para la creación de objetos. Al crear un objeto, se ha de especificar a qué clase pertenece para que el compilador sepa qué características posee.

Entre las ventajas de este tipo de lenguaje hay que destacar la facilidad para reutilizar el código, el trabajo en equipo o el mantenimiento del software.

Una desventaja es que el concepto de un programador puede ser distinto a otros, por lo que se realiza una división entre objetos distinta.

Los lenguajes orientados a objetos más comunes son C++, Java, Ada, Smalltalk y Ruby, entre otros.



### ponte a prueba

**¿Qué capacidad (en bits) tiene el registro EAX?**

- a) 8 bits
- b) 16 bits
- c) 32 bits
- d) El registro EAX no existe

**¿Cuál de los siguientes lenguajes no son de alto nivel?**

- a) Python
- b) Java
- c) C
- d) Ensamblador

## 1.5. FASES DEL DESARROLLO DE UNA APLICACIÓN: ANÁLISIS, DISEÑO, CODIFICACIÓN, PRUEBAS, DOCUMENTACIÓN, MANTENIMIENTO Y EXPLOTACIÓN

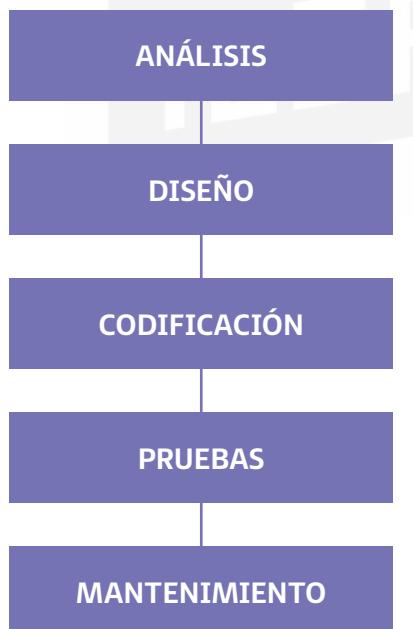
**Modelos de desarrollo**  
[youtu.be/MibXdrHxYVw](https://youtu.be/MibXdrHxYVw)



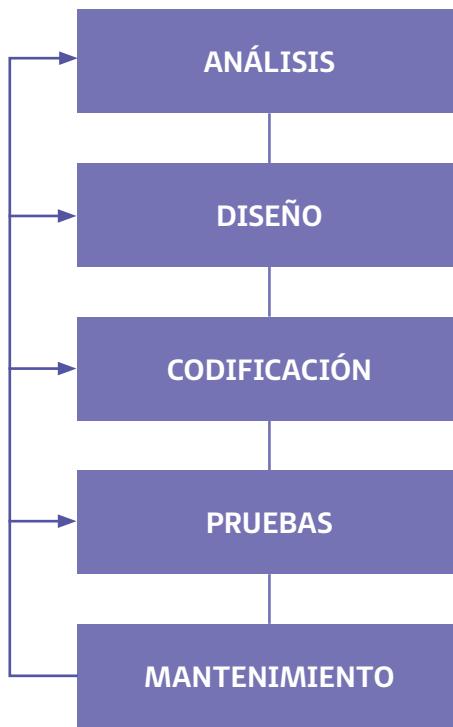
Cuando queramos realizar un proyecto de software, antes debemos crear un ciclo de vida en el que examinemos las características para elegir un modelo de desarrollo u otro.

### Modelo en cascada

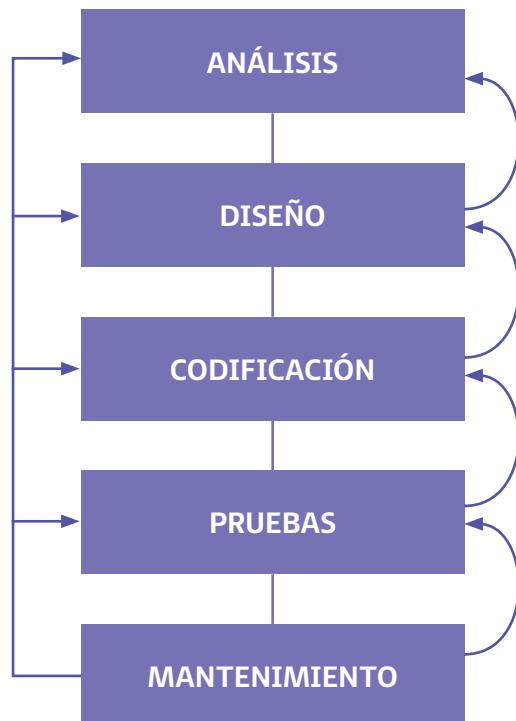
En este modelo, las etapas para el desarrollo de software tienen un orden, de tal forma que, para empezar una etapa, es necesario finalizar la etapa anterior. Después de cada etapa se realiza una revisión para comprobar si se puede pasar a la siguiente.



Este modelo permite hacer iteraciones. Por ejemplo, si el cliente requiere una mejora durante la etapa de mantenimiento del producto, esto implica que hay que modificar algo en el diseño, lo cual significa que habrá que hacer cambios en la codificación y se tendrán que realizar de nuevo las pruebas. Es decir, si se tiene que volver a una de las etapas anteriores, hay que recorrer de nuevo el resto de las etapas.



Este modelo tiene distintas variantes, una de la más utilizadas es el **modelo en cascada con realimentación**, que produce una realimentación entre etapas. Supongamos que en cualquiera de las etapas se detectan fallos (los requisitos han cambiado, han evolucionado, ambigüedades en la definición de estos, etcétera), entonces, será necesario retornar a la etapa anterior para realizar los ajustes pertinentes. A esto se le conoce como realimentación, pudiendo volver de una etapa a la anterior o incluso de varias etapas a otra anterior.

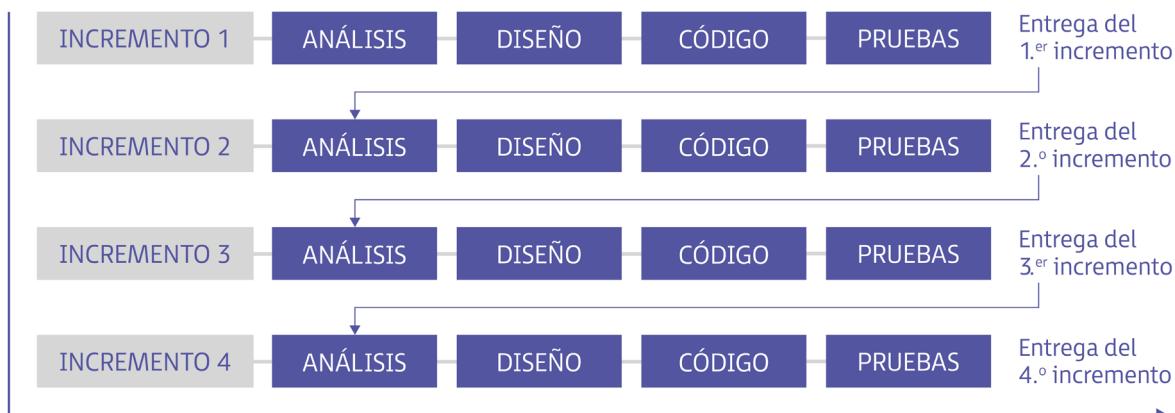


Ventajas	Inconvenientes
Fácil de comprender, planificar y seguir.	La necesidad de tener todos los requisitos definidos desde el principio.
La calidad del producto resultante es alta.	Es difícil volver atrás si se cometen errores en una etapa (es un modelo inflexible)
Los recursos que se necesitan son mínimos.	El producto no está disponible para su uso hasta que no está completamente terminado.
<b>Se recomienda cuando:</b>	
<ul style="list-style-type: none"> <li>• El proyecto es similar a alguno que ya se haya realizado con éxito anteriormente.</li> </ul>	
<ul style="list-style-type: none"> <li>• Los requisitos son estables y están bien comprendidos.</li> </ul>	
<ul style="list-style-type: none"> <li>• Los clientes no necesitan versiones intermedias.</li> </ul>	

### Modelo iterativo incremental

El modelo incremental está basado en varios ciclos en cascada realimentados aplicados repetidamente. Este modelo entrega el software en partes pequeñas, pero utilizables, llamadas incrementos o prototipos. En general, cada incremento se construye sobre aquel que ya ha sido entregado.

A continuación, se muestra un diagrama del modelo bajo un esquema temporal donde se observa de forma iterativa el modelo en cascada para la obtención de un nuevo incremento mientras progresá el tiempo en el calendario.



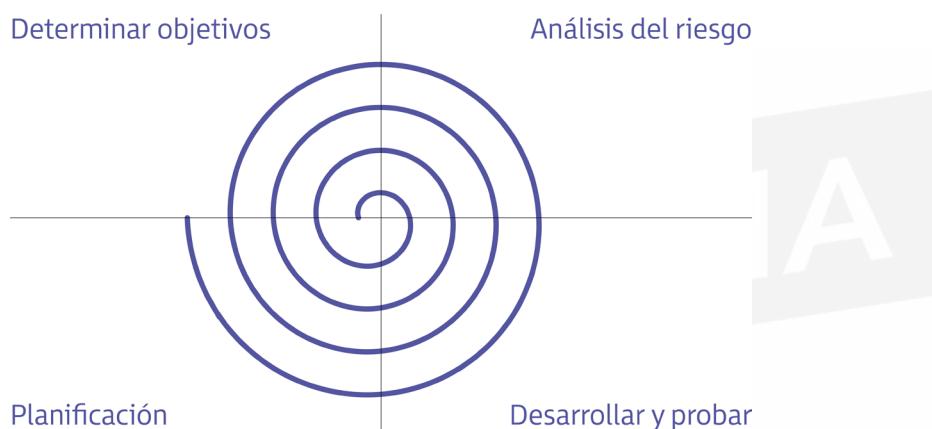
Al desarrollar el software de manera incremental, resulta más barato y fácil realizar cambios en el software conforme este se va desarrollando.

Cada incremento del aplicativo incorpora algunas de las funciones que necesita el cliente. Esto significa que el cliente puede evaluar el desarrollo del sistema en una etapa temprana.

Ventajas	Inconvenientes
No necesitan conocer todos los requisitos. Se reduce, por tanto, el coste de adaptar los requerimientos cambiantes del cliente.	Es difícil estimar el esfuerzo y el coste final necesarios.
Permite la entrega temprana al cliente de partes operativas del software.	Se tiene el riesgo de no acabar nunca.
Las entregas facilitan la realimentación de los próximos entregables.	No es recomendable para desarrollo de sistemas de tiempo real, de alto nivel de seguridad, de procesamiento distribuido y/o de alto índice de riesgos.
	La incorporación de muchos cambios hace que el software se vuelva inestable.
<b>Se recomienda cuando:</b>	
<ul style="list-style-type: none"> <li>Los requisitos o el diseño no están completamente definidos y es posible que haya grandes cambios.</li> </ul>	
<ul style="list-style-type: none"> <li>Se están probando o introduciendo nuevas tecnologías.</li> </ul>	

### Modelo en espiral

Este modelo combina el modelo en cascada con el modelo iterativo de construcción de prototipos. El proceso de desarrollo del software se representa como una espiral donde en cada ciclo se desarrolla una parte de este. Cada ciclo está formado por cuatro fases y, cuando se termina, produce una versión incremental del software con respecto al ciclo anterior. En este aspecto, se parece al modelo iterativo incremental, con la diferencia de que en cada ciclo se tiene en cuenta el análisis de riesgos.



Durante los primeros ciclos, la versión incremental podría estar compuesta de maquetas en papel o modelos de pantallas (prototipos de interfaz); en el último ciclo, se tendría un prototipo operacional que implementa algunas funciones del sistema. Para cada ciclo, los desarrolladores siguen estas fases:

- 1. Determinar objetivos:** cada ciclo de la espiral comienza con la identificación de los objetivos, las alternativas para alcanzar los objetivos y las restricciones impuestas a la aplicación de las alternativas.
- 2. Análisis del riesgo:** a continuación, hay que evaluar las alternativas en relación con los objetivos y limitaciones. Con frecuencia, en este proceso se identifican los riesgos involucrados y la manera de resolverlos (requisitos no comprendidos, mal diseño, errores en la implementación, etcétera). Se aconseja realizar un análisis minucioso para reducir los riesgos. Utiliza la construcción de prototipos como mecanismo de reducción de riesgos.



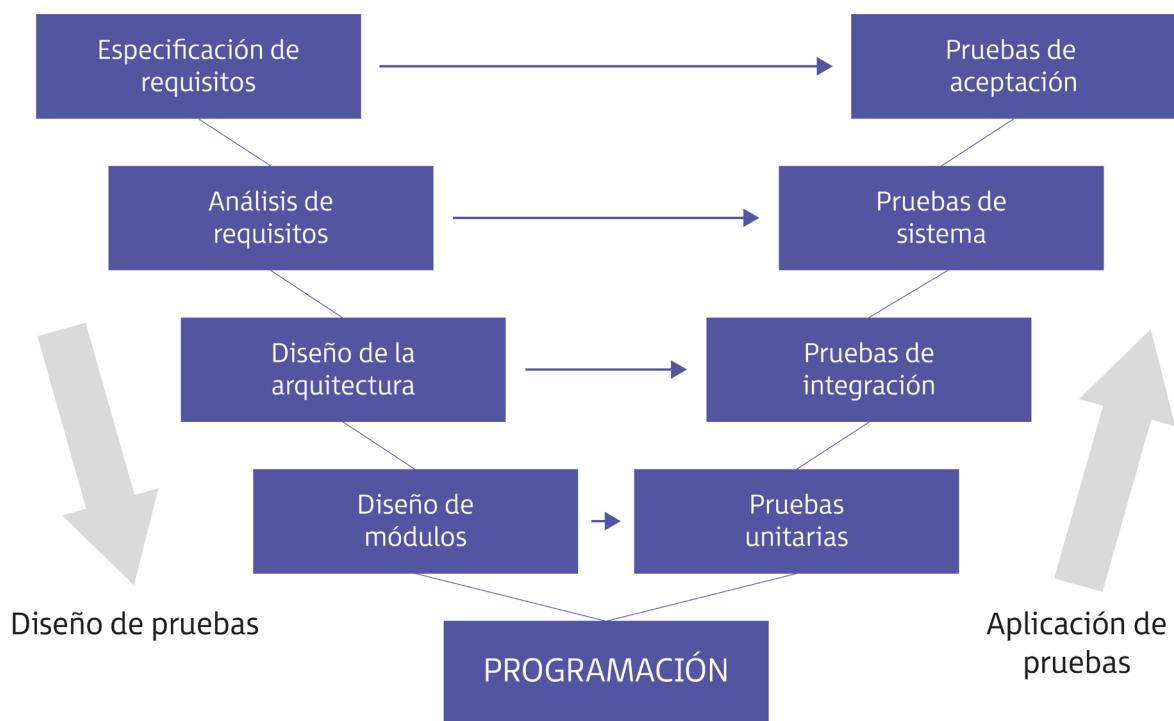
3. **Desarrollar y probar:** desarrollar la solución al problema en este ciclo y verificar que es aceptable. Por ejemplo, si existen riesgos en una interfaz de usuario, podríamos ir creando prototipos hasta conseguir un enfoque deseable.
4. **Planificación:** revisar y evaluar todo lo que se ha hecho y, con ello, decidir si se continua; entonces hay que planificar las fases del ciclo siguiente.

Ventajas	Inconvenientes
No requiere una definición completa de los requisitos para empezar a funcionar.	Es difícil evaluar los riesgos.
Análisis del riesgo en todas las etapas.	El costo del proyecto aumenta a medida que la espiral pasa por sucesivas iteraciones.
Reduce riesgos del proyecto.	El éxito del proyecto depende en gran medida de la fase de análisis de riesgos.
Aumento de la productividad.	Es difícil hacer ver al cliente que este enfoque evolutivo es controlable.
<b>Se recomienda para:</b>	
<ul style="list-style-type: none"><li>• Proyectos de gran tamaño y que necesitan constantes cambios.</li><li>• Proyectos donde sea importante el factor riesgo.</li></ul>	

## Modelo en V

Es un proceso que representa la secuencia de pasos en el desarrollo del ciclo de vida de un proyecto. En él se describen las actividades y resultados que deben producirse durante el desarrollo del producto. El **lado izquierdo** de la V representa la descomposición de las necesidades y la creación de las especificaciones del sistema. El **lado derecho** de la V representa la integración de las piezas y su verificación. Es muy similar al modelo en cascada, ya que es muy rígido y contiene una gran cantidad de iteraciones.

Indistintamente del modelo que escojamos, deberemos seguir una serie de etapas:



Ventajas	Inconvenientes
Facilita la localización de fallos.	Las pruebas pueden llegar a ser costosas.
Modelo muy sencillo.	El cliente debe tener paciencia hasta el producto final.
El cliente está involucrado en las pruebas.	Pueden no estar bien definidos los requisitos del cliente.
Se recomienda para:	
<ul style="list-style-type: none"><li>• Ser aplicado en sistemas sencillos pero de confiabilidad alta (transacciones en bases de datos).</li></ul>	



### ponte a prueba

**¿Qué desventaja tiene el modelo en espiral?**

- a) Es un modelo muy rígido.
- b) Solo es útil para proyectos muy pequeños y con pocos cambios.
- c) Es difícil evaluar los riesgos.
- d) Necesita una especificación de muy completa de requisitos.

**Un cliente pide que se realice una base de datos de su web. ¿Qué modelo de desarrollo es el más adecuado?**

- a) Modelo en cascada con realimentación.
- b) Modelo en espiral.
- c) Modelo en V.
- d) Ninguno de los modelos señalados.

**En el modelo en V, las pruebas se representan en la parte derecha y en la parte izquierda, las especificaciones del sistema.**

- a) Verdadero.
- b) Falso.

#### 1.5.1. ANÁLISIS

Al realizar un proyecto, la parte más importante es entender qué se quiere realizar y analizar las posibles alternativas y soluciones. Por ello, es **fundamental** analizar los requisitos que el cliente ha solicitado.

Por tanto, el análisis consiste en la especificación de las características operativas del software, indica cuál es la interfaz que ha de desarrollarse y marca las restricciones de este.

Aunque pueda parecerlo, no es una tarea fácil, ya que a menudo el cliente es poco claro y durante el desarrollo pueden surgir nuevos requerimientos. Habrá que tener una buena comunicación entre el cliente y los desarrolladores para evitar futuros problemas. Para la obtención de estos requisitos, se usarán distintas técnicas:



- **Entrevistas:** técnica tradicional en la que hablamos con el cliente. En un ambiente más relajado (por ejemplo, compartiendo un café fuera de la oficina), el cliente tiene a expresarse con más claridad.
- **Desarrollo conjunto de aplicaciones (JAD, *join application design*):** entrevista de dinámica de grupo (talleres) en la que cada integrante aporta su conocimiento (usuarios, administradores, desarrolladores, analistas, etcétera).
- **Planificación conjunta de requisitos (JRP, *joint requirements planning*):** el objetivo de estas sesiones es involucrar a la dirección para obtener mejores resultados en el menor tiempo posible. La diferencia con el JAD es la participación del nivel más alto de la organización en la visión general del negocio (director, promotor, especialistas de alto nivel, entre otros).
- **Brainstorming:** reuniones en las que se intentan crear ideas desde distintos puntos de vista (tormenta de ideas). Idónea para el comienzo del proyecto.

- **Prototipos:** versión inicial del sistema en el que se puede ver el problema y sus posibles soluciones. Se puede desechar o usar para añadir más cosas.
- **Casos de uso:** “Los casos de uso simplemente son una ayuda para definir lo que existe fuera del sistema (actores) y lo que debe realizar el sistema (casos de uso)” (Ivar Jacobson, 2005). Este tipo de diagramas son fundamentales en la ingeniería de requisitos.

Podemos clasificar los requisitos en:

- **Requisitos funcionales:** nos describen al detalle la función que realiza el sistema, la reacción ante determinadas entradas y cómo se comporta en distintas situaciones.
- **Requisitos no funcionales:** son limitaciones sobre la funcionalidad que ofrece el sistema. Estos requerimientos se refieren a las propiedades emergentes del sistema, como la fiabilidad o la capacidad de almacenamiento. Incluyen restricciones impuestas por el estándar del aplicativo.

A la hora de representar estos requisitos, podemos hacerlo con distintos modelos:

- **Modelo basado en el escenario:** se basa en el punto de vista de los actores del sistema. Lo podemos representar con los casos de uso o las historias de usuario.

UC-NUM	LOGIN	
Versión	1.0 (dd/mm/yyyy)	
Autores		
Dependencias	<ul style="list-style-type: none"> <li>· <a href="#">[OBJ-0001] Objetivo Principal</a></li> <li>· <a href="#">[OBJ-0009] Gestión de usuarios</a></li> </ul>	
Descripción	El sistema deberá comportarse tal como se describe en el siguiente caso de uso cuando <i>el usuario invitado entra en la aplicación</i>	
Precondición	El usuario está registrado	
Secuencia normal	Paso	Acción
	1	El sistema pide nickname y password
	2	El actor <a href="#">Invitado (ACT-0001)</a> introduce los datos y selecciona “entrar”
	3	El sistema comprueba en la base de datos que los datos son válidos y el caso de uso finaliza
Postcondición	El usuario está dentro de la aplicación	
Excepciones	Paso	Acción
	3	Si los datos no son válidos, el sistema no permite el acceso, a continuación este caso de uso queda sin efecto

Ejemplo de caso de uso: registro de usuario.

- **Modelo de datos:** muestra el entorno y la información del problema. Lo podemos representar con el diccionario de datos.

- **Diccionario de datos (DD):** descripción detallada de los datos utilizados por el sistema que gráficamente están representados por los flujos de datos y almacenes presentes sobre el conjunto de DFD.

En esta primera fase de análisis, es fundamental que todo lo que se realice quede plasmado en el documento Especificación de Requisitos de Software (ERS). Debe ser un documento completo, sin ambigüedades, sencillo de usar a la hora de verificarlo, modificarlo o de identificar el origen y las consecuencias de los requisitos. Cabe destacar que nos servirá para la siguiente fase en el desarrollo.

### EJEMPLO

Denominación = tratamiento formal + nombre  
+ apellido

Tratamiento formal = [Don|Doña|Sr.|Sra.]

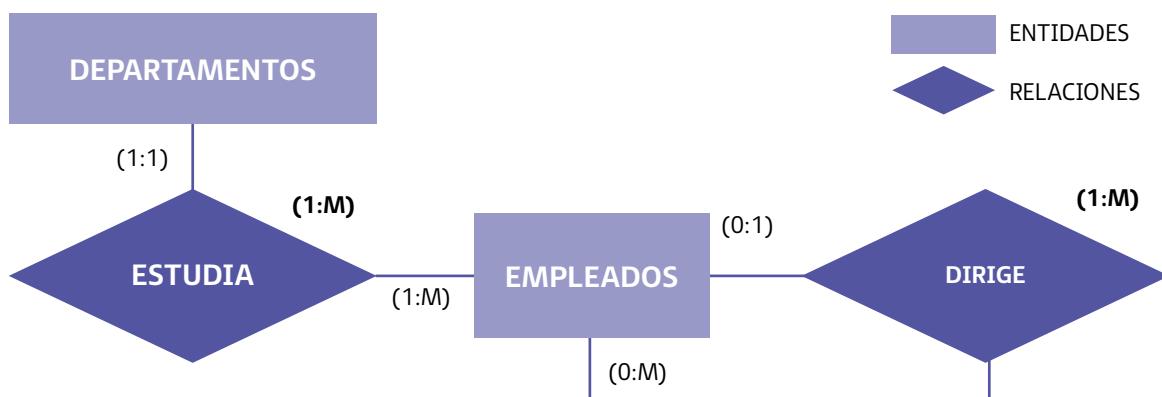
Nombre = {carácter}

Apellido = {carácter}

DNI = {carácter}

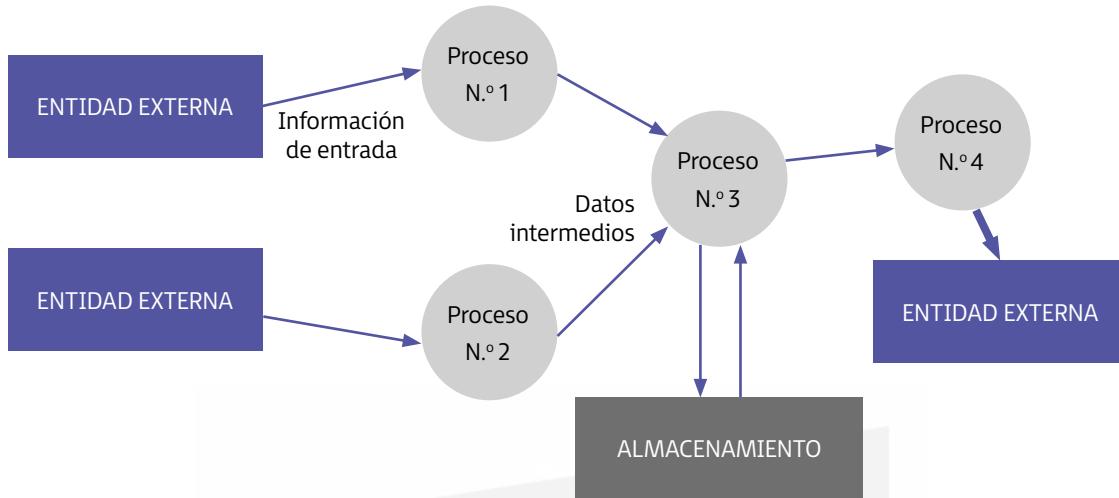
Carácter = [A-Z|a-z|0-9]

- **Diagrama entidad/relación (DER):** se usa para representar datos y sus relaciones. Representa los datos que se introducen, almacenan y transforman en el sistema.



- **Modelos orientados al flujo:** representan los elementos funcionales del sistema de tal forma que reflejan cómo se transforman los datos a medida que avanzan dentro del aplicativo.

- **Diagramas de flujo de datos (DFD):** nos va a representar el flujo de datos entre procesos, entidades externas (componentes que no son del sistema) y almacenes del sistema (datos desde el punto de vista estático):
  - **Procesos** → burbujas ovaladas o circulares.
  - **Entidades externas** → rectángulos.
  - **Almacenes** → dos líneas horizontales y paralelas.
  - **Flujo de datos** → flechas.



- **Diagramas de flujo de control (DFC):** similar al anterior, pero en vez de flujo de datos muestra el flujo de control. Un gran número de aplicaciones son causadas por eventos y no por datos, producen información de control y procesan información con mucha atención al tiempo y al rendimiento.
- **Diagramas de transición de estados (DTE):** representa el comportamiento del sistema dependiente del tiempo. Se aplican, sobre todo, en sistemas en tiempo real.

**ponte a prueba**

**En la fase de análisis, realizamos los diagramas de clases para modelar el sistema.**

- a) Verdadero
- b) Falso

**En la fase de análisis, capturamos los requisitos no funcionales.**

- a) Verdadero
- b) Falso



### 1.5.2. DISEÑO

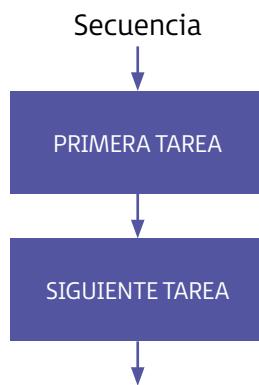
Fase de diseño  
[youtu.be/pn1PEOFhlec](https://youtu.be/pn1PEOFhlec)



Una vez que hemos identificado los requerimientos necesarios, ahora tendremos que componer la forma para solucionar el problema. Traduciremos los requisitos funcionales y los no funcionales en una representación de software. “Las preguntas acerca de si el diseño es necesario o digno de pagarse están más allá de la discusión: el diseño es inevitable. La alternativa al buen diseño es el mal diseño, no la falta de diseño” (Douglas Martin, 1994).

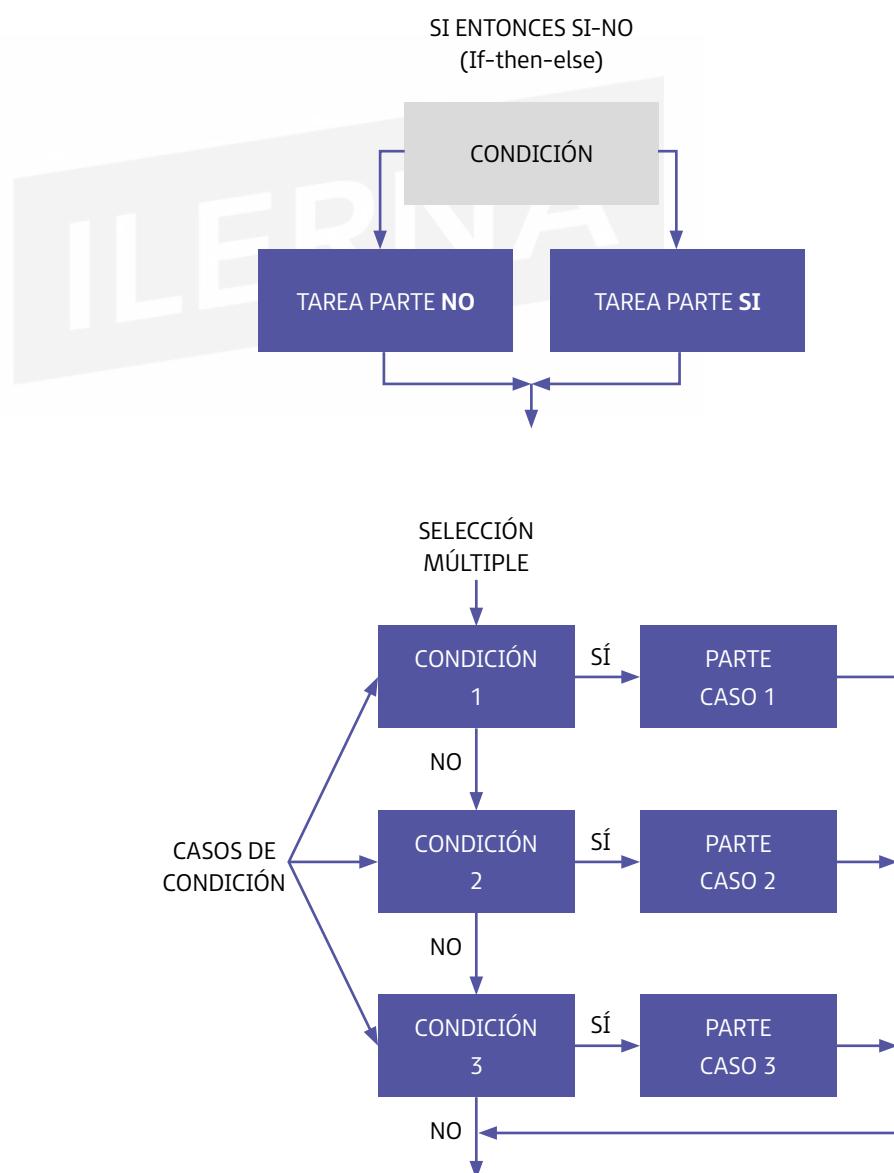
Existen dos tipos de diseño:

- **Diseño estructurado:** basado en el flujo de datos a través del sistema. Dentro de la programación estructurada, existen una serie de construcciones (bucles) que son fundamentales para el diseño a nivel de componentes. Estas construcciones son: secuencial, condicional y repetitiva:
  - **Construcción secuencial:** se refiere a la ejecución sentencia por sentencia de un código fuente, es decir, hasta que no termine la ejecución de una sentencia, no pasará a la siguiente.



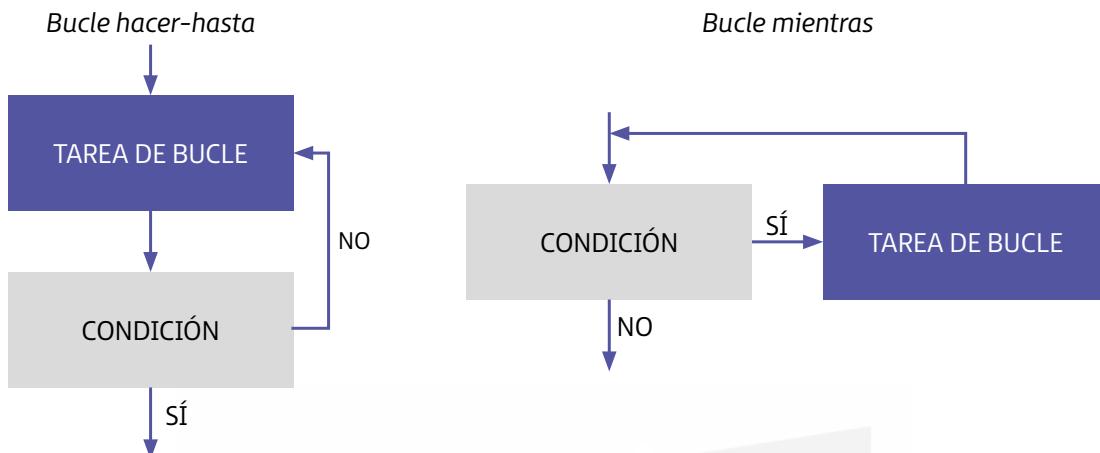
– **Construcción condicional:** selecciona un proceso u otro según una condición lógica (rombo). Si se cumple, se realiza la parte SI, si no, se realiza la parte NO.

La selección múltiple es una extensión de la estructura *Si entonces si-no*; un parámetro se prueba por continuas decisiones hasta que alguna es verdadera y ejecuta según ese camino.



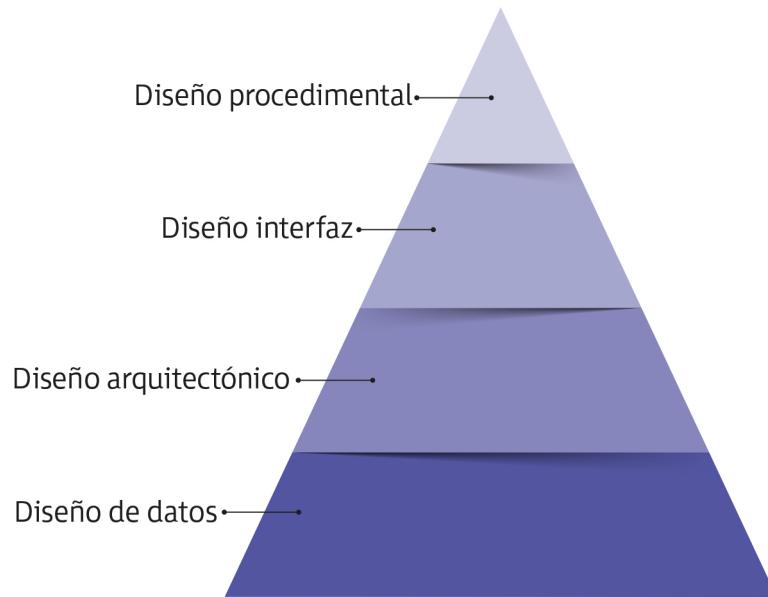
### – Construcción repetitiva:

- **Hacer hasta:** se ejecuta una primera vez la tarea y al finalizarla se comprueba la condición. Si esta no se cumple, se realiza de nuevo hasta que se cumpla la condición y finalice la tarea. Se realiza al menos una vez.
- **Mientras:** en este caso, se comprueba antes la condición y, después, se realiza la tarea continuamente siempre que se cumpla la condición. Se finaliza cuando la condición no se cumpla.



El diseño estructurado podemos dividirlo en:

- **Diseño de datos.** Transforma la información relativa al análisis en estructuras de datos para su posterior implementación mediante diferentes lenguajes de programación. Por ejemplo, esquemas lógicos de datos.
- **Diseño arquitectónico.** Es un esquema similar al plano de una casa. Se centra en la representación de la estructura de los componentes del software, sus propiedades y sus interacciones. Este diseño se basa en la información del entorno del aplicativo que realizar y de los modelados de requerimientos, como DFD o DFC.
- **Diseño de la interfaz.** Detalla la comunicación que realiza el software consigo mismo, los sistemas que operan con él y los usuarios. El resultado es la creación de formatos de pantalla. Los elementos importantes son la interfaz de usuario (UI) e interfaces externas con otros sistemas, dispositivos o redes.
- **Diseño a nivel de componentes (diseño procedimental).** Es similar a los planos de cada habitación de una casa. Convierte elementos estructurales de la arquitectura del software en una descripción procedural de los componentes del software. El resultado será el diseño de cada componente con el detalle necesario para que sirva de guía en la generación del código fuente. Se realiza mediante diagramas de flujo, diagramas de cajas, tablas de decisión, pseudocódigo, etcétera.



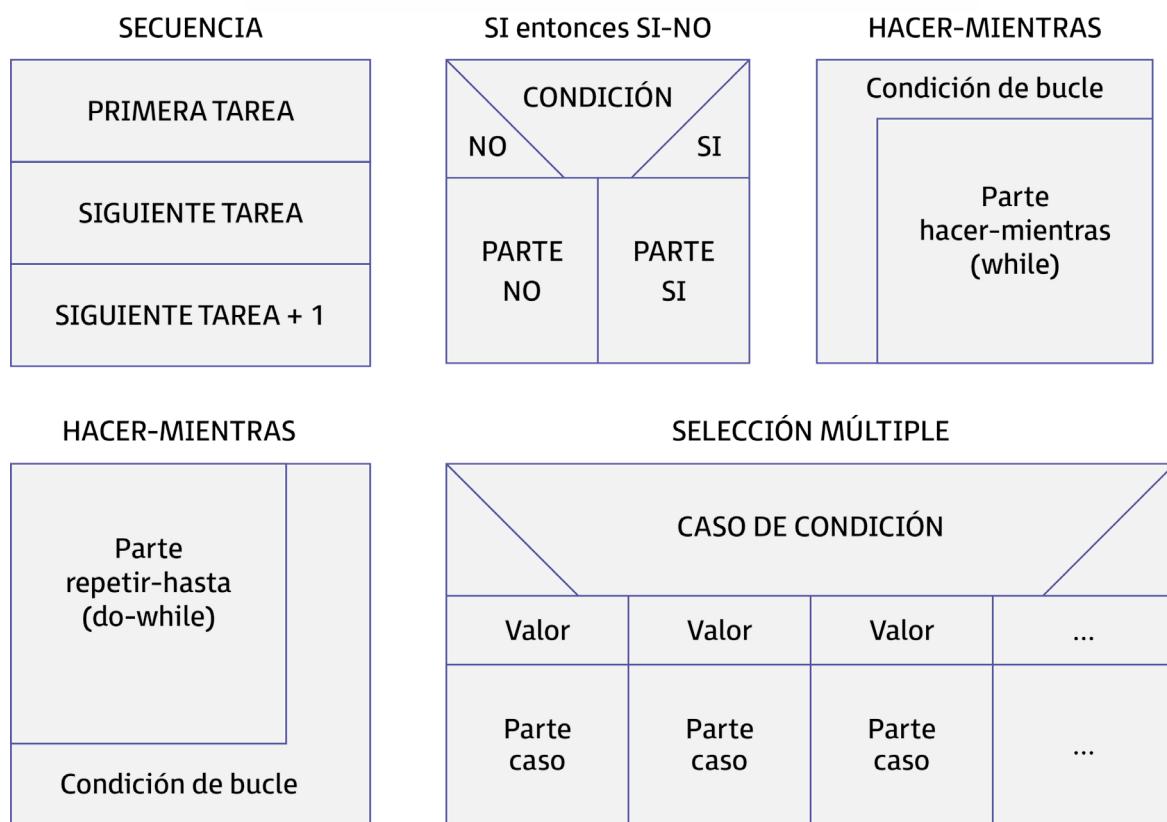
### Notaciones gráficas para el diseño procedural

Al representar el diseño, usaremos algunas herramientas básicas, como los diagramas de flujo, los diagramas de cajas, las tablas de decisión o el pseudocódigo.

- **Diagramas de flujo:** herramienta muy usada para el diseño procedural, donde:
  - **Caja:** paso del proceso.
  - **Rombo:** condición lógica.
  - **Flechas:** flujo de control.



- **Diagrama de cajas:** otra representación de nuestro diseño estructurado son los diagramas de cajas, en los que:
  - **Secuencia:** varias cajas seguidas.
  - **Condicional:** una caja para la parte SI y otra para la parte NO. Encima indicamos la condición.
  - **Repetitiva:** proceso que se repite, se encierra en una caja que se sitúa dentro de otra en la que indicamos la condición del bucle en la parte superior (*while*) o inferior (*do-while*).
  - **Selección múltiple:** la parte superior indica el caso de condición, mientras que en la parte inferior se definen tantas columnas como valores se quieran comprobar. Debajo de cada valor se indica la parte que se debe ejecutar.



- **Tablas de decisión:** nos permiten representar en una tabla las condiciones y las acciones que se llevarán a cabo al combinar esas condiciones. Proporcionan una notación que traduce las acciones y condiciones a una forma estructurada.

Se dividirá en dos:

- **Condiciones**: son las combinaciones posibles de nuestro sistema.
- **Acciones**: sentencias que se ejecutan cuando se cumplen determinadas condiciones.

	REGLAS				
CONDICIONES	1	2	3	4	n
Condición N. <sup>o</sup> 1	<input checked="" type="checkbox"/>			<input checked="" type="checkbox"/>	
Condición N. <sup>o</sup> 2		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		
Condición N. <sup>o</sup> 3	<input checked="" type="checkbox"/>			<input checked="" type="checkbox"/>	
ACCIONES					
Acción N. <sup>o</sup> 1	<input checked="" type="checkbox"/>			<input checked="" type="checkbox"/>	
Acción N. <sup>o</sup> 2	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>			
Acción N. <sup>o</sup> 3		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
Acción N. <sup>o</sup> 4			<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	

En las diferentes columnas podemos implementar las reglas que se deben cumplir para una determinada acción.

Al construir la tabla, seguimos los siguientes pasos:

1. Crear una lista con todas las acciones y todas las condiciones.
2. Relacionar los conjuntos con las acciones específicas, eliminando las combinaciones imposibles.
3. Determinar las reglas indicando la acción o acciones que ocurren para un conjunto de condiciones.

- **Pseudocódigo**: esta herramienta utiliza un texto descriptivo para crear el diseño de un algoritmo. Se asemeja al lenguaje de programación, ya que mezcla el lenguaje natural con la sintaxis de la programación estructurada, incluyendo palabras clave.

No existe un estándar definido y, al no ser un lenguaje de programación, no puede compilarse. La representación en pseudocódigo de las estructuras básicas de la programación estructurada es:

Secuencial	Instrucción 1 Instrucción 2 ... Instrucción n
Condicional	Si <Condición> Entonces <Instrucciones> Si no <Instrucciones> Fin Si
Condicional múltiple	Según sea <Variable> Hacer Caso valor 1: <Instrucciones> Caso valor 2: <Instrucciones> Caso valor 3: <Instrucciones> Otro caso: <Instrucciones> Fin Según
Hacer mientras	Hacer <Instrucciones> Mientras <Condición>
Mientras	Mientras <Condición> Hacer <Instrucciones> Fin Mientras

A continuación, vemos un ejemplo de pseudocódigo:

```

Inicio
    Abrir Archivo
    Leer Datos del Archivo
    Mientras no sea Fin de Archivo Hacer
        Procesar Datos Leido
        Leer Registro del Archivo
    Fin mientras
    Cerrar Archivo
Fin

```

- **Diseño orientado a objetos (DOO)**: conforme el diseño evoluciona, deberemos definir un conjunto de clases para afinar los detalles de nuestro sistema, lo que nos permitirá implementar una infraestructura que apoye nuestra solución del producto.

Podemos especificar cinco clases de diseño:

- Clases de interfaces: definimos todas las interacciones entre el usuario y la máquina.
- Clases de domino de negocio: identificamos las clases y los métodos que se necesitan para implementar elementos del dominio.
- Clases de proceso: implementación a bajo nivel para la gestión de las clases de domino de negocio.
- Clases de persistencia: definimos el almacenamiento de los datos (por ejemplo, una base de datos). Estas clases se mantendrán más allá de la ejecución de un determinado software.
- Clases de sistema: se definen funciones que permiten al sistema comunicarse con el exterior.

En el DOO utilizamos un **UML** (lenguaje de modelado unificado). Es un lenguaje de modelado basado en diagramas para expresar modelos anteriores.



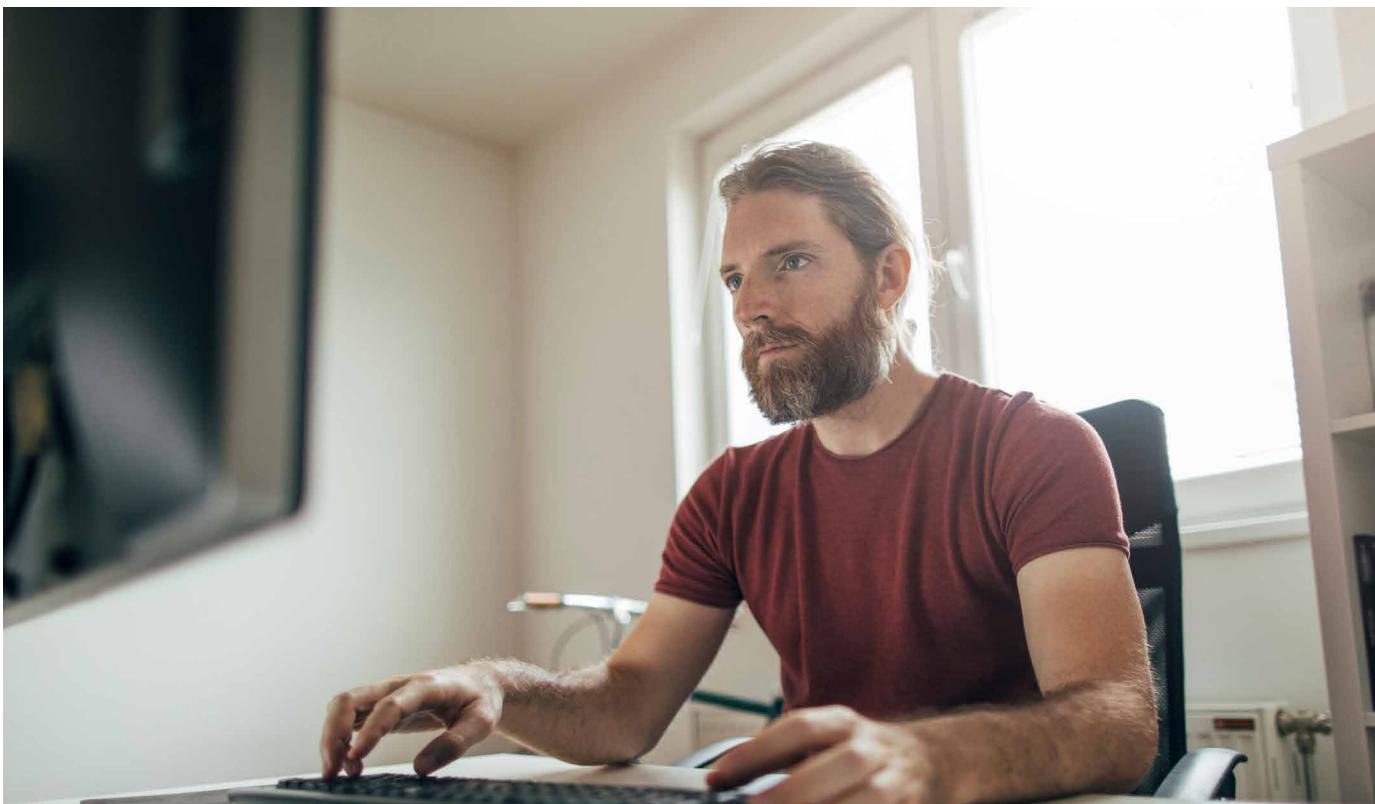
### ponte a prueba

#### ¿Qué es el pseudocódigo?

- a) Un diagrama que nos permite hacer un análisis de nuestro sistema.
- b) Un tipo de tabla de decisión.
- c) Una representación de nuestros algoritmos.
- d) Un tipo de prueba de caja negra.

#### ¿Cuál de las siguientes representaciones son utilizadas para la fase de diseño?

- a) Brainstorm.
- b) Diagramas de flujo.
- c) Tablas de decisión.
- d) B y C son correctas.



### 1.5.3. CODIFICACIÓN

**Fase de implementación**  
[youtu.be/xH7fApKpR9o](https://youtu.be/xH7fApKpR9o)



La tercera fase, una vez realizado el diseño, consistirá en el proceso de codificación. Aquí el programador se encarga de recibir los datos del diseño y transformarlo en lenguaje de programación. A estas instrucciones las llamaremos **código fuente**.

En cualquier proyecto en que se trabaje con un grupo de personas, habrá que tener unas normas de codificación y estilo que sean **sencillas, claras y homogéneas**, las cuales nos facilitará la corrección en caso de que sea otra persona la que lo ha realizado.

A continuación, veremos algunas series de normas en código **Java**:

- **Nombre de ficheros:** los archivos de código fuente tendrán como extensión `.java`, y los archivos compilados `.class`.
- **Organización de ficheros:** cada archivo deberá tener una clase pública y podrá tener otras clases privadas e interfaces que irán definidas después de la pública y estarán asociadas a esta. El archivo se dividirá en varias secciones:

– **Comentarios:** cada archivo debe empezar con un comentario en el que se indique el nombre de la clase, la información de la versión, la fecha y el aviso de derechos de autor.

– **Sentencias de tipo package e import:** se sitúan después de los comentarios en este orden: primero la sentencia *package* y después la de *import*.

– **Declaraciones de clases e interfaces:** consta de las siguientes partes:

- Comentario de documentación (*/\*\*...\*/*) acerca de la clases o interfaz.
- Sentencia tipo *class* o interfaz.
- Comentario de la implementación (*/\*...\*/*) de la clase o interfaz.
- Variables estáticas, en este orden: públicas, protegidas y privadas.
- Variables de instancia en este orden: públicas, protegidas y privadas.
- Constructores.
- Métodos.

• **Indentación:**

– Se usarán cuatro espacios (como recomendación) como unidad de indentación.

– Longitud de líneas de código no superior a 80 líneas.

– Longitud de líneas de comentarios no superior a 70 líneas.

– Si una expresión no cabe en una sola línea, se deberá romper antes de una coma o un operador y se alinearán al principio de la anterior.

• **Comentarios:** contendrán solo información que sea relevante para la lectura y comprensión del programa. Habrá dos tipos: de documentación y de implementación.

Los primeros describen la especificación del código como las clases Java, interfaces, constructores, métodos y campos. Se situarán antes de la declaración. La herramienta **Javadoc** genera páginas HTML partiendo de este tipo de comentarios. Un ejemplo sería:

```
/*
 * Esta clase Prueba nos proporciona...
 */
public class Prueba (...)
```

Los comentarios de implementación sirven para hacer algún comentario sobre la aplicación en particular. Pueden ser de tres tipos:

– De bloque:

```
/*
 * Esto es un comentario de bloque
 */
```

– De línea:

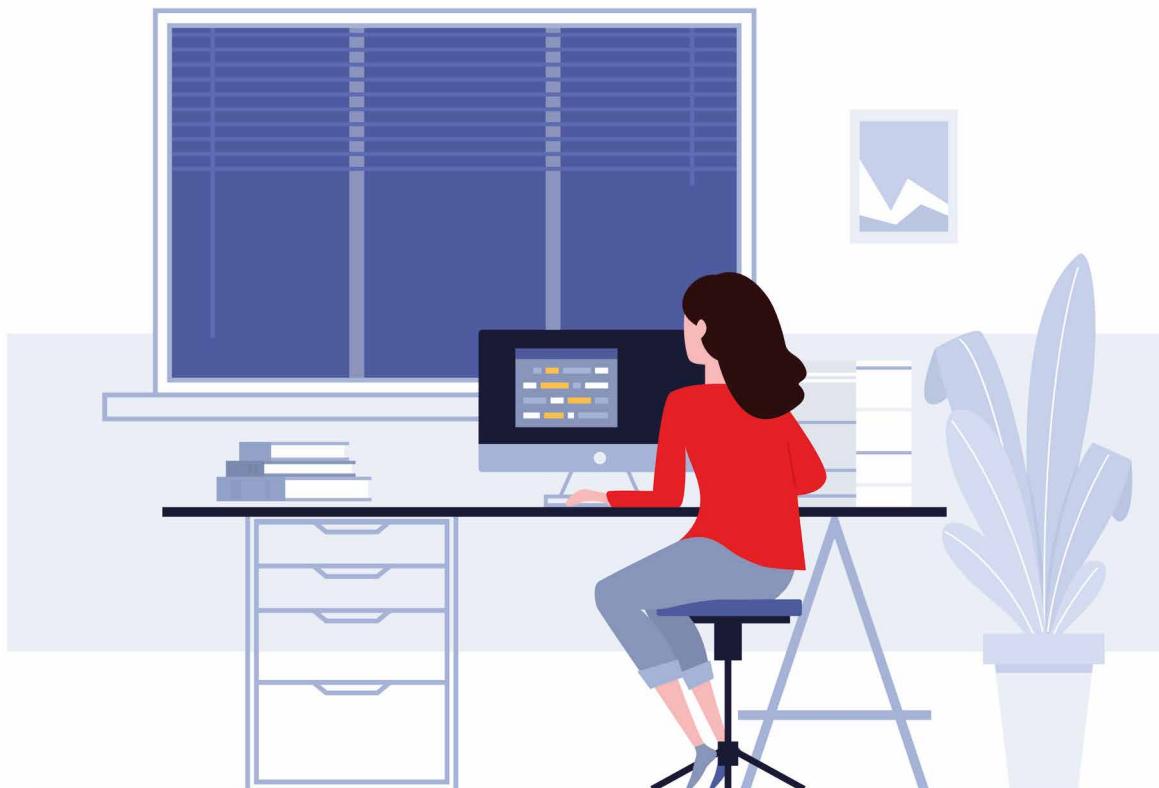
```
/* Comentario de linea */
```

– Corto:

```
// Comentario corto
```

- **Declaraciones:**

- Declarar una variable por línea.
- Inicializar una variable local al comienzo, donde se declara, y situarla al comienzo del bloque.
- En clases o interfaces:
  - No poner espacios en blanco entre el nombre del método y el paréntesis “(“.
  - Llave de apertura “{“, situarla en la misma línea que el nombre del método.
  - Llave de cierre “}”, situarla en una línea aparte y en la misma columna que el inicio del método. Excepto cuando esté vacío.
  - Métodos separados por una línea en blanco.





- **Sentencias:**

- Cada línea contendrá una sentencia.
- Si es un bloque, debe estar sangrado con respecto a lo anterior y entre llaves, aunque solo tenga una sentencia.
- Sentencias *if-else*, *if else-if else*. Nos definen bloques y tendrán todos los mismos niveles de sangrado.
- Bucles. Tendrán las normas anteriores y, si están vacío, no irán entre llaves.
- Sentencias *return* no irán entre paréntesis.

- **Separaciones:** hacen más legible el código. Se utilizarán (como recomendación):

- **Dos líneas en blanco** entre definiciones de clases e interfaces.
- **Una línea en blanco** entre métodos, definición de variables locales y la primera instrucción, antes de un comentario, entre secciones lógicas dentro de un método.
- **Un carácter en blanco** entre una palabra y un paréntesis, después de una coma, de operadores binarios menos el punto, de expresiones del *for* y entre un *cast* y la variable.

- **Nombres:** los nombres de las variables, métodos, clases, etcétera, hacen que los programas sean más fáciles a la hora de leerlos. Las normas que hay que seguir para asignar nombres son:

- **Paquetes:** se escriben en minúscula. Se podrán utilizar puntos para algún tipo de organización jerárquica, por ejemplo, *java.io*.
- **Clases e interfaces:** deben ser sustantivas o descriptivas, según lo que estemos creando. Si están compuestas por varias palabras, la primera letra de cada palabra irá en mayúscula.

- **Métodos:** se usarán verbos en infinitivo. Si están formados por varias palabras, el verbo estará en minúscula y la siguiente palabra empezará con mayúscula.
- **Variables:** deben ser cortas y significativas. Si están formadas por varias palabras, la primera debe ir en minúscula.
- **Constantes:** el nombre debe ser descriptivo. Será totalmente escrito en mayúscula y, si son varias palabras, separadas por un carácter de subrayado.

Vamos a ver un ejemplo sencillo en Java:

1. Creamos la clase *Persona* con sus atributos privados y su instancia.

```
package Ejemplo;

/**
 * Clase utilizada para crear nuevas personas
 * Pueden contener una edad y un nombre
 * @autor Martin Rivero
 * @version 1.0
 */
public class Persona {
    /**
     * Variable privada para guardar el nombre de la Persona
     */
    private String nombre;
    /**
     * Variable privada para guardar la edad de la Persona
     */
    private int edad;

    /**
     * Instancia una nueva persona sin nombre y sin edad.
     */
    public Persona() {
    }

    /**
     * Instancia una nueva PErsona.
     *
     * @param nombre el nombre
     * @param edad la edad
     */
    public Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }
}
```

2. Posteriormente, creamos los *getters* y *setters*.

```

    /**
     * Recupera el nombre.
     *
     * @return nombre
     */
    public String getNombre() {
        return nombre;
    }

    /**
     * Establece el nombre.
     *
     * @param nombre Nombre de la Persona
     */
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    /**
     * Recupera la edad.
     *
     * @return edad
     */
    public int getEdad() {
        return edad;
    }

    /**
     * Establece la edad.
     *
     * @param edad Edad de la Persona
     */
    public void setEdad(int edad) {
        this.edad = edad;
    }

```

3. Por último, en el código, creamos un método de tipo *string* para recuperar el nombre y la edad de la persona.

```

    /**
     * Recupera la Persona en formato String..
     *
     * @return string de Persona
     */
    @Override
    public String toString() {
        return "persona{" +
            "nombre=" + nombre + '\'' +
            ", edad=" + edad +
            '}';
    }

```

Cuando hemos terminado de escribir el código, lo tendremos que traducir al lenguaje máquina a través de **compiladores** o **intérpretes**. El resultado será el código objeto, aunque este no será todavía ejecutable hasta que lo enlazemos con las librerías para obtener así el **código ejecutable**. Una vez obtenido este código, tendremos que comprobar el programa para ver si cumple con nuestras especificaciones en el diseño.

Junto con el desarrollo del código, deberemos escribir **manuales** técnicos y de referencia, así como la parte inicial del manual de usuario. Estas partes serán esenciales para la etapa de prueba y mantenimiento, así como para la entrega del producto.



### ponte a prueba

#### **¿Cómo debe comenzar un archivo en java?**

- a) Sentencias tipo package.
- b) Interfaces.
- c) Nombre de la clase, la información de la versión, la fecha y el aviso de derechos de autor.
- d) Constructores.

**Es recomendable que las clases tengan una longitud de líneas de código superior a 80 caracteres.**

- a) Verdadero.
- b) Falso.

### 1.5.4. PRUEBAS

Al iniciar la etapa de pruebas, ya contaremos con el software, por lo que trataremos de encontrar errores en la codificación, en la especificación o en el diseño. Durante esta fase, se realizan las siguientes tareas:

- **Verificación:** probar que el software cumple con los requerimientos. Esto quiere decir que, en el documento de requerimientos, debe haber, como mínimo, una prueba por cada uno de ellos.
- **Validación:** encontrar alguna situación donde el software sea incorrecto o no cumple con las especificaciones.



Fase de pruebas

[youtu.be/HZmVWu7psbE](https://youtu.be/HZmVWu7psbE)

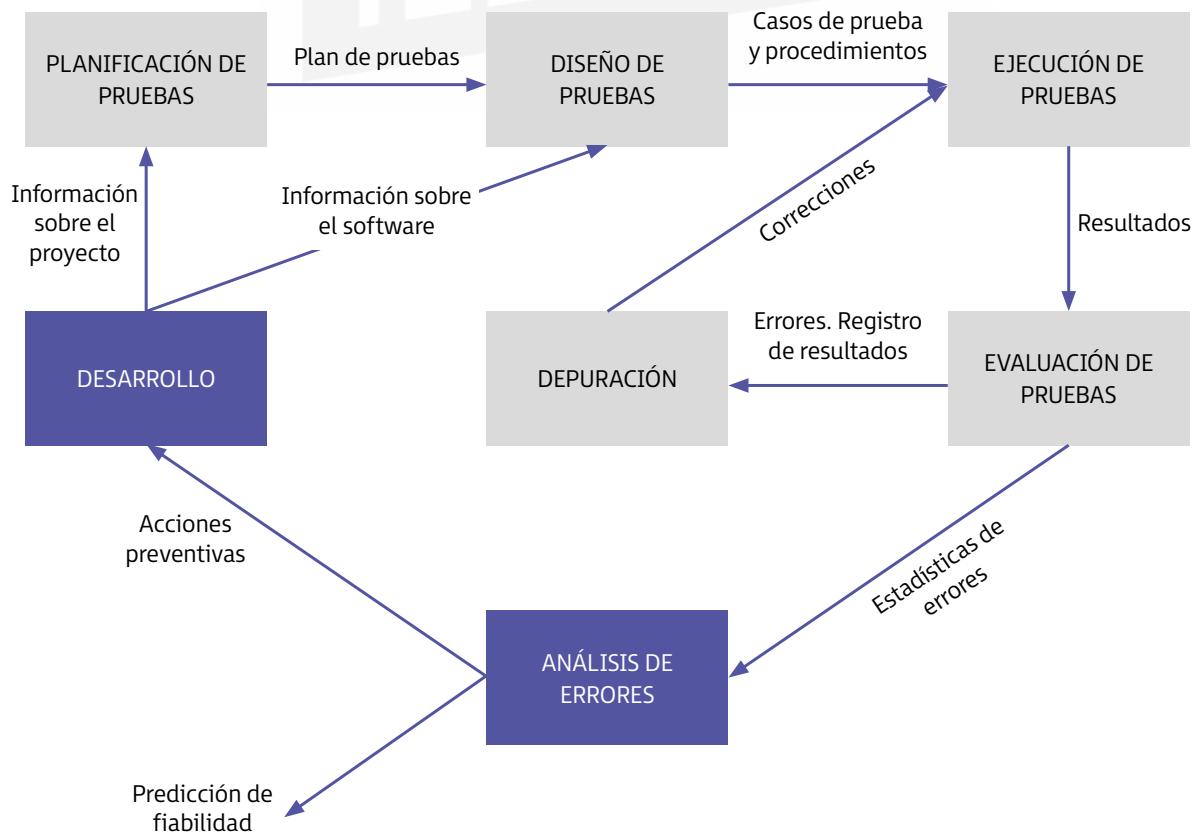


En esta etapa trataremos de comprobar los distintos tipos de errores, haciéndolo en el menor tiempo y con el menor esfuerzo posibles. Una prueba tendrá éxito si encontramos algún error no detectado anteriormente.

Las recomendaciones para llevar a cabo las pruebas son las siguientes:

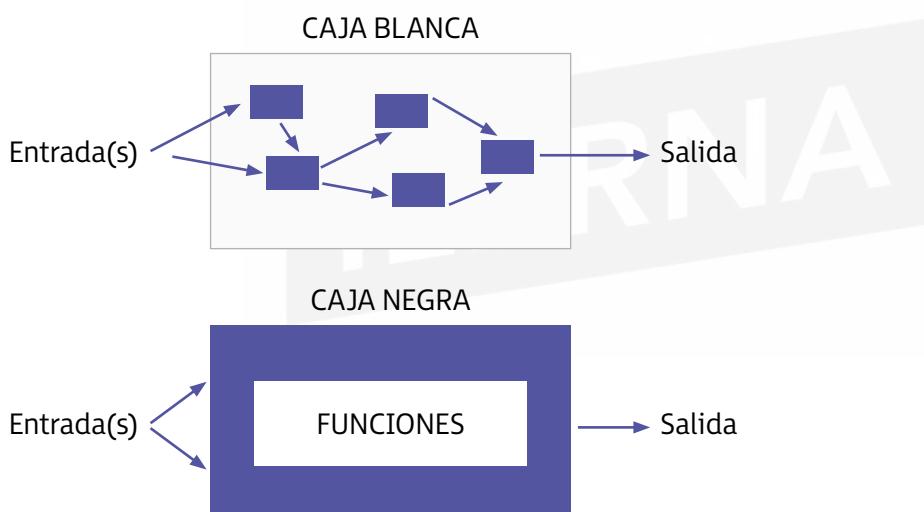
- Cada prueba definirá los resultados de la salida esperados.
- Evitar que el programador pruebe sus propios programas.
- Comprobar cada resultado en profundidad.
- Incluir todo tipos de datos, tanto válidos y esperados como inválidos e inesperados.
- Comprobar que el software hace lo que debe y lo que no debe hacer.
- No hacer pruebas que no estén documentadas.
- No suponer que en las pruebas no se van a cometer errores.
- Cuantas más pruebas se realicen, mayor es la probabilidad de encontrar errores y, una vez solucionados, tendremos una capacidad mayor de poder perfeccionar nuestro sistema.

El flujo del proceso a la hora de probar el software es el siguiente:



1. Primero **generamos un plan de pruebas** a partir de la documentación del proyecto y de la documentación del software que debemos probar.
2. Después se **diseñan las pruebas**: qué técnicas vamos a utilizar.
3. **Ejecución de las pruebas.**
4. **Evaluación**: identificación de posibles errores.
5. **Depuración**: localizar y corregir errores. Testar de nuevo tras encontrarse un error para verificar que se ha corregido.
6. **Análisis de errores**: analizar la fiabilidad del software y mejorar los procesos de desarrollo.

Para la realización del diseño de prueba se usan dos técnicas: **prueba de caja blanca** y **prueba de caja negra**. La primera valida la estructura interna del sistema, y la segunda, los requisitos funcionales sin observar el funcionamiento interno del programa. No son pruebas excluyentes y podemos combinarlas para descubrir distintos tipos de error.



**ponte a prueba**

**Estamos realizando las pruebas de un método que realiza el factorial de un número. Estamos introduciendo el número 4 y nos da como salida 24. ¿Qué pruebas estamos llevando a cabo?**

- a) Prueba de caja blanca
- b) Prueba de caja negra
- c) Pruebas de integración del sistema
- d) Pruebas de seguridad

### 1.5.5. DOCUMENTACIÓN

Cada etapa del desarrollo tiene que quedar perfectamente documentada. Para ello, necesitaremos reunir los documentos generados y hacer una clasificación según el nivel técnico de sus descripciones.

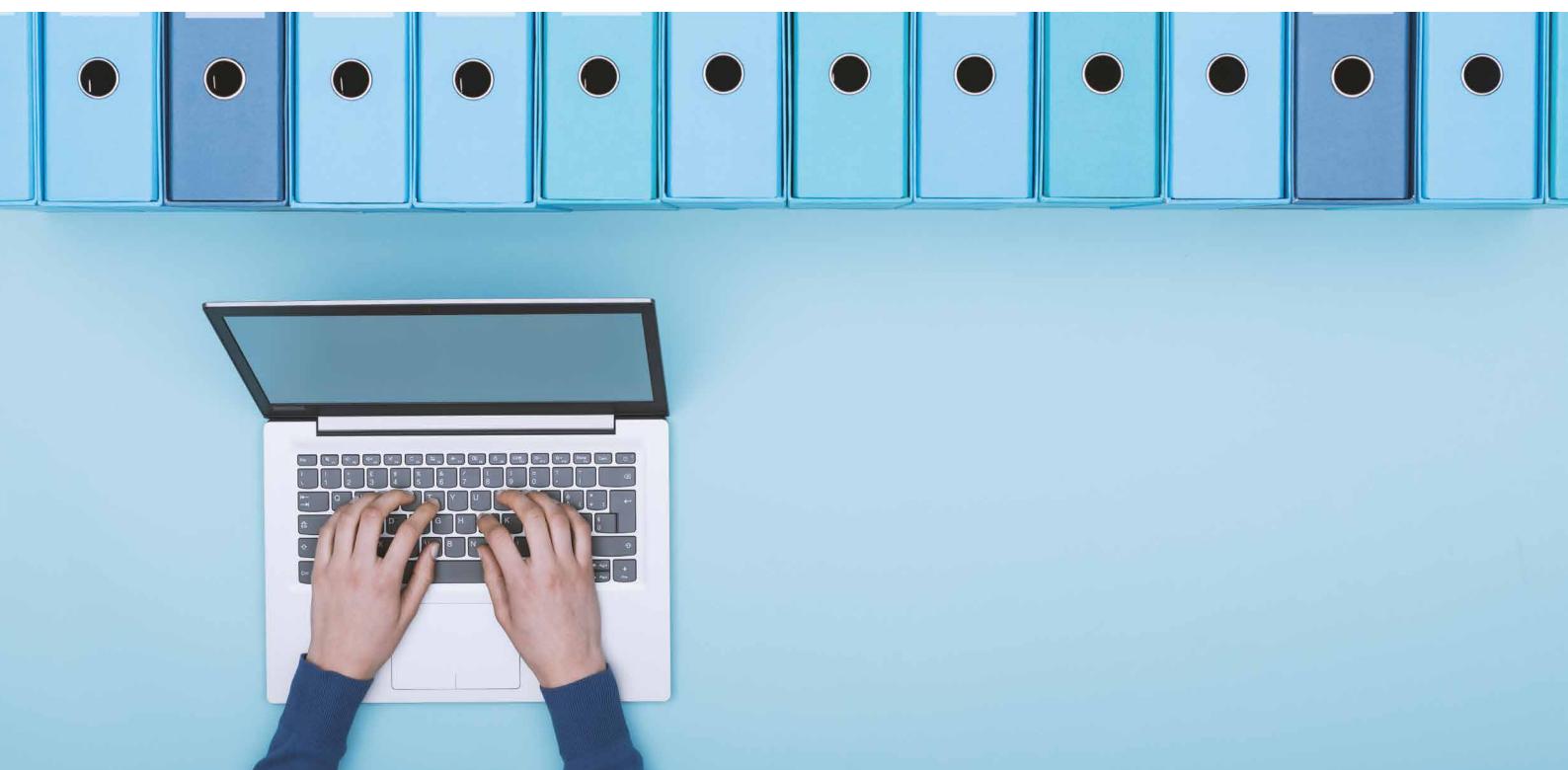
Estos documentos:

- Deben actuar como medio de comunicación para que los miembros del equipo se puedan comunicar entre sí.
- Deben ser un almacén de información del sistema para poder ser utilizado por personal de mantenimiento.
- Proporcionan información para facilitar la planificación de gestión del presupuesto y programar el proceso de desarrollo del software.
- Algunos documentos deben especificar al usuario cómo debe usar y administrar el sistema.

La documentación se puede dividir en dos clases:

1. **Documentación del proceso.** En estos documentos se registra el proceso de desarrollo y mantenimiento. Así, se incluyen planes, estimaciones y horarios que se usan para predecir y controlar el proceso de software.
2. **Documentación del producto.** Este documento describe el producto que está siendo desarrollado e incluye la documentación del sistema y la documentación del usuario.

Mientras que la documentación del proceso se usa para gestionar todo el proceso de desarrollo del software, la documentación del producto se usará una vez que el sistema ya esté funcionando.



## Documentación del usuario

Hay dos tipos de documentación:

- **Orientada a usuarios finales:** describe la funcionalidad del sistema con el objetivo de que el usuario pueda interactuar con este.
- **Orientada a administradores del sistema:** describe la gestión de los programas utilizados por los usuarios finales. Como administradores del sistema, podemos encontrar gestores de redes o técnicos especializados en resolver problemas a los usuarios finales.

Al existir distintos tipos de usuario, tendremos que entregar un tipo de documento a cada uno. En el siguiente esquema se muestran los diferentes documentos:



<b>Manual introductorio</b>	Destinado a usuarios noveles.	Explicaciones sencillas de cómo empezar a usar el sistema. Solución de errores.
<b>Manual de referencia del sistema</b>	Destinado a usuarios experimentados.	Descripción detallada del sistema y lista completa de errores.
<b>Guía del administrador del sistema</b>	Destinado a administradores.	Para sistemas en los que hay comandos, describir las tareas, los mensajes producidos y las respuestas del operador.

## Documentación del sistema

Serán los documentos que describen el sistema, desde la especificación de los requisitos hasta las pruebas de aceptación, y serán esenciales para entender y mantener el software.

Deberán incluir:

<b>Fundamentos del sistema</b>	Se describen los objetivos del sistema.
<b>El análisis y especificación de requisitos</b>	Información exacta de los requisitos.
<b>Diseño</b>	Se describe la arquitectura del sistema.
<b>Implementación</b>	Descripción de la forma en que se expresa el sistema y acciones del programa en forma de comentarios.
<b>Plan de pruebas del sistema</b>	Evaluación individual de las unidades del sistema y las pruebas que se realizan.
<b>Plan de pruebas de aceptación</b>	Descripción de pruebas que el sistema debe pasar antes de que los usuarios las acepten.
<b>Los diccionarios de datos</b>	Descripciones de los términos que se relacionan con el software en cuestión.

En ocasiones, cuando los sistemas son más pequeños, se obvia la documentación.

Es necesario, como mínimo, incluir la especificación del sistema, el documento de diseño arquitectónico y el código fuente del programa.

El mantenimiento de esta documentación muchas veces se descuida y se deja al usuario desprotegido ante cualquier problema sobre el manejo y errores de la aplicación.

### Estructura del documento

El documento debe tener una organización para que, a la hora de consultarla, se localice fácilmente la información, por lo que tendrá que estar dividido en capítulos, secciones y subsecciones. Algunas pautas son las siguientes:

- Deben tener una portada, tipo de documento, autor, fecha de creación, versión, revisores, destinatarios del documento y la clase de confidencialidad del documento.
- Debe poseer un índice con capítulos, secciones y subsecciones.
- Incluir al final un glosario de términos.

Según el estándar IEEE std 1063-2001, la documentación de usuario tiene que estar formada por los siguientes elementos:

Componente	Descripción
<b>Datos de identificación</b>	Título e identificación.
<b>Tabla de contenidos</b>	Capítulo, nombre de sección y número de página. Es obligatoria en documentos de más de ocho páginas.
<b>Lista de ilustraciones</b>	Números de figura y títulos (optativo).
<b>Introducción</b>	Propósito del documento y breve resumen.
<b>Información para el uso de la documentación</b>	Sugerencia sobre cómo usar la documentación de forma eficaz.
<b>Conceptos de las operaciones</b>	Explicación del uso del software.
<b>Procedimientos</b>	Instrucciones sobre cómo utilizar el software.
<b>Información sobre los comandos software</b>	Descripción de cada uno de los comandos del software.
<b>Mensajes de error y resolución de problemas</b>	Descripción de los mensajes de error y los tipos de resolución.
<b>Glosario</b>	Definiciones de términos especiales.
<b>Fuentes de información relacionadas</b>	Enlaces a otros documentos para proporcionar información adicional.
<b>Características de navegación</b>	Permite encontrar su ubicación actual y moverse por el documento.
<b>Índice</b>	Lista de términos clave y páginas a las que estos hacen referencia.
<b>Capacidad de búsqueda</b>	Forma de buscar términos específicos (en documentos electrónicos).



### 1.5.6. EXPLOTACIÓN

En esta etapa se lleva a cabo la instalación y puesta en marcha de producto. Se deberán indicar las tareas programadas (como la gestión de *backups*), la monitorización o la gestión de la capacidad. Se llevarán a cabo las siguientes tareas:

- **Gestión de backups:** descripción detallada de la política de *backups* del sistema. Se incluirá cada cuánto tiempo se realiza un respaldo, un almacenamiento de datos y la gestión de versiones del aplicativo.
- **Carga y descarga de datos:** se llevará un control de la carga y descarga masiva de datos y se detallarán las situaciones por las cuales se han llevado a cabo.
- **Monitorización y mapeo del aplicativo.**
- **Estrategia para la implementación del proceso:** se definen procedimientos para recibir, registrar, solucionar y hacer un seguimiento de los problemas y probar el producto.
- **Soporte al usuario:** dar asistencia y consultoría al usuario. Las peticiones y acciones que se hagan deberán registrarse y supervisarse.

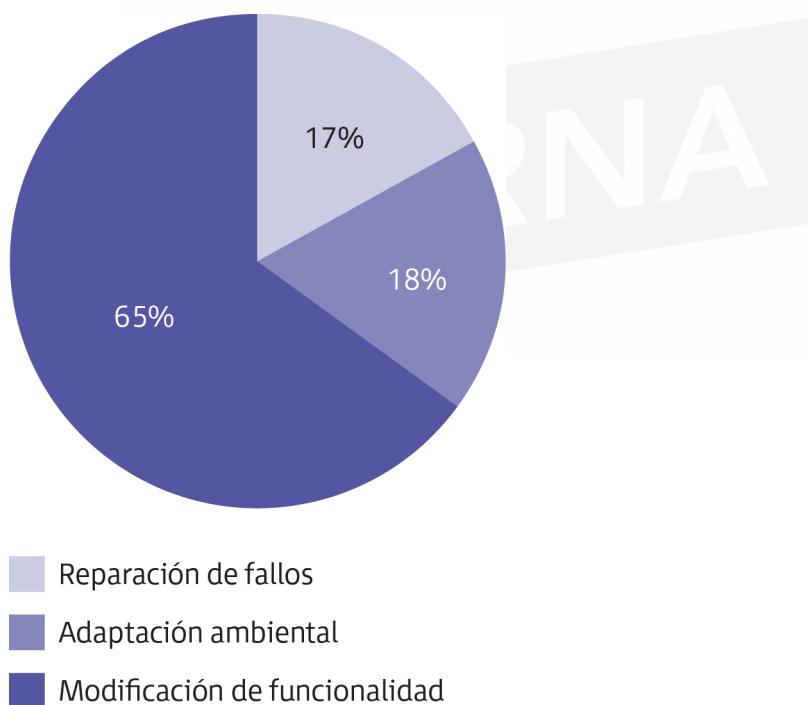
## 1.5.7. MANTENIMIENTO

La fase de mantenimiento consiste en la modificación del producto software después de la entrega al usuario/cliente para corregir fallos, mejorar el rendimiento o adaptar el producto a un entorno modificado.

Existen tres tipos de mantenimiento:

- **Recuperación de fallos:** los errores de codificación son los menos costosos. Los errores de diseño son más caros porque pueden verse afectados muchos componentes del software.
- **Adaptación ambiental:** mantenimiento respecto del entorno del sistema, como el hardware, el sistema operativo, etcétera.
- **Incremento de funcionalidad:** varían los requisitos del sistema debido, por ejemplo, a un cambio empresarial o en la organización.

La distribución del esfuerzo de mantenimiento es el siguiente:



En general, el coste de agregar una nueva funcionalidad después de haber realizado el aplicativo es mayor que añadir esta funcionalidad durante el desarrollo de este. Esto se debe sobre todo a:

- **La estabilidad del equipo:** una vez que el proyecto finaliza, por término general, el equipo de trabajo se separa y se encarga de nuevos proyectos. El nuevo equipo de mantenimiento debe emplear un tiempo en conocer la estructura y los componentes del sistema.

- **Diferencia de contratos:** generalmente, el desarrollo y el mantenimiento suelen ir en contratos diferentes, incluso el mantenimiento puede ser encargado a una compañía diferente. Por tanto, ese equipo de mantenimiento apenas cuenta con estímulos suficientes.
- **Habilidades del equipo de mantenimiento:** por término general, el equipo de mantenimiento suele ser más inexperto y no está familiarizado con los entornos de desarrollo del sistema. De hecho, pueden existir sistemas antiguos de lenguajes de programación obsoletos (como Basic o sistemas MS-DOS) en los que no se tenga experiencia.
- **Antigüedad del programa:** según se van realizando cambios, la estructura del programa tiende a degradarse. Los programas van envejeciendo y resultan más complicados de escalar y mantener. La documentación puede estar obsoleta si no ha sido actualizada o los programas se desarrollaron con técnicas de programación ya extintas.

Por tanto, intentar predecir el número de solicitudes de cambio para un sistema requiere un sobrecoste de entendimiento de la relación entre el sistema y su relación con el exterior.



Con base en la experiencia, podemos intentar predecir:

- ¿Cuántas peticiones cambio suelen producirse?
- ¿Qué partes del sistema suelen ser las más afectadas por los cambios?
- ¿Qué es lo más costoso en el mantenimiento?
- ¿Cómo se distribuye el coste en el sistema?
- ¿Cómo evolucionan esos costes a lo largo de vida del proyecto?

## 1.6. METODOLOGÍAS ÁGILES

**Metodologías ágiles**  
<https://youtu.be/41fgLS4i-5Y>



Las metodologías ágiles son métodos de gestión que permiten adaptar la forma de trabajo al contexto y naturaleza de un proyecto, basándose en la flexibilidad y la inmediatez y teniendo en cuenta las exigencias del mercado y de los clientes. Los pilares fundamentales de las metodologías ágiles son el trabajo colaborativo y en equipo.

Trabajar con metodologías ágiles conlleva las siguientes ventajas:

- Ahorrar tanto en tiempo como en costes (son baratas y más rápidas).
- Mejorar la satisfacción del cliente.
- Mejorar la motivación e implicación del equipo de desarrollo.
- Mejorar la calidad del producto.
- Eliminar aquellas características innecesarias del producto.
- Alertar rápidamente tanto de errores como de problemas.

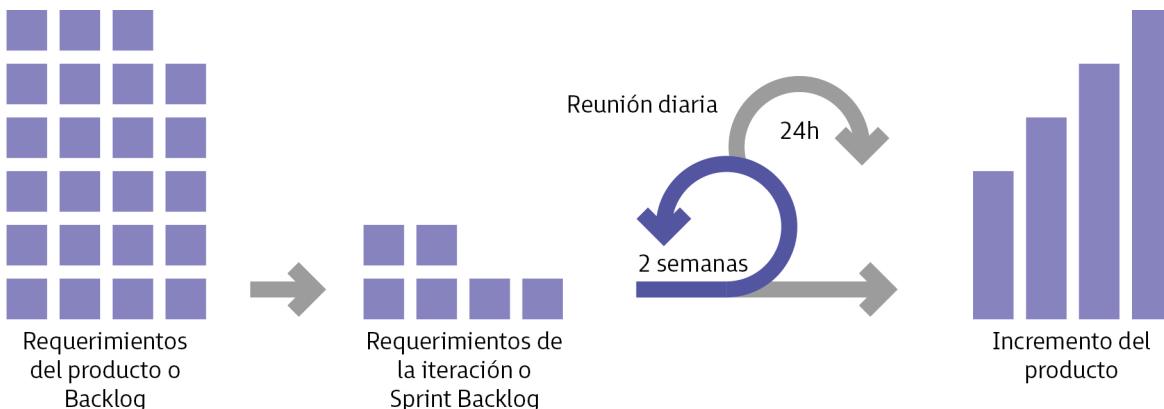
Aunque hay muchas ventajas a la hora de aplicar estas metodologías, estos principios son difíciles a veces de cumplir:

- Aunque la idea de que el cliente esté involucrado en todo el proceso es un incentivo positivo, los clientes tienen que ocuparse de otros procesos de gestión y están sometidos a presiones muy diferentes.
- La idea de que el equipo de trabajo esté cohesionado es, en ocasiones, ideal. No todos los componentes del equipo tienen la misma personalidad ni el mismo compromiso.
- Los cambios son complicados con demasiados participantes.
- Mantener esta línea de trabajo a veces es complicado por razones externas: plazos de entrega, cada componente del equipo trabaja a un ritmo, etcétera.
- Las grandes compañías están sujetas a muchos cambios a lo largo del tiempo, por lo que muchos métodos de trabajo se pueden ver modificados y pueden existir muchas alteraciones en las metodologías.

En este punto, vamos a destacar cuatro metodologías:

## SCRUM

El enfoque de esta metodología se basa en un trabajo iterativo.



Existen tres fases de esta metodología:

- **Planificación:** donde se establecen los objetivos generales del proyecto y cómo será la arquitectura.
- **Ciclos (sprints):** en cada uno de estos ciclos se desarrolla un incremento o iteración.
- **Documentación:** donde se desarrolla la ayuda del sistema y los manuales de usuario.

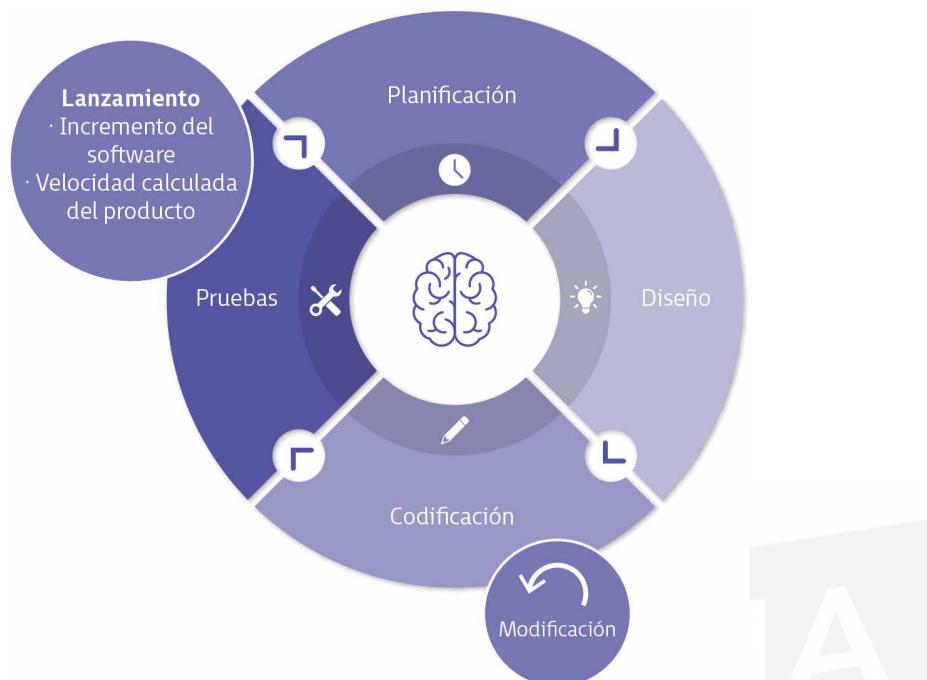
Por tanto, aporta una estrategia de desarrollo incremental en lugar de la planificación y ejecución completa del producto. La calidad del resultado se basa principalmente en el conocimiento innato de las personas en equipos autoorganizados antes que en la calidad de los procesos empleados.

### Características específicas de SCRUM:

- Una de las bases de las metodologías ágiles es el ciclo de vida iterativo e incremental. El ciclo de vida iterativo o incremental es aquel en el que se va liberando el producto por partes, periódicamente, iterativamente, poco a poco y, además, cada entrega es el incremento de funcionalidad respecto de la anterior. Cada periodo de entrega es un *sprint*. Estos *sprints* tienen una longitud fija de entre dos y cuatro semanas.
- Reunión diaria. Máximo, 15 minutos. Se trata de ver qué se hizo ayer, qué se va a hacer hoy y qué problemas se han encontrado.
- Reunión de revisiones del *sprint*. Al final de cada *sprint*, se trata de analizar qué se ha completado y qué no.
- Retrospectiva del *sprint*. También se realiza al final de *sprint*, y sirve para que los implicados den sus impresiones sobre *sprint*. Se utiliza para la mejora del proceso.

- Durante la fase de cómo va a ser el *sprint*, se asignan las prioridades del proyecto y riesgos de este.
- A diferencia de XP, SCRUM no hace indicaciones concretas sobre cómo detallar los requerimientos.

## Programación extrema (XP)

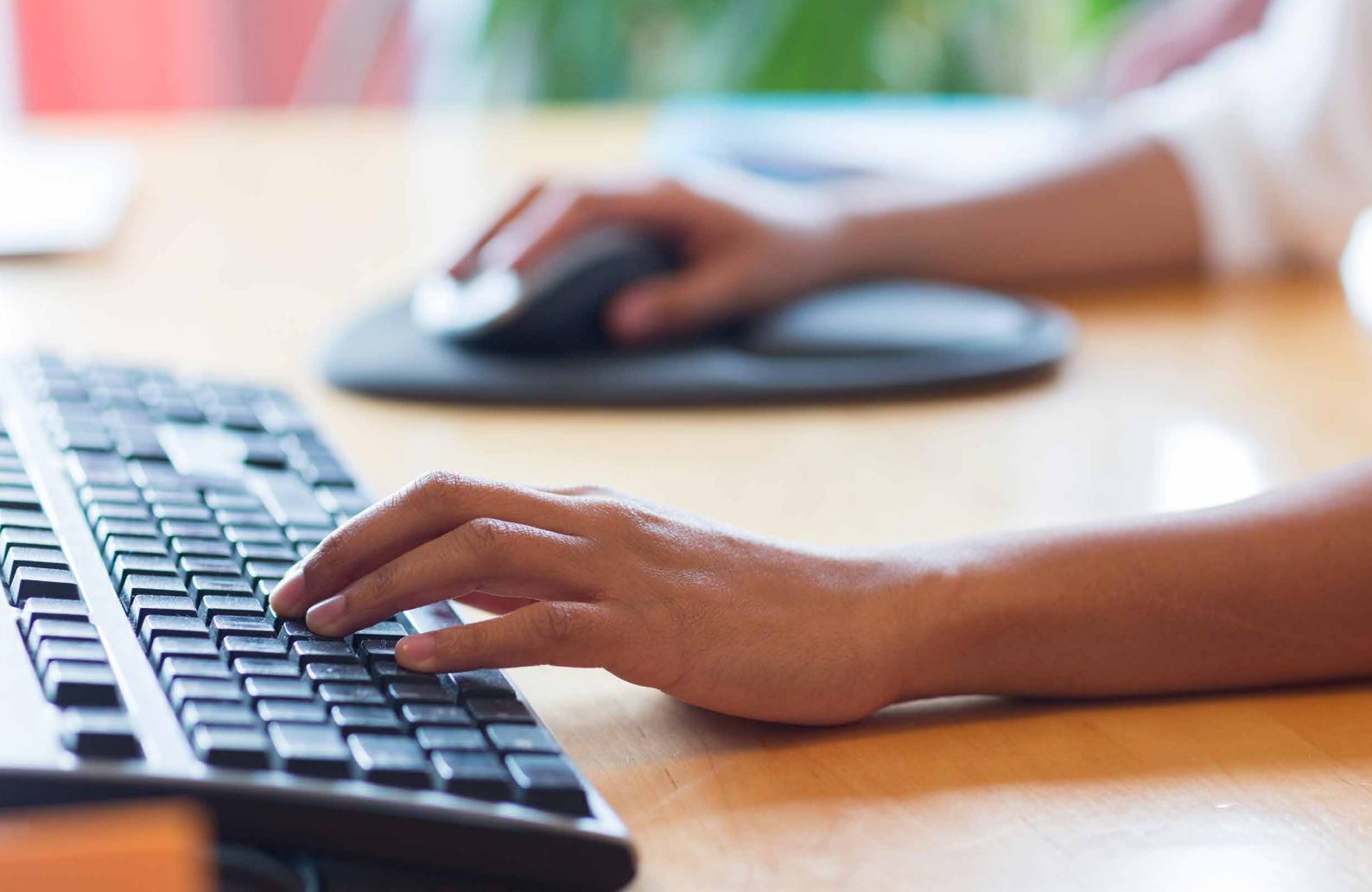


Metodología ágil centrada en potenciar las relaciones interpersonales como clave para el éxito en desarrollo del software, promoviendo el trabajo en equipo, preocupándose por el aprendizaje de los desarrolladores y propiciando un buen clima de trabajo.

XP se basa en la retroalimentación continua entre cliente y el equipo de desarrollo. La XP es especialmente adecuada para proyectos con requisitos imprecisos y muy cambiantes.

### Características específicas de la XP:

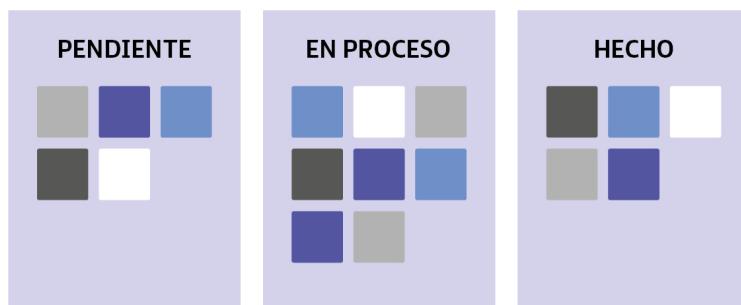
- Los requerimientos se expresan como escenarios (también llamados historias de usuario).
- Las liberaciones que se van haciendo del sistema se basan en esas historias de usuario.
- Todos los procesos se basan en la **programación a pares**: trabajo colectivo del grupo.
- Existe, por tanto, una refactorización constante.
- Se valora al individuo y las interacciones del equipo de desarrollo sobre el proceso y las herramientas. La gente es el principal factor de éxito de un proyecto de software.



- Se trata de desarrollar un software que funcione, más que conseguir una buena documentación.
- Se propone que exista una interacción y colaboración constante entre el cliente y el equipo de desarrollo.
- La habilidad de responder a los cambios que puedan surgir a lo largo del proyecto determina también el éxito o el fracaso de este. La planificación no debe ser estricta, sino flexible y abierta.

## Kanban

Es una palabra japonesa que significa "tarjetas visuales". Esta técnica se creó en Toyota, y se utiliza para controlar el avance del trabajo en el contexto de una línea de producción. Actualmente, está siendo aplicado en la gestión de proyectos software.



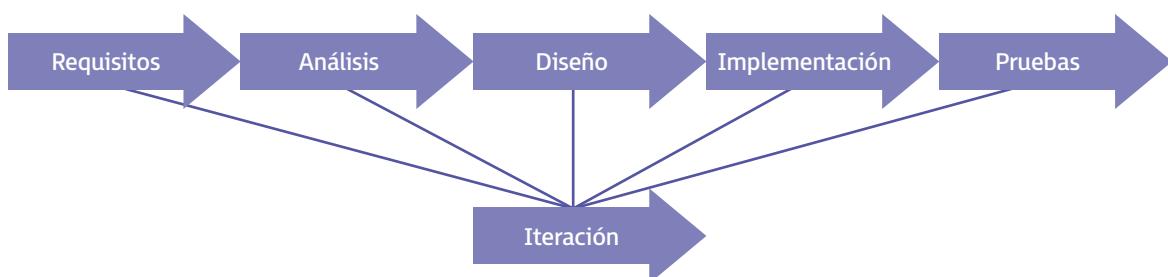
## Características específicas de Kanban

- Visualiza tu trabajo dividido en bloques.
- Tiene varias barras de progreso donde la tarea va avanzando según su estado.
- Tener una visión global de todas las tareas pendientes, de las tareas que estamos realizando y de las tareas que hemos realizado.
- Se pueden hacer cálculos de tiempo con las tareas finalizadas y tareas similares que vamos a realizar, para tener una idea del tiempo que cada uno puede tardar a realizarlas.
- Limita tu WIP (*work in progress*). Asigna límites en lo que respecta a cuántos elementos puedan estar en progreso en cada estado.

## Rational unified process (RUP)

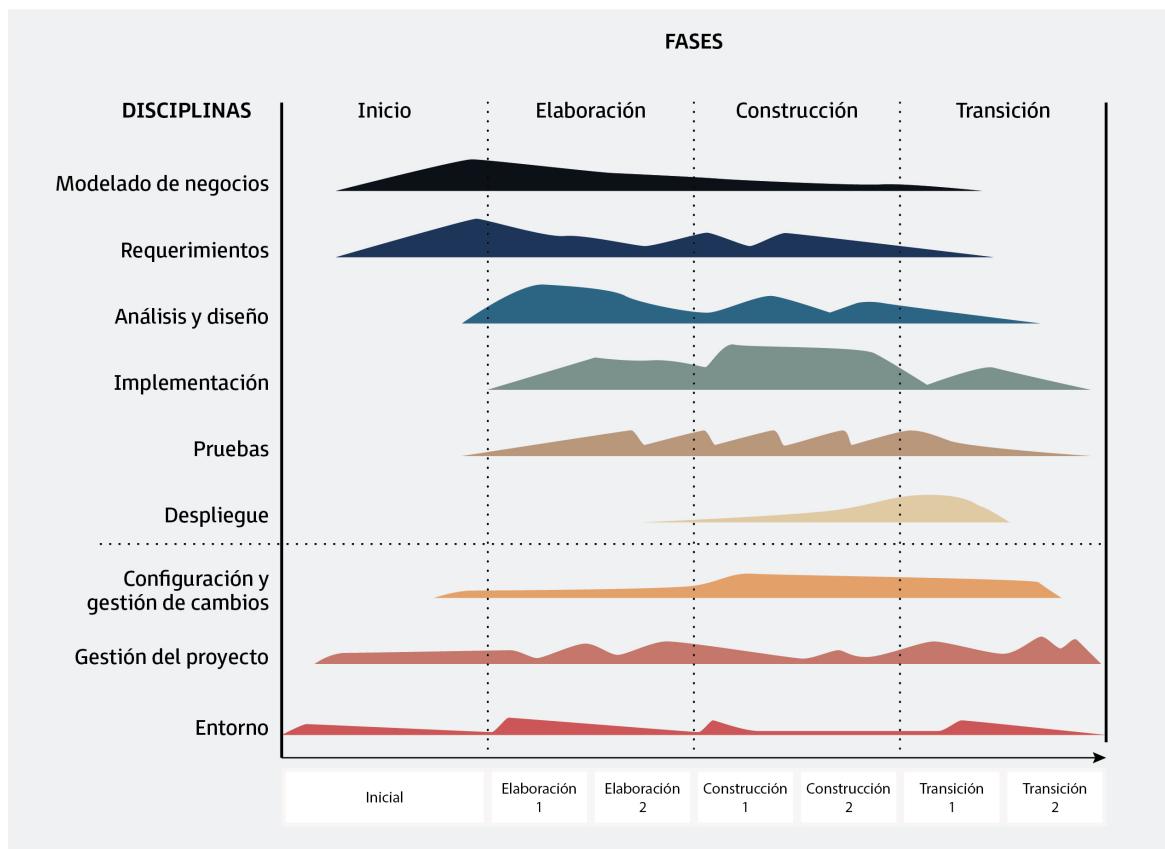
El objetivo es estructurar y organizar el desarrollo del software. Las características son:

- Está dirigido por los casos de uso.
- Centrado en la arquitectura del SW. Se centra en las diferentes vistas del sistema correspondiente a las fases explicadas anteriormente: análisis, diseño e implementación. Analiza el sistema como un todo y a la vez analiza cada una de sus partes.
- Es un proceso iterativo e incremental: Se divide en pequeños proyectos donde se va incrementando en funcionalidad.



La estructura de RUP se pone en práctica en tres perspectivas:

1. La dinámica: contendrá fases del sistema sobre el tiempo de desarrollo.
2. La estática: donde se muestran todas las actividades del proceso.
3. La práctica: donde se ponen de manifiesto las buenas prácticas de desarrollo (gestión de los requisitos, modelado en UML, verificación de calidad, etcétera).



ponte a prueba

**¿En qué metodología se trabaja por “sprints”?**

- a) SCRUM
- b) XP
- c) KANBAN
- d) Ninguna de las anteriores respuestas es correcta

**¿Qué caracteriza la metodología “programación extrema”?**

- a) La colaboración
- b) Valoración del programador
- c) Respuesta rápida en los cambios de la plataforma
- d) Todas las respuestas son correctas

## 1.7. PROCESO DE OBTENCIÓN DE CÓDIGO A PARTIR DEL CÓDIGO FUENTE. HERRAMIENTAS IMPLICADAS

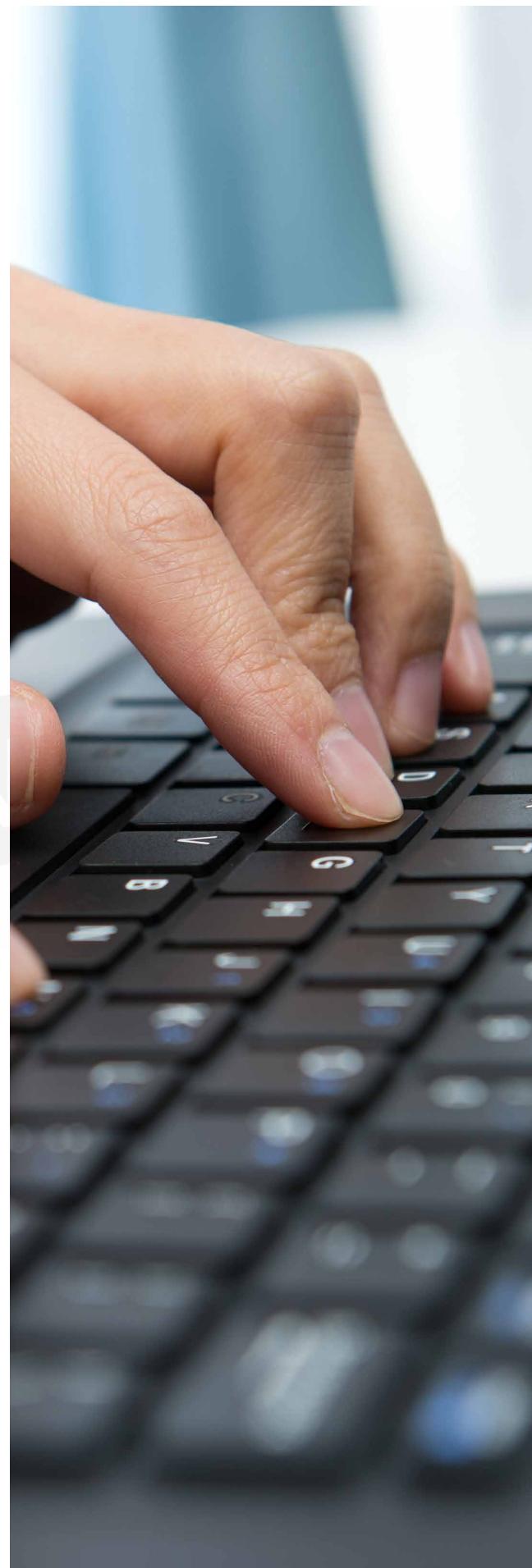
La generación de código fuente se lleva a cabo en la etapa de codificación. En esta etapa, el código pasa por diferentes estados.

### 1.7.1. TIPOS DE CÓDIGO:

- **Código fuente:** es el código escrito por los programadores, utilizando algún editor de texto o alguna herramienta de programación. Se utiliza un lenguaje de alto nivel.
- **Código objeto:** es el código resultante de compilar el código fuente. No es ejecutable por el ordenador ni entendido por la máquina, es un código intermedio de bajo nivel.
- **Código ejecutable:** es el resultado de enlazar el código objeto con una serie de rutinas y librerías, obteniendo así el código que es directamente ejecutable por la máquina.

### 1.7.2. OBTENCIÓN Y EDICIÓN DEL CÓDIGO EJECUTABLE. HERRAMIENTAS:

- **Librerías:** son archivos que contienen diferentes funcionalidades. Pueden ser llamados desde cualquier aplicación de un sistema operativo (por ejemplo, Windows está compuesto de librerías de enlaces dinámicos llamadas DLL que generan extensiones como .exe o .fon para las fuentes).
- **Editor:** es una herramienta para crear código fuente en un lenguaje de programación. Puede marcar la sintaxis de dicho lenguaje y realizar tareas de autocompletado. Ejemplos de editores son IntelliJ IDEA, Notepad++ o Sublime Text.
- **Compilador:** es la herramienta que nos proporciona el lenguaje máquina entendible por nuestro ordenador. A este proceso de traducción se le denomina compilación.
- **Enlazador:** el objetivo de esta herramienta es coger los objetos que se crean en los primeros pasos de la compilación, quitar los recursos que no necesita y enlazar el código objeto con sus bibliotecas. El resultado es un archivo ejecutable.





# 2

## INSTALACIÓN Y USO DE ENTORNOS DE DESARROLLO

En esta segunda parte de la unidad vamos a estudiar la utilización de los entornos de desarrollo, de los que conoceremos sus características, evaluaremos sus entornos integrados de desarrollo, instalación y configuración y abordaremos la creación de modelos de datos y desarrollo de programas.

## 2.1. FUNCIONES DE UN ENTORNO DE DESARROLLO

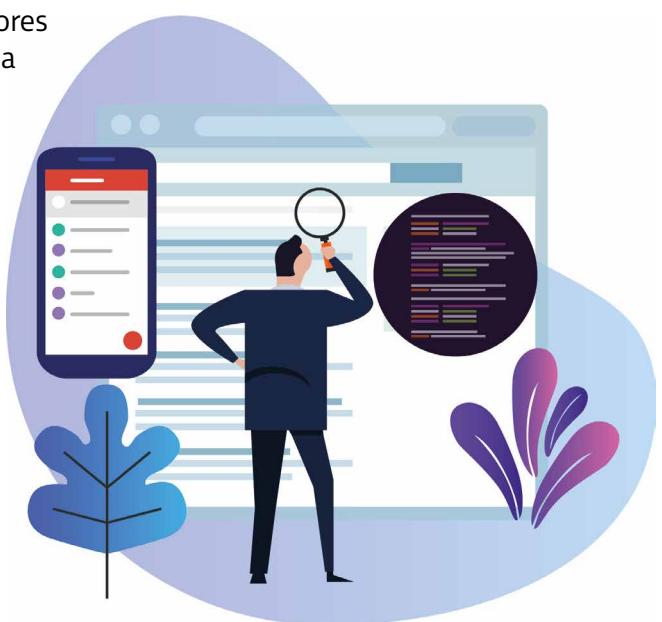
**Entornos de desarrollo**  
<https://youtu.be/7XUCiBOIwT8>



Un **IDE** (entorno integrado de desarrollo) es una aplicación informática que estará formada por un conjunto de herramientas de programación que simplifican la tarea al programador y agilizan el desarrollo de programas. Puede usarse con uno o varios lenguajes.

Los **componentes de un entorno de desarrollo** son:

- **Editor de texto.** Parte en la que escribimos el código fuente.
- **Compilador.** Se encarga de traducir el código fuente escrito en lenguaje de alto nivel a un lenguaje de bajo nivel que la máquina sea capaz de interpretar y ejecutar.
- **Intérprete o interpretador.** Realiza la traducción a medida que se ejecuta la instrucción. Son más lentos que los compiladores, pero no dependen de la máquina, sino del propio intérprete.
- **Depurador (debugger).** Depura y limpia los errores en el código fuente. Permite detener el programa en cualquier punto de ruptura para examinar la ejecución.
- **Constructor de interfaz gráfica.** Simplifica la creación de interfaces gráficas de usuario permitiendo la colocación de controles usando un editor WYSIWYG (acrónimo del inglés *what you see is what you get*) de arrastrar y soltar. Por ejemplo, en Java, la parte gráfica podemos trabajarla con Swing.
- **Control de versiones.** Controla los cambios realizados sobre las aplicaciones, obteniendo así revisiones y versiones de las aplicaciones en un momento dado. Ejemplos de control de versiones son GIT o TFS.



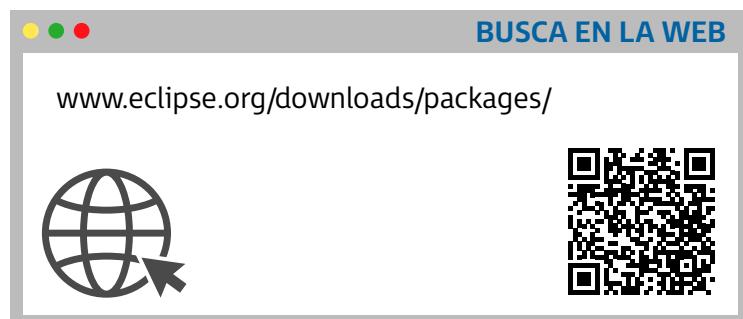


## 2.2. INSTALACIÓN DE UN ENTORNO DE DESARROLLO

Existen muchos tipos de entornos de desarrollo. Vamos a centrarnos en la instalación del IDE de Eclipse y del plugin de Swing. Eclipse es una herramienta con bastante éxito en el mercado.

### 2.2.1. INSTALACIÓN DE ECLIPSE

Accedemos a la web oficial de Eclipse para su descarga gratuita. Se va a descargar la versión para desarrolladores de Java:



Los requisitos de instalación son los siguientes:

- Sistema operativo Windows, Linux o Mac.
- Tener instalarlo el JDK (Java Development Kit) de Oracle (recomendado la instalación de JDK 7 o superior).

Deberemos seguir estos pasos para su instalación:

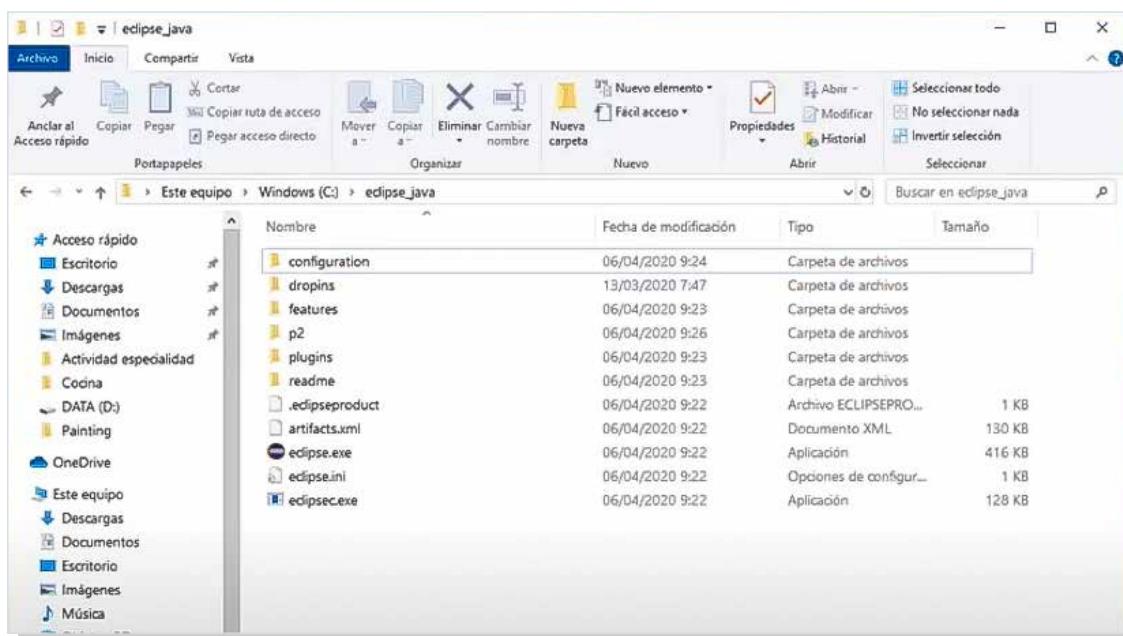
1. Descargar el editor de la página oficial: [eclipse.org](https://eclipse.org/).
2. Descomprimir el fichero en un directorio.
3. Ejecutar el fichero `eclipse.exe`.
4. Configurar el espacio de trabajo (`workspace`).

Veamos la instalación paso a paso:

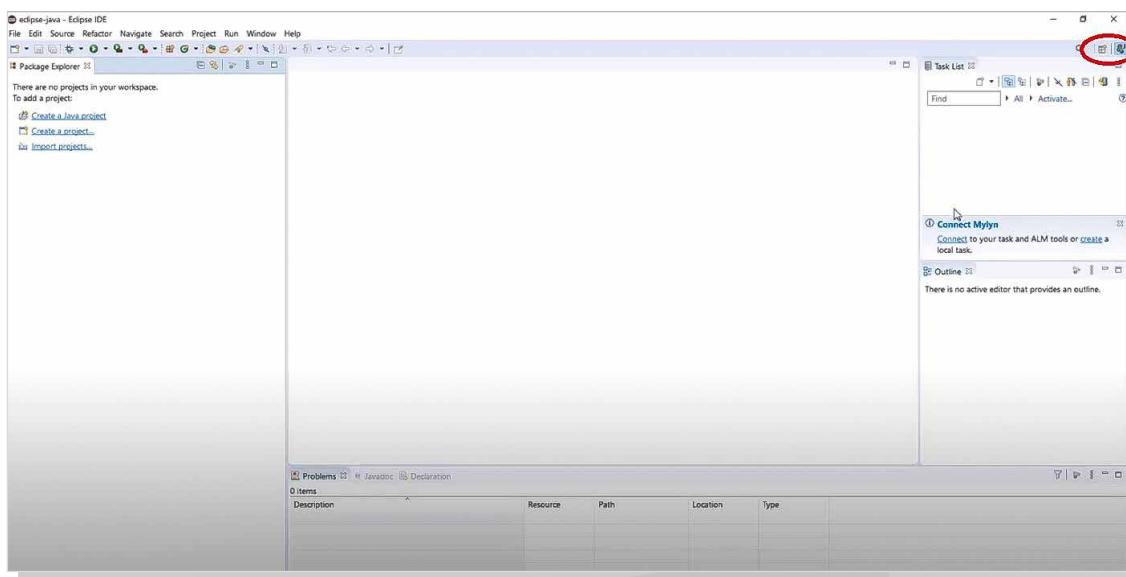
1. Descargamos la versión Eclipse IDE for JAVA Developers.

The screenshot shows the Eclipse IDE 2020-03 R Packages download page. It features a large central image of the Eclipse logo. Below it, the text "Eclipse IDE 2020-03 R Packages" is displayed. To the left of the packages, there is a small icon of a computer monitor with code on the screen. To the right, there is a download button with a downward arrow and the text "Windows 64-bit". The entire page has a clean, modern design with a white background and blue accents.

2. Una vez descargado el fichero, lo descomprimimos en una carpeta (en nuestro caso, la hemos llamado **`eclipse_java`**).

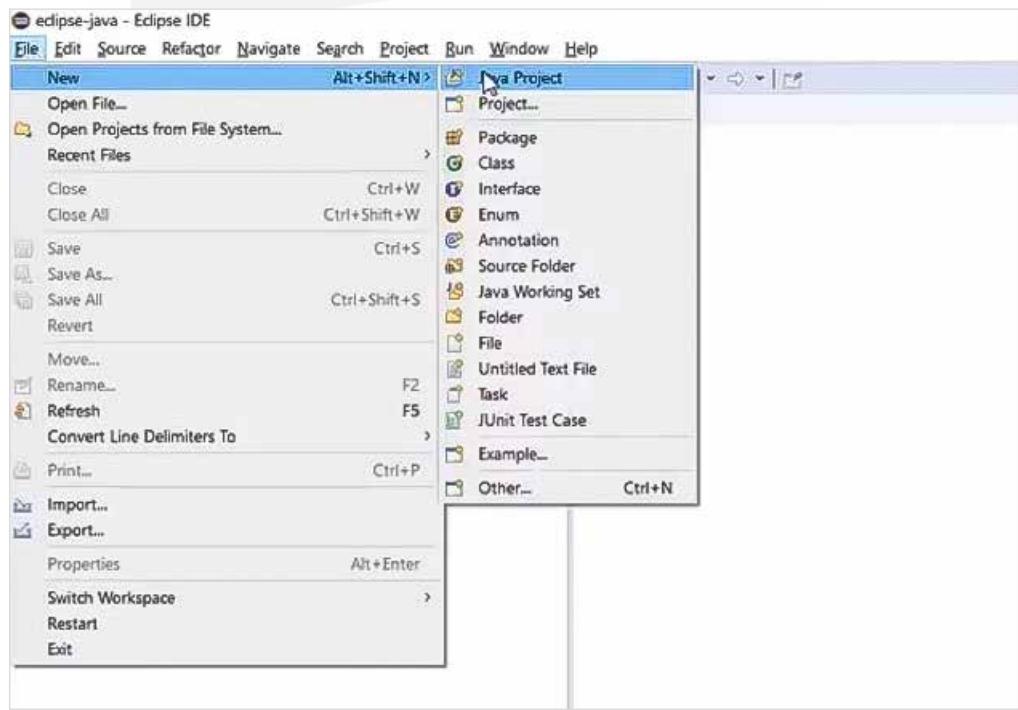


3. Ejecutamos el .exe y se nos abrirá una ventana donde nos pedirá el espacio de trabajo (*workspace*). Aquí guardaremos las preferencias de configuración y nuestros proyectos.
4. A continuación, se nos abre la interfaz donde vamos a desarrollar nuestros programas. Por defecto, se nos abre la perspectiva de Java.

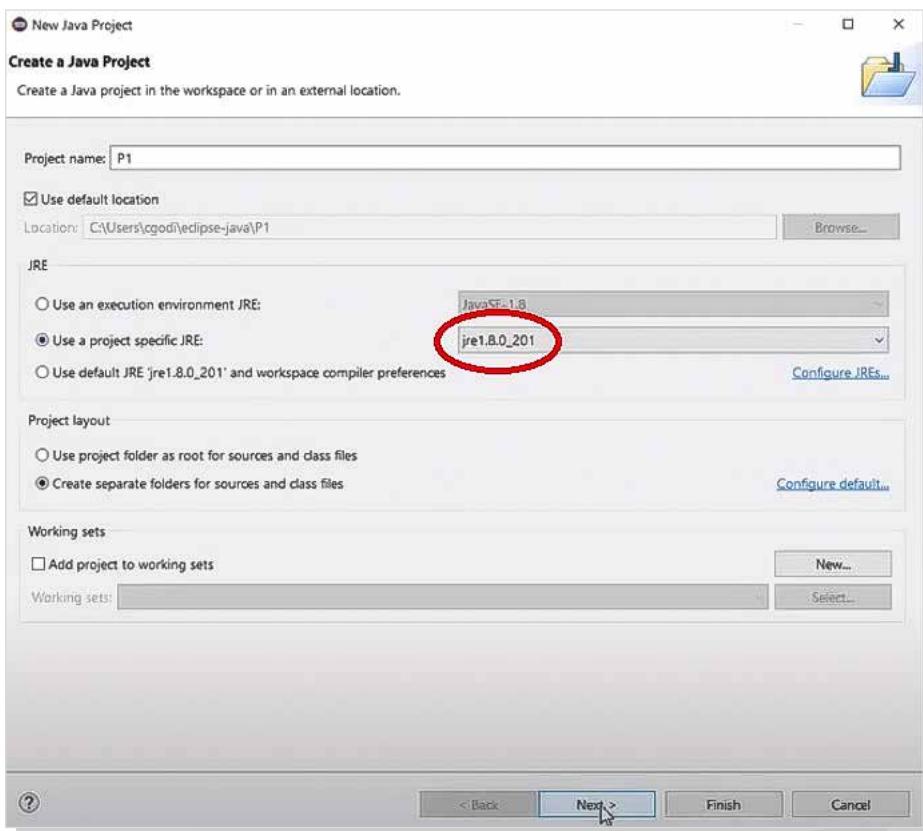


Podemos instalar diferentes *plugins* para poder trabajar con otros lenguajes (como XML).

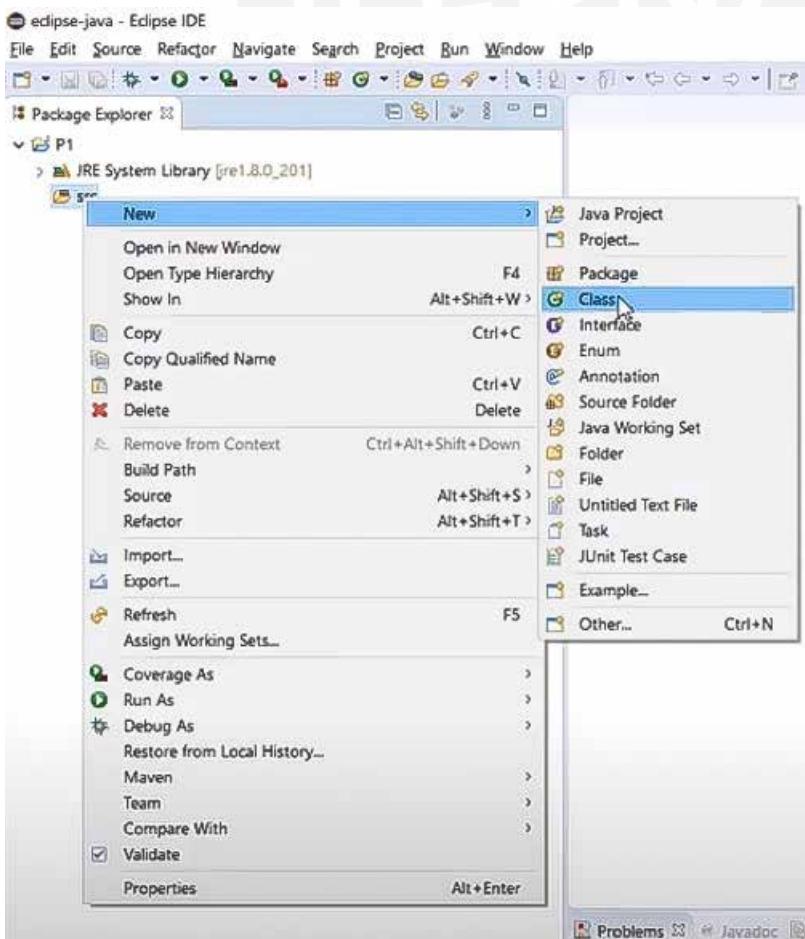
5. Para crear un nuevo proyecto en Java, vamos a Archivo -> New -> Java Project.



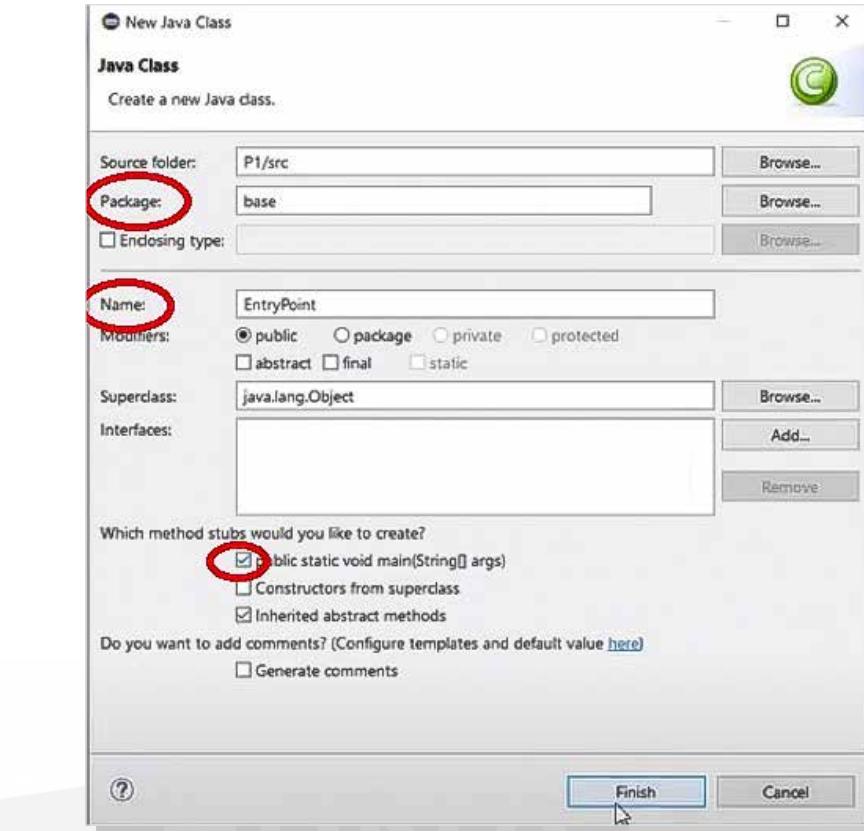
6. Especificamos en nuestro proyecto qué versión de JDK tenemos instalada.



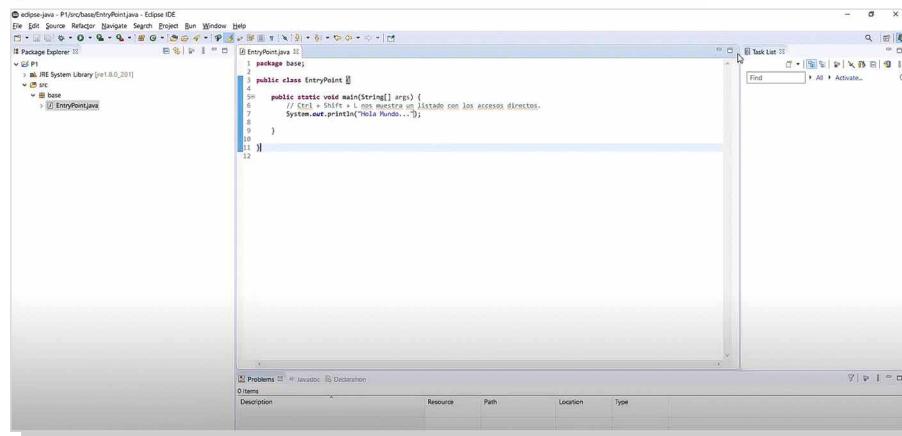
7. Ya podemos empezar a trabajar con nuestro IDE. Iremos a nuestra carpeta src (que se encuentra dentro de nuestra carpeta de proyecto), New -> Class.



8. Introducimos el nombre de nuestro paquete, de nuestra clase y, al ser nuestra primera clase, creamos el método público *Main ()*.



9. Ya podemos empezar a trabajar en nuestros proyectos creando las clases necesarias para el desarrollo de nuestro aplicativo.

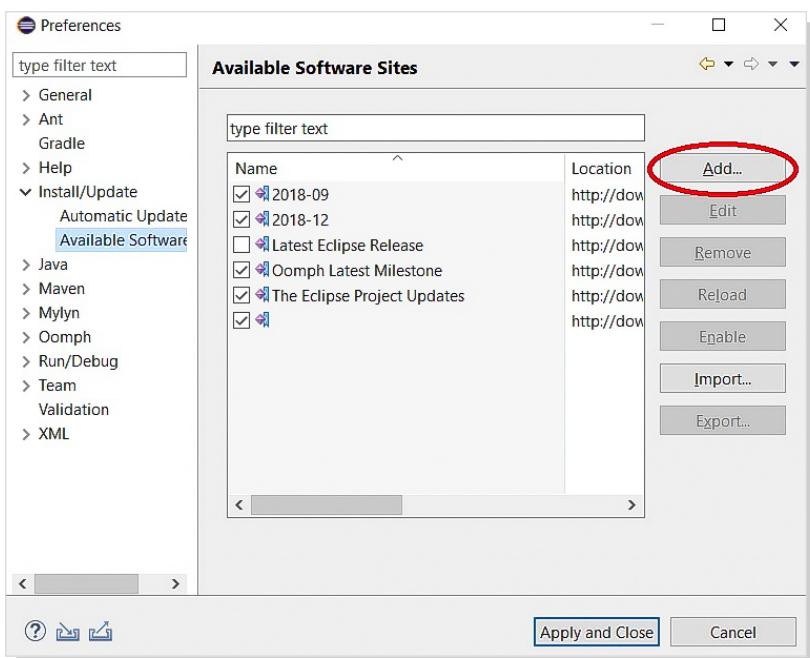


## 2.2.2. ACTUALIZACIÓN DE UN ENTORNO DE DESARROLLO

Para actualizar Eclipse IDE a la versión más actual, primero se necesita añadir el repositorio de la nueva versión de la forma siguiente:

1. Ventana -> Preferencias -> Instalar/Actualizar -> Sitios de software disponibles.

- Hacemos clic en Add e introducimos la URL del nuevo repositorio (por ejemplo, <https://download.eclipse.org/releases/2020-03/>).

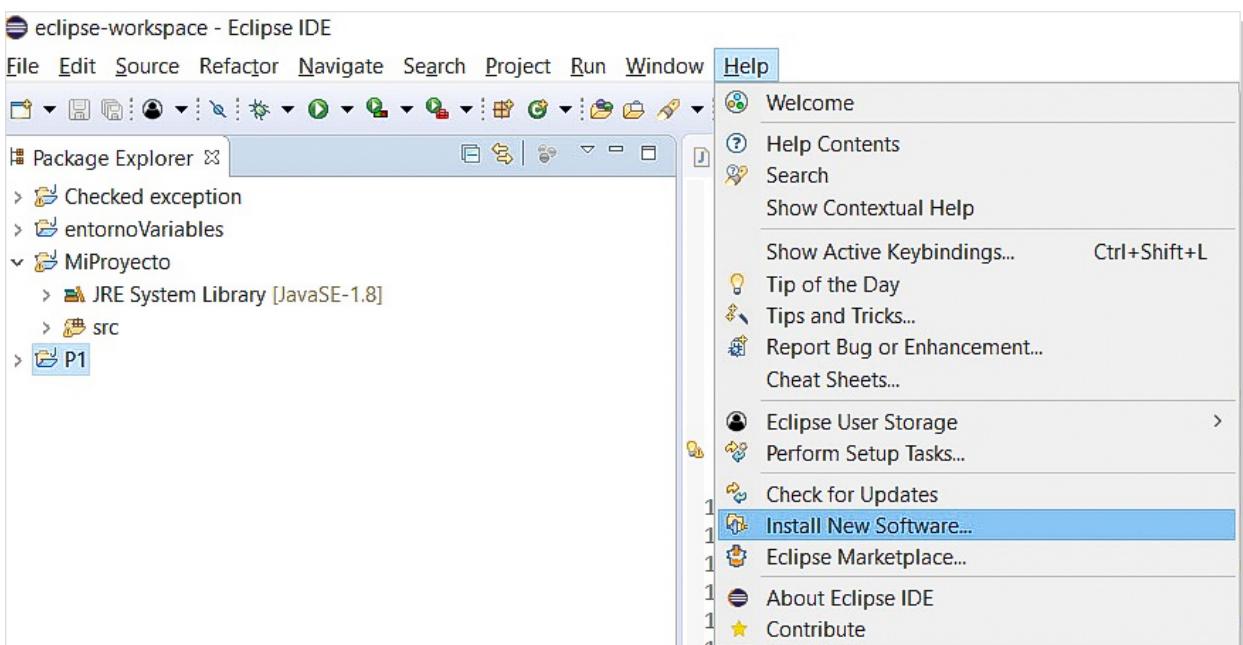


- Para que esas instalaciones sean efectivas, hay que reiniciar nuestro IDE porque nuestra pantalla de inicio puede almacenarse en caché.

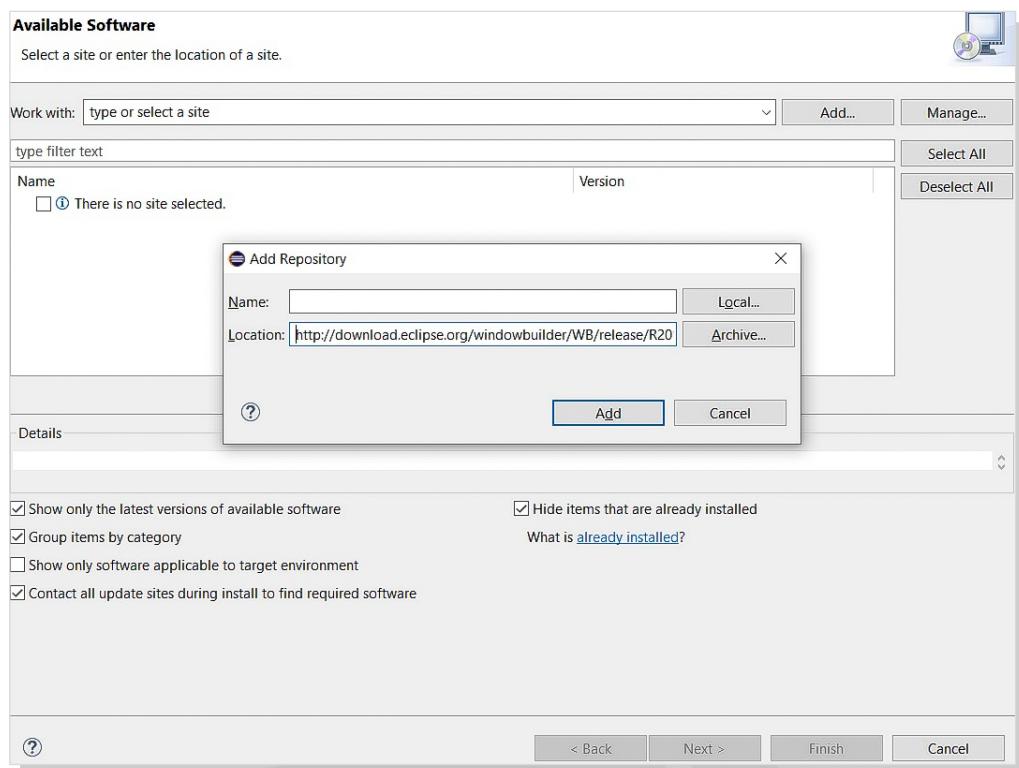
### 2.2.3. USO BÁSICO DE UN ENTORNO DE DESARROLLO. CREACIÓN DE UN PROYECTO CON UNA INTERFAZ GRÁFICA

A la hora de desarrollar un proyecto de interfaz gráfica, debemos instalar los *plugin* de Swing:

- Vamos al menú principal de nuestro IDE y seleccionamos Help -> Install New Software.

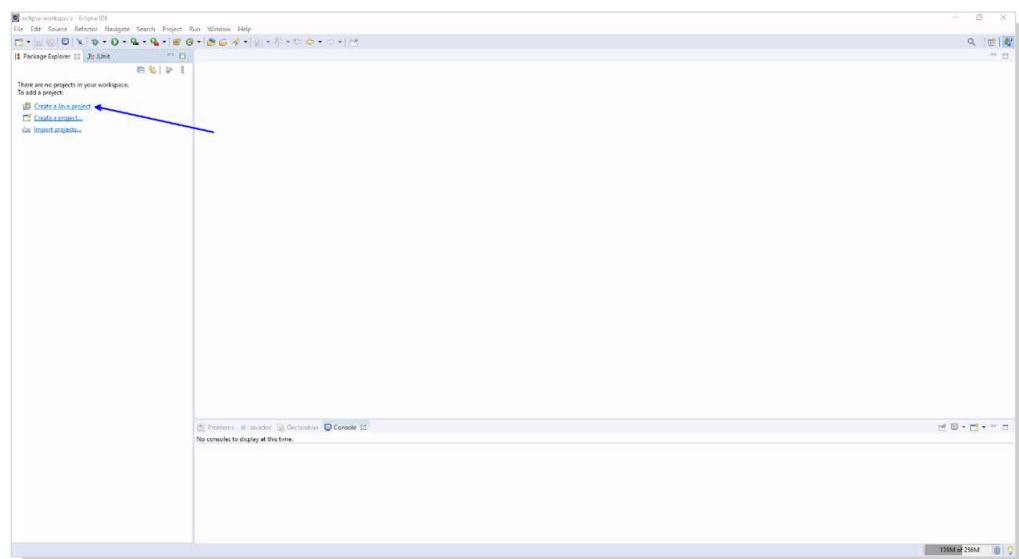


2. Seleccionamos el botón **Add** y añadimos la dirección de nuestro plugin: <http://archive.eclipse.org/windowbuilder/WB/release/R201206261200/3.7/>



3. Seleccionamos todos los paquetes de diseño de Swing.
4. Reiniciamos nuestro ordenador para que los cambios se hagan efectivos.
5. En el menú contextual, elegimos **New/Other**. Abrimos la carpeta **WindowsBuilder/Swing Designer** y escogemos **Application Window**.

Se habrá creado una clase con el nombre, no se podrá editar en modo *Source* (fuente) o en modo *Design* (diseño). En esta vista se podrán distinguir varios bloques:



- **Structure:** se verán de forma jerárquica los componentes que se han agregado a la ventana.
- **Properties:** mostrará las propiedades del elemento seleccionado.
- **Palette:** mostrará los elementos de tipo contenedor que se pueden añadir a la ventana.
- **La ventana o formulario:** espacio donde se van añadiendo los elementos.

Para añadir los componentes a dicha ventana, primero pulsamos en **Absolute layout (Palette/Layouts)** y lo arrastramos al marco interno de la ventana, lo que nos va a permitir colocar los componentes en cualquier parte de esta.

Otra forma para poder abrir la ventana en modo diseño es pulsando sobre el clic derecho del ratón y seleccionar **Open With/WindowsBuilder Editor**.

## 2.2.4. ENTORNOS DE DESARROLLO LIBRES Y COMERCIALES MÁS USUALES

Vamos a analizar los principales IDE que nos ofrece el mercado para los distintos lenguajes de programación:

- **Eclipse:** es un entorno de programación de código abierto y multiplataforma. Esta orientado a Java, pero podemos adaptar la programación casi a cualquier lenguaje con la instalación de *plugins*. Así, podemos trabajar con C++, PHP o Perl. Podemos realizar aplicaciones multiplataforma trabajando con Swing, que es una herramienta de interfaz gráfica de usuario (GUI).
- **Visual Studio:** diseñado por Microsoft, con versión de pago. Actualmente, tenemos IDE gratuitos como Visual Studio Community. Podemos desarrollar aplicaciones multiplataformas en WPF (Windows Presentation Foundation) trabajando con XAML. Este IDE es válido para lenguajes de Microsoft.
- **JetBrain:** es una compañía de desarrollo de software encargada de proporcionar herramientas para los programadores. Dentro de estas herramientas de desarrollo libre, podemos encontrar el IDE IntelliJ IDEA Community, donde podemos trabajar con lenguajes como Kotlin, Android, Java o Groovy.
- **Netbeans:** entorno integrado libre, multilenguaje y multiplataforma que fue desarrollado, en un principio, para un entorno Java. Actualmente, existen *plugins* para poder desarrollar en Android, Javascript, C/C++, etcétera.
- **CodeLite:** de código abierto bajo licencia GNU cuyo entorno integrado usa wxWidgets para su interfaz, manteniendo su filosofía de utilizar herramientas libres. Soporta lenguajes como PHP y Node.js.



## 2.2.5. USO DE HERRAMIENTAS CASE EN EL DESARROLLO DE SOFTWARE

Las herramientas CASE son un conjunto de aplicaciones informáticas cuyo fin es automatizar las actividades que realizamos durante el ciclo de vida del software.

Podemos dividirlas en dos grupos:

- **Upper case:** herramientas utilizadas en las etapas de capturas de requisitos, análisis y diseño (como, por ejemplo, herramientas para modelar diagramas en UML).
- **Low case:** herramientas utilizadas en las etapas de implementación, pruebas y mantenimiento (como ejemplo tenemos la herramienta **Bugzilla** para encontrar errores de código).

Otra clasificación que podemos realizar de estas herramientas es **según su funcionalidad**:

- **Herramientas de diagramas:** nos sirven para representar componentes del sistema y su interacción. Un ejemplo sería la herramienta Flow Chart Maker.
- **Herramientas de documentación:** recordamos que debemos documentar todo el proceso de ingeniería, desde los requisitos hasta su mantenimiento. Por tanto, debe-

mos crear documentación tanto para el usuario como para profesionales (manuales de usuario, de instalación, de sistema, etcétera). Un ejemplo de herramienta es Doxygen.

- **Herramientas de análisis:** estas herramientas analizan de forma automática si hay algún error en los diagramas o alguna inconsistencia. Así, por ejemplo, disponemos de la herramienta CaseComplete.
- **Herramientas de control de cambios:** gracias a ellas, podemos ver los cambios realizados en el software. Estas herramientas automatizan la opción de destacar los cambios, la gestión de estos o incluso la gestión de archivos. Sería el caso de la herramienta GitHub.
- **Herramientas de mantenimiento:** nos ayudan a gestionar y organizar la fase de mantenimiento del software, reportando errores o analizando cuál puede ser la causa. Un ejemplo es la herramienta Bugzilla.

**ponte a prueba**

**Con el entorno de desarrollo de Eclipse podemos modelar en UML.**

a) Verdadero  
b) Falso

**¿Qué caracteriza a la herramienta MySQL Workbench?**

a) Herramienta visual de diseño de bases de datos  
b) Nos permite administrar bases de datos  
c) Podemos hacer un mantenimiento de nuestras bases de datos  
d) Todas las respuestas son correctas

**¿Qué funcionalidades nos proporciona la herramienta CASE?**

a) Generación semiautomática de código  
b) Refactorización  
c) Editores de UML  
d) Todas las respuestas son correctas



# 3

## DISEÑO Y REALIZACIÓN DE PRUEBAS

En esta sección, vamos a aprender cómo se usan las distintas técnicas para realizar casos de prueba. Veremos cómo probar nuestros códigos y detectar errores. Además, aprenderemos a usar la herramienta JUnit para poder elaborar pruebas unitarias para clases Java.

## 3.1. PLANIFICACIÓN DE PRUEBAS

Un caso de prueba son los tipos de entradas que podemos realizar a través de unas condiciones de ejecución y unos resultados que esperamos, y será desarrollado con el fin de conseguir un objetivo particular de prueba. Será necesario que se definan unas **precondiciones** y **poscondiciones**, saber los valores de entrada que tenemos que darle y conocer cómo reacciona el programa ante la entrada de estos valores. Tras realizar el análisis y haber introducido los datos en el sistema, observamos el comportamiento y si es el previsto por el sistema o no. De esta manera, determinaremos si ha superado la prueba.

Una **precondición** es una condición determinada que debe cumplir un conjunto de parámetros. Fijar estas precondiciones nos ayuda a ver que hay casos que no son necesarios testear. Por ejemplo, en una división, una precondición que debe cumplir es que el divisor sea distinto de 0.

Una **poscondición** es una condición que cumplirá el valor devuelto. Las poscondiciones se prueban mediante **assertiones** que están incluidas dentro del código.

Una **aserción** es un predicado incluido en el código por parte del programador donde se asegura que siempre se va a cumplir en ese punto del programa.

Por ejemplo:

```
int numero=10;
if ((numero % 2) ==0 {//numero es par}
else {//numero es impar
    assert (total % 2 == 1);}
```

Para el diseño de los casos de prueba tendremos dos tipos de técnicas, como ya hemos visto con anterioridad: prueba **de caja blanca**, centrada en validar la estructura interna del programa, y **prueba de caja negra**, basada en los requisitos funcionales sin fijarse en el funcionamiento interno del programa. Ambas pueden combinarse para descubrir cualquier tipo de error.

### 3.1.1. PRUEBAS DE CAJA BLANCA

Su funcionamiento se basa en un exhaustivo examen de los detalles **procedimentales del código**.

En las pruebas de caja blanca, tendremos acceso al código fuente y se analizan todas posibilidades de nuestro código.

A *priori*, podemos pensar que podemos recorrer todos los posibles casos, pero en la práctica veremos que es inviable. Por ejemplo:

```
If (a<100 && b<10) { // código que modifique a o b}
```

Considerando el rango de valores de 1 a 100, tendríamos  $100^2 = 10000$  posibilidades. Obviamente, si tuviéramos tres condiciones tendríamos  $100^3$  posibilidades.

Lo que sí podemos es analizar un conjunto de caminos independientes y crear los casos de prueba que nos permitan probar dichos caminos en la ejecución.

Se podrán obtener casos de prueba que aseguren que:

- Se ejecutan por lo menos una vez todos los caminos de cada módulo.
- Todas las sentencias sean ejecutadas al menos una vez.
- Todas las decisiones lógicas se ejecuten al menos una vez en parte verdadera y otra en la falsa.
- Todos los bucles sean ejecutados en sus límites.
- Se usen todas las estructuras de datos internas que aseguren su validez.

Una de las pruebas más utilizadas de este tipo será la del camino básico, que abordaremos más adelante.

### 3.1.2. PRUEBA DE CAJA NEGRA

También se conoce como prueba de comportamiento. Se realiza sobre la interfaz sin necesidad de conocer la estructura del programa (el código fuente no está disponible) ni cómo funciona. Lo que se busca es que demuestre que las **funciones del software son operativas**.

El sistema se verá como una caja negra en la que solo podremos observar la entrada y la salida que nos devuelve en función de los datos aportados.

Los errores que pretendemos buscar son los siguientes:

- Errores de interfaz, en estructuras de datos o en las bases de datos externas.
- Funcionalidades erróneas en el inicio o la finalización del programa.

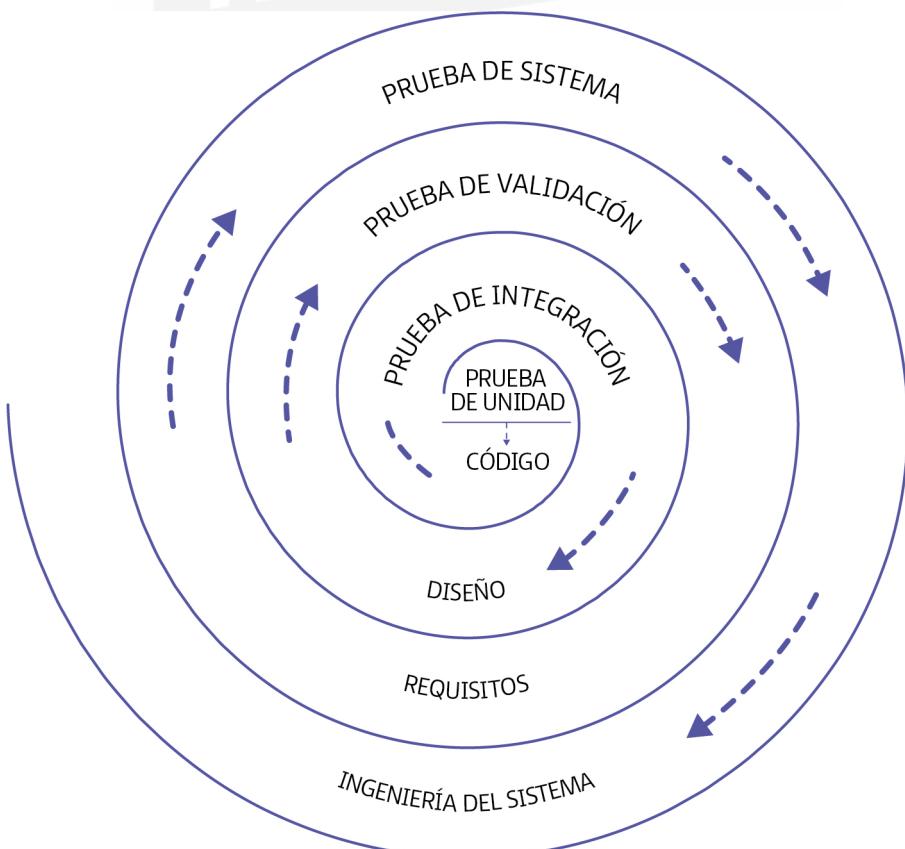
Algunas técnicas para estos casos son:

- Clases de equivalencia.
- Análisis de valores límite.

## 3.2. ESTRATEGIAS DE PRUEBAS DE SOFTWARE. AUTOMATIZACIÓN DE PRUEBAS

Podemos representar las estrategias de pruebas de software como una enorme espiral en la que ubicaremos los diferentes tipos de prueba:

- En el vértice situaremos la **prueba de unidad**. Está centrada en la unidad más pequeña, el módulo, tal cual está en el código fuente.
- La siguiente es la **prueba de integración**. Construimos una estructura con los módulos probados en la prueba anterior. El diseño será el foco de atención.
- Seguidamente, nos encontramos con la **prueba de validación**. Es la prueba que realizará el usuario en el entorno final de trabajo.
- La última es la **prueba del sistema**. Se probará que cada elemento esté construido de forma eficaz y funcional. El software del sistema se prueba como un todo.



### 3.2.1. PRUEBA DE UNIDAD

En esta prueba vamos a comprobar cada módulo para eliminar cualquier tipo de error en la interfaz o en la lógica interna. Utiliza ambas técnicas, tanto la prueba de la caja negra como la de la blanca. Se realizarán pruebas sobre:

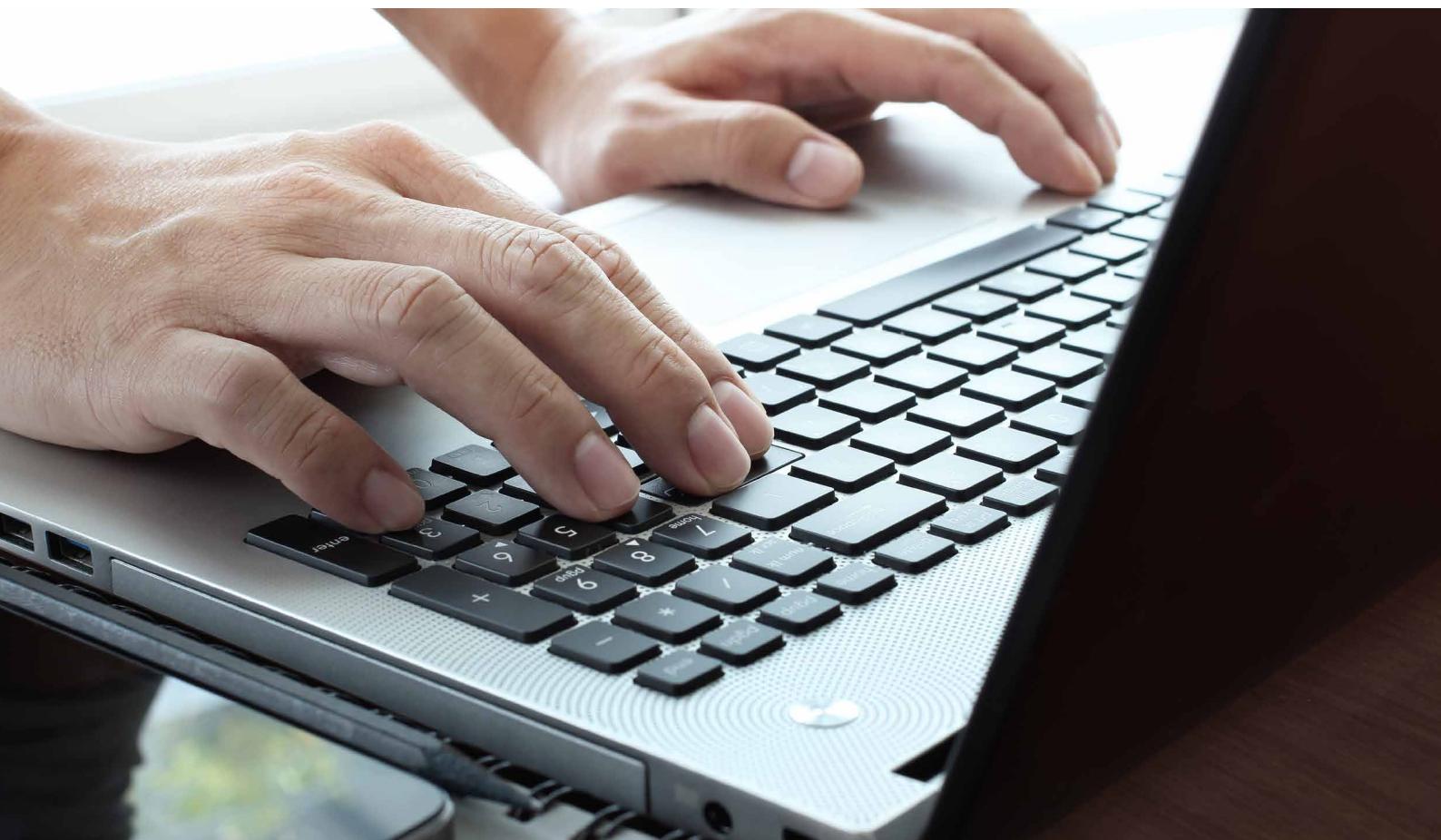
- La interfaz del módulo.
- La estructura de datos locales: comprobación de integridad.
- Las condiciones límite: comprobación de que funciona en los límites establecidos.
- Caminos independientes de la estructura de control, lo que implica asegurarse de que se ejecutan las sentencias al menos una vez.
- Todos los caminos de manejo de errores.

Algunas herramientas usadas para estas pruebas son JUnit, CPPUNIT, PHPUnit, entre otras.

#### Pruebas unitarias con JUnit

En este apartado veremos cómo funciona un programa que realiza pruebas para verificar nuestro aplicativo.

La herramienta que utilizaremos para las pruebas automatizadas será **JUnit**. Estará integrada en Eclipse, por lo que no deberemos descargarnos ningún paquete. Estas pruebas se realizarán sobre una clase, independientemente del resto de clases.



## Preparación y ejecución de las pruebas

Antes de preparar el código, vamos a ver los tipos de métodos para realizar comprobaciones, estos métodos devolverán tipo *void*:

MÉTODOS	MISIÓN
<code>assertTrue(boolean expresión)</code> <code>assertTrue(String mensaje, boolean expression)</code>	Comprueba que la expresión se evalúe <i>true</i> . Si no es <i>true</i> y se incluye el <i>string</i> , al producirse error se lanzará el <i>mensaje</i> .
<code>assertFalse(Boolean expresión)</code> <code>assertFalse(String mensaje, Boolean expresión)</code>	Comprueba que la expresión se evalúe <i>false</i> . Si no es <i>false</i> y se incluye el <i>string</i> , al producirse error se lanzará el <i>mensaje</i> .
<code>assertEquals(valorEsperado, valorReal),</code> <code>assertEquals(String mensaje, valorEsperado, valorReal)</code>	Comprueba que el <i>valorEsperado</i> sea igual al <i>valorReal</i> . Si no son iguales y se incluye el <i>string</i> , entonces se lanzará el <i>mensaje</i> . <i>ValorEsperado</i> y <i>ValorReal</i> pueden ser de diferentes tipos.
<code>assertNull(Object objeto),</code> <code>assertNull(String mensaje, Object objeto)</code>	Comprueba que el <i>objeto</i> sea <i>null</i> . Si no es <i>null</i> y se incluye el <i>string</i> , al producirse error se lanzará el <i>mensaje</i> .
<code>assertNotNull(Object objeto),</code> <code>assertNotNull(String mensaje, Object objeto)</code>	Comprueba que el <i>objeto</i> no sea <i>null</i> . Si no es <i>null</i> y se incluye el <i>string</i> , al producirse error se lanzará el <i>mensaje</i> .
<code>assertSame(Object objetoEsperado, Object objetoReal)</code> <code>assertSame(String mensaje, Object objetoEsperado, Object objetoReal)</code>	Comprueba que <i>objetoEsperado</i> y <i>objetoReal</i> sean el mismo objeto. Si no son el mismo y se incluye el <i>string</i> , al producirse el error se lanzará el <i>mensaje</i> .
<code>assertNotSame(Object objetoEsperado, Object objetoReal)</code> <code>assertNotSame(String mensaje, Object objetoEsperado, Object objetoReal)</code>	Comprueba que <i>objetoEsperado</i> y <i>objetoReal</i> no sean el mismo objeto. Si no son el mismo y se incluye el <i>string</i> , al producirse el error se lanzará el <i>mensaje</i> .
<code>fail()</code> <code>fail(String mensaje):</code>	Hace que la prueba falle. Si se incluye un <i>string</i> , la prueba falla, lanzando el <i>mensaje</i> .

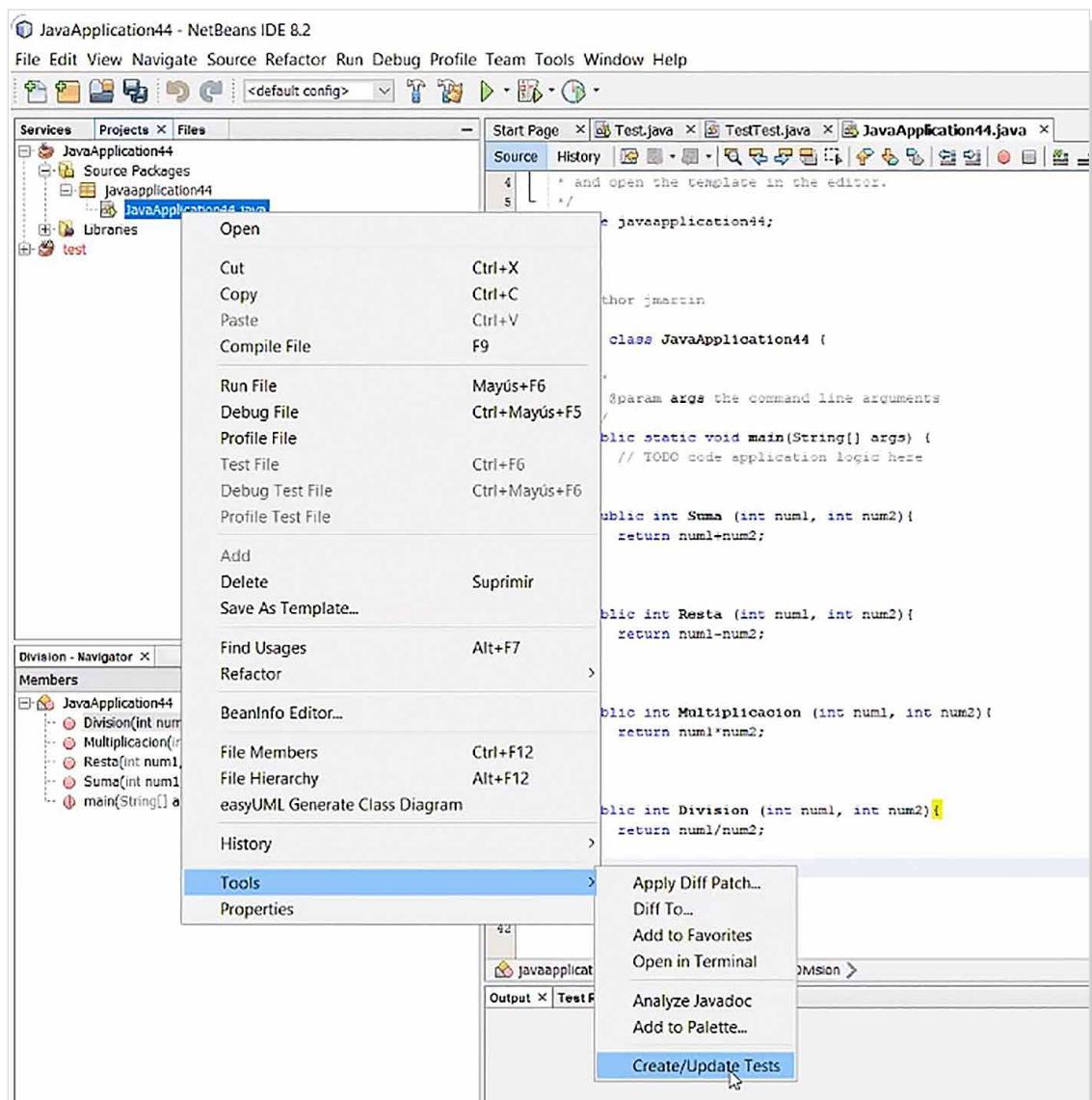
## Creación de pruebas

Vamos a utilizar el IDE de Netbeans para crear las pruebas de nuestro código.

Tenemos el siguiente fragmento de código, donde realizamos cuatro módulos de suma, resta, multiplicación y división:

```
/*
 * To change this license header, choose License Headers in Project
Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package test;
/**
 *
 * @author jmartin
 */
public class Test {
    public static void main(String[] args) {
    }
    /**
     *
     * @param num1 Este es mmi parámetro 1
     * @param num2
     * @return
     */
    public int Suma (int num1, int num2){
        return num1+num2;
    }
    public int Resta (int num1, int num2){
        return num1-num2;
    }
    public int Multiplicacion (int num1, int num2){
        return num1*num2;
    }
    public int Division (int num1, int num2){
        return num1/num2;
    }
}
```

Iremos a nuestro IDE (en este caso, utilizaremos Netbeans), y crearemos sobre nuestra clase un nuevo test.



Si la versión del IDE no incluye las librerías de JUnit, iremos a la página oficial (<https://junit.org/junit5/>) y descargaremos las librerías junit.jar y hamcrest-core.jar

Una vez creadas estas pruebas, tendrán la siguiente estructura:

```
public void testSuma() {
    System.out.println("Suma");
    int num1 = 10;
    int num2 = 10;
    test.Test instance = new test.Test();
    int expResult = 30;
    int result = instance.Suma(num1, num2);
    assertEquals(expResult, result);
    // TODO review the generated test code and remove the default call to fail.
    //fail("The test case is a prototype.");
}
```

### Tipos de anotaciones

JUnit tiene disponibles unas anotaciones que permiten ejecutar el código antes y después de las pruebas:

- **@Before**: si queremos que se inicie un método antes de que se ejecute cualquier método de prueba, usaremos esta anotación en dicho método. Puede haber varios con esta etiqueta.
- **@After**: si colocamos esta etiqueta en el método, haremos que se ejecute después de cualquier tipo de prueba. Puede haber varios métodos con esta anotación.
- **@BeforeClass**: solo podrá tener esta etiqueta un solo método, y se iniciará al principio de realizar las pruebas.
- **@AfterClass**: solo podrá tener esta etiqueta un solo método, y se ejecutará una vez que se hayan finalizado todas las pruebas.

### Pruebas parametrizadas

Para ejecutar una prueba varias veces pero con distintos valores, JUnit nos lo permite, siguiendo estos pasos:

1. Añadir etiqueta **@RunWith(Parameterized.class)** a la clase de prueba. Con esta especificación indicamos que será usada para realizar varios tipos de prueba. Se declarará un parámetro para cada tipo de prueba, y un constructor tendrá tantos argumentos como parámetros.
2. Para devolver la lista de valores que testear, incluiremos en el método la etiqueta **@Parameters**.

## 3.2.2. PRUEBA DE INTEGRACIÓN

Se comprobará la interacción de los distintos módulos del programa. Hemos visto en la anterior prueba que los módulos pueden funcionar individualmente, pero en esta prueba es donde realmente se comprueba que funcionan correctamente, al tener que hacerlo de forma conjunta.

Podemos enfocarla de dos formas distintas:

- **Integración no incremental o big bang**. Comprobación de cada módulo por separado y después se prueba de forma conjunta. Se detectan muchos errores y la corrección es difícil.
- **Integración incremental**. En este caso, el programa se va creando y probando en pequeñas secciones, por lo que localizar los fallos es más sencillo. En esta integración podemos optar por dos estrategias:

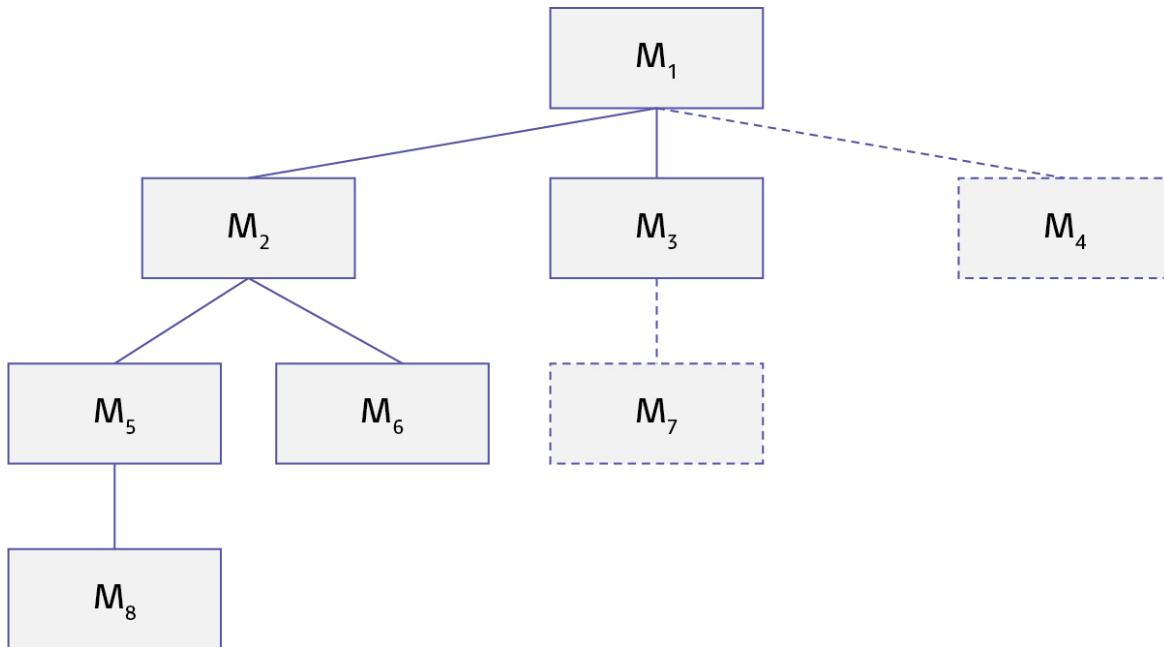
**ponte a prueba**

**¿Qué realiza la siguiente instrucción en Junit?**

```
assertTrue(String mensaje,
boolean expression)
```

- Comprueba que la expresión se evalúe a true. Si no es true y se incluye el string, al producirse, error se lanzará un mensaje.
- Comprueba que la expresión se evalúe a false. Si no es false y se incluye el string, al producirse error se lanzará un mensaje.
- Comprueba que el objeto no sea nulo.
- Comprueba que la expresión se evalúe a true. Si es true y se incluye el string, al producirse error, se lanzará un mensaje.

- **Ascendente.** Se comienza con los módulos más bajos del programa.
- **Descendente.** Se empieza por el módulo principal, descendiendo por la jerarquía de control.



El recorrido puede hacerse en profundidad ( $M_1, M_2, M_5, M_8$ ) o en anchura (Nivel 1  $\rightarrow M_2, M_3, M_4$ , Nivel 2  $\rightarrow M_5, M_6, M_7$ ).



### ponte a prueba

#### ¿Qué es una integración *big bang*?

- Una prueba de unidad.
- Una prueba donde integramos todos los módulos sin niveles establecidos.
- Una prueba donde integramos todos los módulos desde los niveles más bajos a los más altos.
- Una prueba donde integramos todos los módulos desde los niveles más altos a los más bajos.

### 3.2.3. PRUEBA DE VALIDACIÓN

Las pruebas de validación darán comienzo cuando:

- El software ya está ensamblado y se han detectado y corregido los errores.
- Las pruebas están realizadas de tal forma que las acciones ya son visibles para el usuario.
- Por tanto, ya se cumple el documento especificado de requisitos de software.

Se llevarán a cabo pruebas con la técnica de caja negra. Se podrán usar estas técnicas:

- **Prueba alfa:** realizada por el cliente o usuario en el lugar de desarrollo. Usará el programa bajo la observación del desarrollador, que irá registrando los errores.
- **Prueba beta:** realizada por los usuarios finales en su lugar de trabajo sin la presencia del desarrollador. En este caso, será el usuario el que registre los errores y se los comunique al desarrollador para que realice las modificaciones correspondientes y cree una nueva versión del producto.

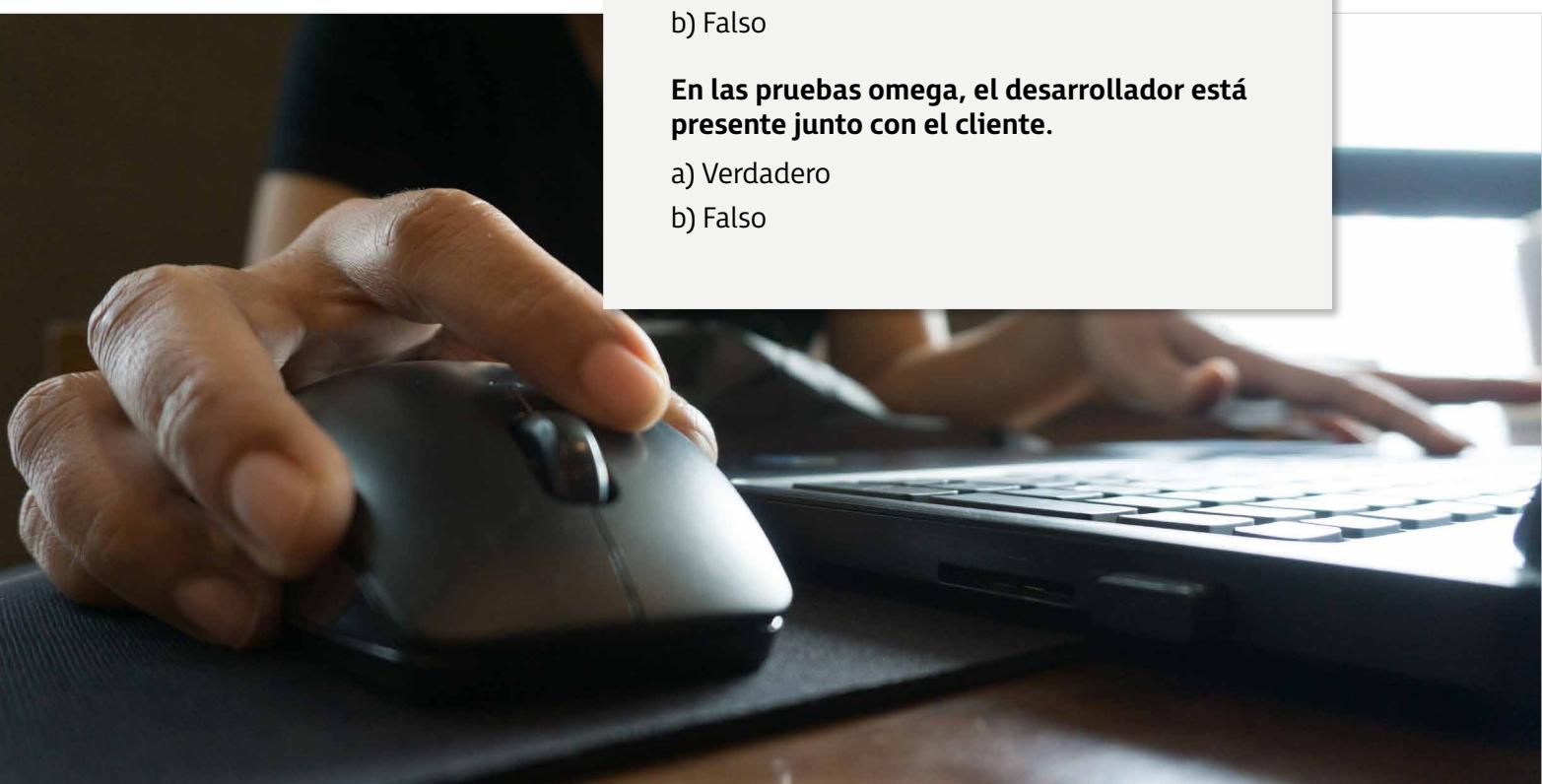
**ponte a prueba**

**En las pruebas beta, el desarrollador se encuentra presente junto con el cliente.**

- a) Verdadero
- b) Falso

**En las pruebas omega, el desarrollador está presente junto con el cliente.**

- a) Verdadero
- b) Falso



### 3.2.4. PRUEBA DEL SISTEMA

Está conformada por varias pruebas que tendrán como misión ejercitarse en profundidad el software. Serán las siguientes:

- **Prueba de recuperación:** se fuerza el fallo del software y se comprueba que la recuperación del sistema se realiza correctamente.
- **Prueba de seguridad:** se comprueba que el sistema esté protegido frente a acciones ilegales y se examina los mecanismos de control.
- **Prueba de resistencia (stress).** Se lleva el sistema al límite de los recursos, sometiéndolo a cargas masivas. El objetivo es comprobar los extremos del sistema.

## 3.3. PRUEBAS DE CÓDIGO: COBERTURA, VALORES LÍMITE Y CLASES DE EQUIVALENCIA

### Tipos de pruebas

<https://youtu.be/Z-HMeUuYVHQ>



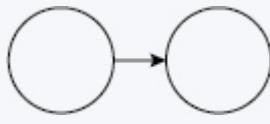
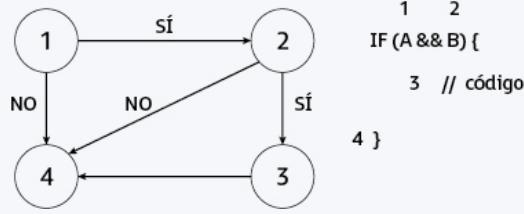
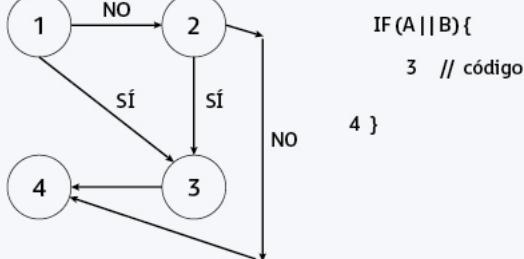
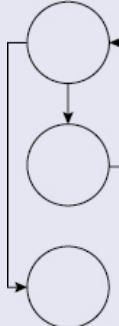
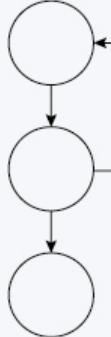
Las pruebas de código consisten en la ejecución del programa para localizar errores. Se comienza para su ejecución con varias entradas y una serie de condiciones de ejecución, y observamos y registramos los resultados para compararlos con los resultados que esperamos. Comprobaremos si el resultado es el esperado o no y por qué.

Para este tipo de pruebas, se mostrarán diferentes técnicas que van a depender del tipo de enfoque que queramos utilizar: de caja blanca o negra.

### 3.3.1. PRUEBA DEL CAMINO BÁSICO

Es una técnica que, mediante pruebas de caja blanca, permite al desarrollador obtener la medida de la complejidad de nuestro sistema. Se puede usar esta medida para la definición de un conjunto de caminos de ejecución. Para obtener esta medida de complejidad, utilizaremos la técnica de representación de **grafo de flujo**.

El grafo de flujo de las estructuras de control se representa de la siguiente forma:

Estructura	Grafo de flujo
<b>SECUENCIAL</b> Instrucción 1 Instrucción 2 ... Instrucción n	
<b>CONDICIONAL</b> Si <Condición> Entonces <Instrucciones> Si No <Instrucciones> Fin si	
<b>IF condición AND</b>  <pre> 1   2 IF (A &amp;&amp; B) { 3 // código 4 } </pre>	<b>IF condición OR</b>  <pre> 1   2 IF (A    B) { 3 // código 4 } </pre>
<b>WHILE</b> Mientras <Condición> Hacer <Instrucciones> Fin Mientras	
<b>DO-WHILE</b> Hacer <Instrucciones> Mientras <Condición>	

## CONDICIONAL MÚLTIPLE

Según sea <Variable> Hacer

Caso opción 1:

<Instrucciones>

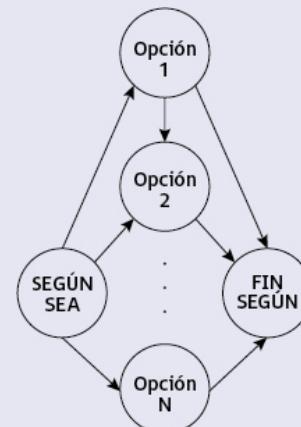
Caso opción 2:

<Instrucciones>

Caso opción 3:

<Instrucciones>

Fin Según



Cada círculo va a representar una o más sentencias, sin bifurcaciones, en pseudocódigo o código fuente. Se numerarán en el diagrama de flujo cada símbolo y los finales de las estructuras, aunque no tengan ningún símbolo.

En el grafo de flujo, cada círculo se llamará *nodo*. Va a representar a una o más sentencias procedimentales.

Las *flechas* del grafo de flujo se llaman *aristas* o *enlaces* y representan el flujo de control. Terminarán en un nodo, aunque este no tenga ninguna sentencia procedural.

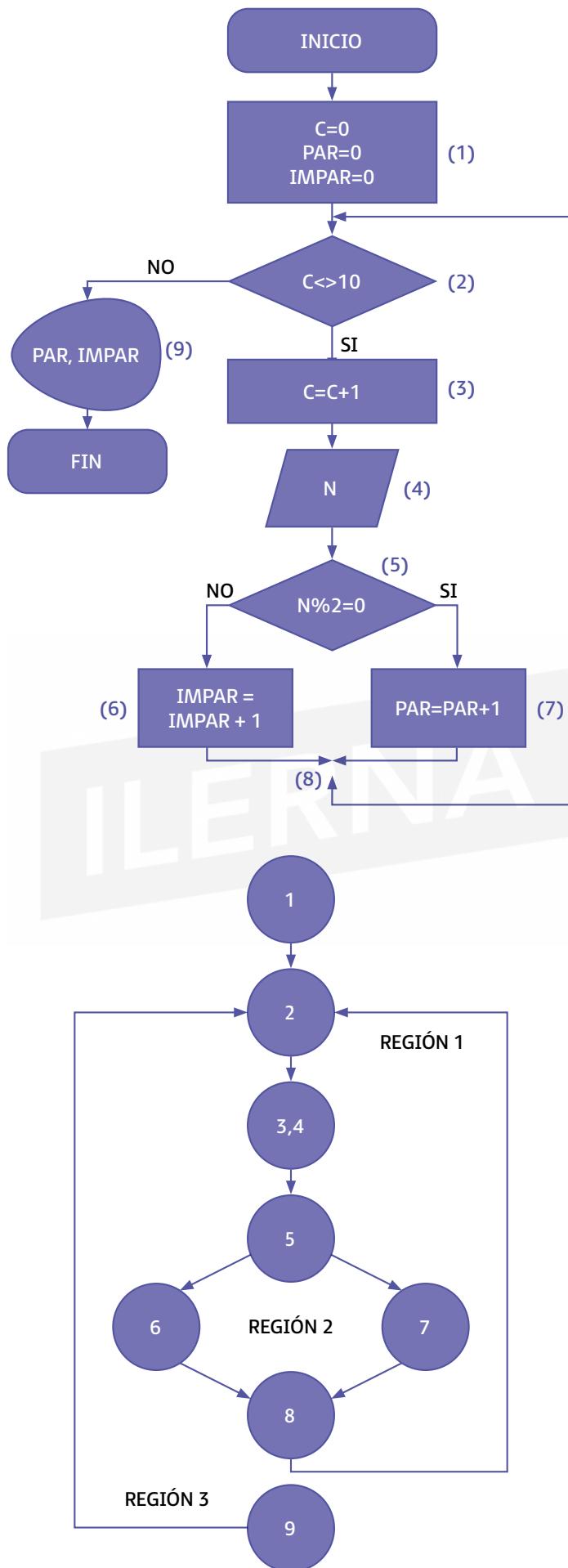
Las *regiones* son áreas que estarán delimitadas por aristas y nodos. Cabe destacar que el área exterior del nodo es otra región más.

El *nodo predicado* contendrá una condición y su principal característica es que salen dos aristas dependiendo de esa condición.

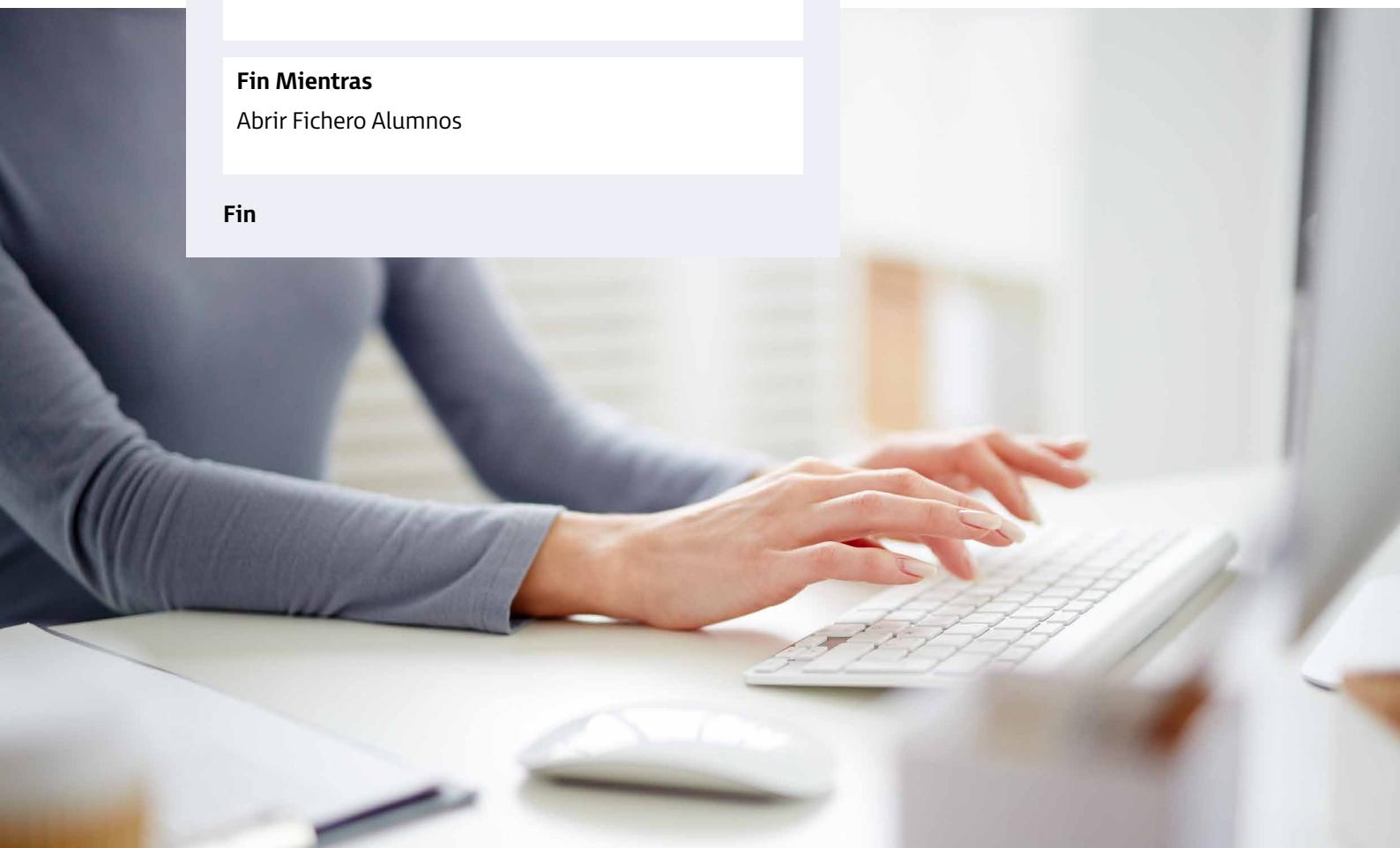
Vamos a ver estas representaciones en un ejemplo.

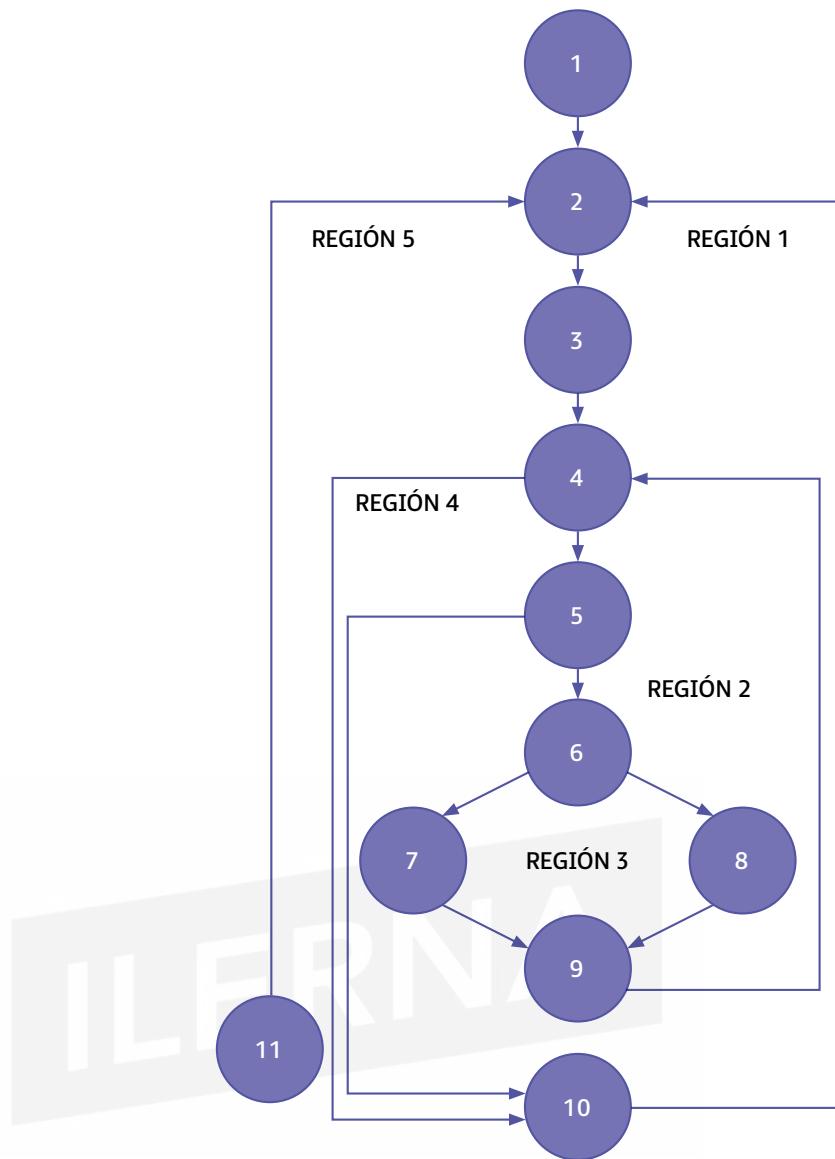
A continuación, tenemos un diagrama de flujo. Vamos a analizar cómo se desarrolla este código:

- (1)** En este nodo, tenemos un conjunto de sentencias de inicialización de variables.
- (2)** Tenemos un nodo decisión. Si C es distinta de 10, iremos al nodo (3), pero si no, iremos al nodo (9), sacaremos por pantalla las variables par e impar y finalizaremos el programa.
- (3) y (4)** Si se cumple que C, ejecutamos las sentencias secuenciales (3) y (4).
- (5)** Nos encontramos con otro nodo predicado. Si no se cumple que n sea par, incrementamos la variable impar (6). Si se cumple, incrementamos la variable par (7).
- (8)** Finalmente, volveremos al nodo predicado (2) para comprobar la condición.



Veamos otro ejemplo respecto del siguiente pseudocódigo:





Como vemos, podemos realizar las pruebas en diferentes representaciones de código (pseudocódigo, diagramas de flujo).

### Complejidad ciclomática

Métrica del software que nos proporciona una medida cuantitativa de la complejidad lógica de un programa. Nos establecerá el número de casos de prueba que deberán ejecutarse para que las sentencias sean ejecutadas al menos una vez.

La complejidad ciclomática  $V(G)$  se podrá calcular de tres formas:

1.  $V(G) = \text{número de regiones del grafo}.$
2.  $V(G) = \text{aristas} - \text{nodos} + 2.$
3.  $V(G) = \text{nodos predicado} + 1.$

Se establecerán los siguientes valores de referencia:

Complejidad ciclomática	Evaluación del riesgo
Entre 1 y 10	Programas o métodos sencillos, sin mucho riesgo.
Entre 11 y 20	Programas o métodos más complejos, riesgo moderado.
Entre 21 y 50	Programas o métodos complejos, alto riesgo.
Mayor que 50	Programas o métodos no testeables, muy alto riesgo.

Esta complejidad ciclomática determina cuál es la cota inferior del número de pruebas que tenemos que realizar para probar, como mínimo, todos los caminos existentes pasando, al menos, una vez por cada nodo y una vez por cada arista del grafo.

### Obtención de los casos de prueba

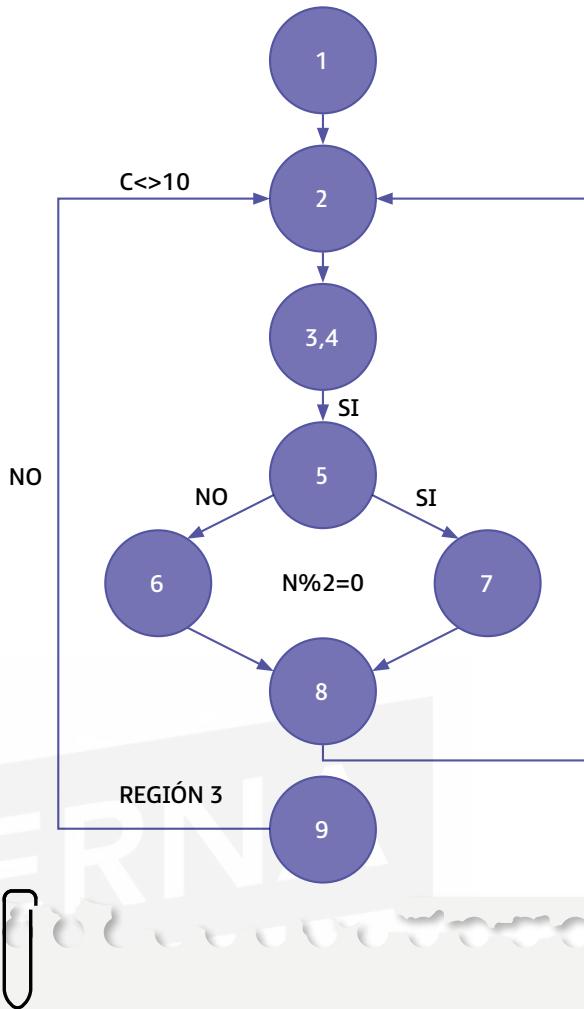
Este será el último paso de la prueba del camino básico y consistirá en construir los casos de prueba que fuerzan la ejecución de cada camino. Para comprobar cada camino, escogeremos los casos de prueba de tal forma que las condiciones de los nodos predicho estén establecidas adecuadamente. Respecto al primer ejemplo que habíamos realizado, representamos los casos de prueba como nos muestra la siguiente tabla:

Camino	Caso de prueba	Resultado esperado
1	Escoger algún valor de C tal que NO se cumpla la condición $C <> 10$ $C=10$	Visualizar el número de pares y de impares
2	Escoger algún valor de C tal que SÍ se cumpla la condición $C <> 10$ Escoger algún valor de N tal que NO se cumpla la condición $N \% 2 = 0$ $C=1, N=5$	Contar números impares
3	Escoger algún valor de C tal que SÍ se cumpla la condición $C <> 10$ Escoger algún valor de N tal que SÍ se cumpla la condición $N \% 2 = 0$ $C=2, N=4$	Contar números pares

Camino 1: 1-2-3-4-5-6-8-2-9

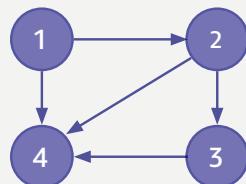
Camino 2: 1-2-3-4-5-7-8-2-9

Camino 3: 1-2-9



**ponte a prueba**

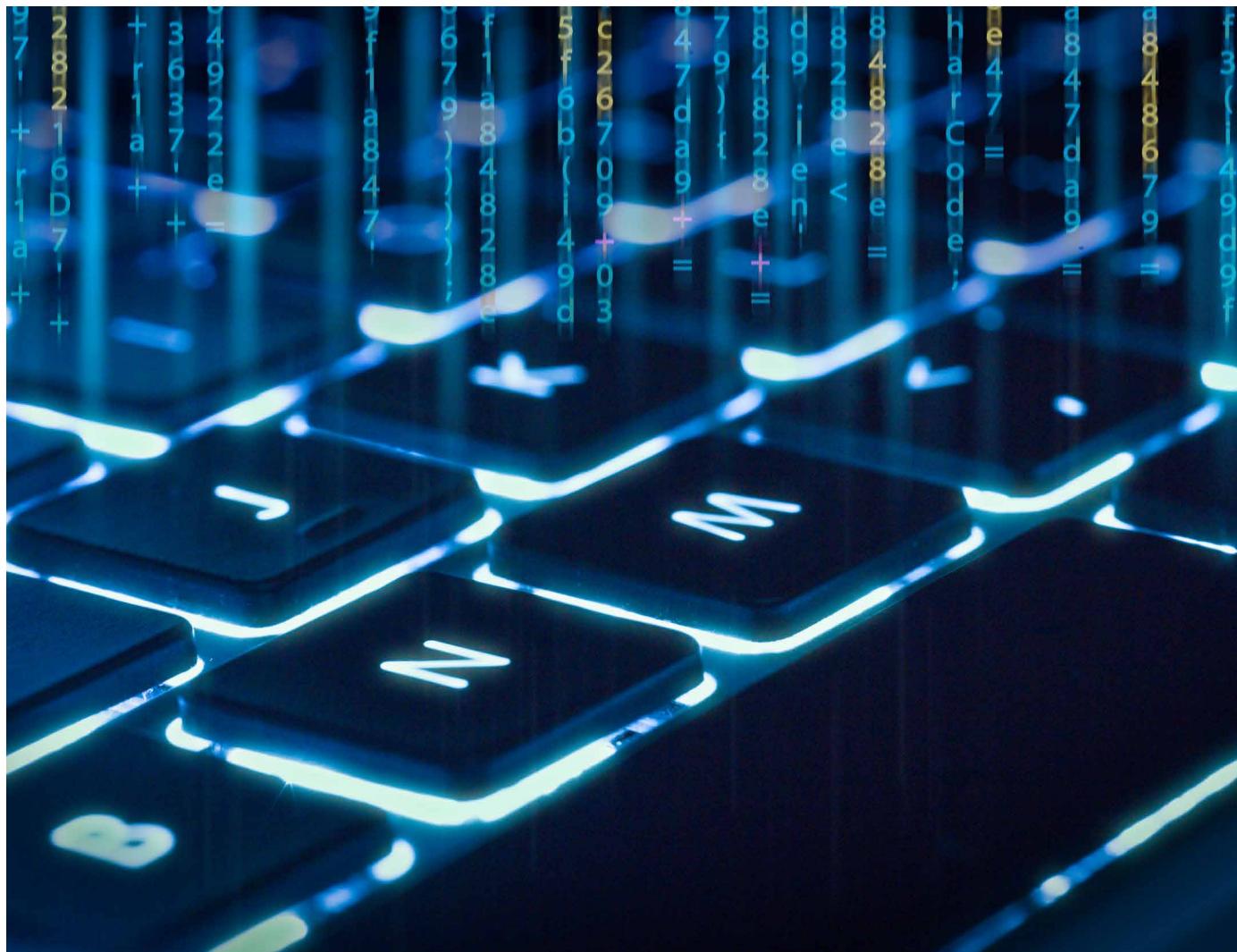
Según el siguiente grafo, ¿cuántas regiones tiene?



- a) 2
- b) 3
- c) 4
- d) Ninguna de las repuestas es correcta

Aquellos programas con una complejidad mayor de 50 son programas de alto riesgo y poco testeables.

- a) Verdadero
- b) Falso



### 3.3.2. PARTICIÓN O CLASES DE EQUIVALENCIA. CASOS DE PRUEBA

La partición equivalente es un método de prueba de caja negra que divide los valores de los campos de entrada de un programa en clases de equivalencia.

Examinaremos cada condición de entrada para poder identificar estas clases de equivalencia y lo dividiremos en dos o más grupos.

Se podrán definir dos tipos de clases de equivalencia:

- **Clases válidas:** valores de entrada válidos.
- **Clases no válidas:** valores de entrada no válidos.

Nuestro objetivo es encontrar un conjunto de entradas válidas (lo mismo para un conjunto de entradas inválidas), probarlo en el módulo que queremos testear y obtener una serie de salidas para tratar de comprobar o detectar si se ha encontrado un error.

Si analizamos este tipo de pruebas, nos damos cuenta de que asumimos riesgos, porque solo estamos probando con un conjunto de valores que damos por sentado que van a ser los representativos

Por ejemplo:

<b>Tipo de entrada</b>	<b>N.º de clases válidas</b>	<b>N.º de clases inválidas</b>
Rango de valores [10...20]	1: valor en rango [15]	2: valor por debajo del rango y valor por encima del rango (8 y 30)
Conjunto de valores {1,3,5,7}	1: valor de conjunto {3}	2: valor fuera del conjunto, tanto por arriba como por abajo (0 y 10)
Condición booleana: "debe ser una letra"	1: valor evaluado como cierto ('T')	2: valor evaluado como falso (\$)
Conjunto de valores admitidos {valor1, valor2, valor3}	1: tantos valores como los admitidos {valor1, valor2, valor3}	2: valor no admitido {valor4}
Clases menores. Dos rangos $8 < \text{valor} < 15$ ; $100 \leq \text{valor} \leq 200$ ;	Se divide en distintas subclases que se manejan exactamente igual que los puntos mencionados arriba (una para $8 < \text{valor} < 15$ y otra para $100 \leq \text{valor} \leq 200$ )	



ponte a prueba

**¿Cuáles son los dos tipos en los que podemos dividir las clases de equivalencia?**

- a) Aristas y nodos
- b) Válidas y no válidas
- c) Superclases y subclases
- d) Nodos predicados y aristas

### 3.3.3. ANÁLISIS DE VALORES LÍMITE. CASOS DE PRUEBA

Vamos a extender las pruebas de clases de equivalencia a un análisis de valores límite. Este análisis se basa en la hipótesis de que suelen ocurrir más errores en los valores extremos de los campos de entrada. Además, no solo estará centrado en las condiciones de entrada, sino que se definen también las clases de salida.

Tendrá las siguientes reglas:

1. Si una condición de entrada especifica un **rango de valores**, deberemos concretar casos de prueba para los límites del rango y para los valores justo por encima y por debajo. Ejemplo: para un rango de valores enteros que estén comprendidos entre 5 y 15, tenemos que escribir casos de prueba para 5, 15, 4 y 16.
2. Si especifica **número de valores**, similar al anterior.
3. Para la condición de salida, aplicaremos la regla 1.
4. Usar también para la condición de salida la regla 2. Tanto en esta regla como en la anterior no se generarán valores que estén fuera del rango.
5. Si la estructura interna posee límites preestablecidos, nos aseguraremos de diseñar casos de prueba que ejerzan la estructura de datos en sus límites, primer y último elemento.



Vamos a ver un ejemplo mezclando ambos tipos de pruebas:

 **EJEMPLO**

Imaginemos que tenemos una aplicación bancaria con los siguientes datos de entrada:

- Código: número de tres dígitos y que no puede empezar ni por 0 ni por 1.
- Clave identificativa: 6 caracteres alfanuméricos.
- Conjunto de órdenes: "cheque", "retirada de fondos", "depósito".

Parámetro de entrada	Regla a aplicar	Clases válidas	Clases no válidas
Código	Rango de valores [200...999]	Código=200 Código=201 Código=998 Código=999	Código=199 Código=1.000
Clave	Conjunto finito de 6 caracteres alfanuméricos	6 caracteres exactos	5 caracteres 7 caracteres
Órdenes	Conjunto de valores admitidos	"cheque", "retirada de fondos", "depósito"	No es una orden válida



ponte a prueba

**Si estamos testeando un módulo que tiene de rango de entradas [0-5], ¿qué valores deberíamos probar?**

- 0 y 5
- 1, 0, 5, 6
- 0
- 5

## 3.4. PRUEBAS FUNCIONALES, PRUEBAS ESTRUCTURALES Y PRUEBAS DE REGRESIÓN

Las pruebas funcionales se definen a partir de las características que están descritas en la documentación y su compatibilidad con el sistema. Podemos ejecutar estas pruebas a cualquier nivel: componentes, integración, sistema, etcétera.

Son pruebas de caja negra porque valoramos el comportamiento externo del sistema.

Un ejemplo de prueba funcional es una prueba unitaria o una prueba de interfaz.

Las pruebas estructurales son pruebas que nos indican si las diferentes estrategias de pruebas de software se han llevado a cabo correctamente. Si todas estas pruebas encajan, podemos decir que la arquitectura construida de software es correcta.

Las pruebas de regresión consisten en volver a probar una parte de sistema o un componente determinado tras haber sido modificado. Se realizan este tipo de pruebas con el objetivo de descubrir cualquier fallo introducido o que no hayamos cubierto debido a esos cambios.

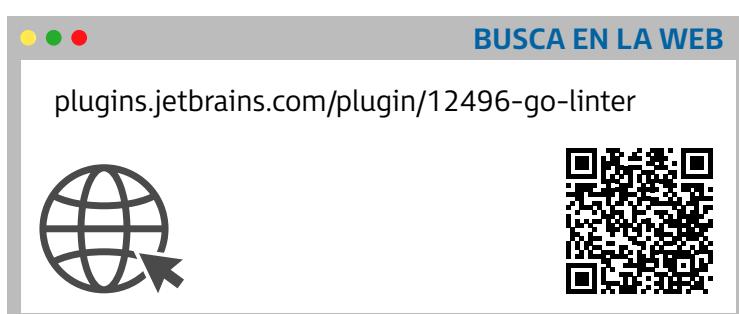
Por tanto, estas pruebas se realizan sobre componentes ya probados.

## 3.5. HERRAMIENTAS DE DEPURACIÓN DE CÓDIGO

En esta sección, hablaremos de las herramientas *linter*. Estas herramientas nos ayudan a detectar código confuso o incompatible, es decir, errores de programación del análisis sintáctico que realiza nuestro compilador.

Suelen ser herramientas que nos detectan fallos, como un uso incorrecto de variables, condiciones o bucles mal formulados o cálculos erróneos (veremos qué es un *bad smell* más adelante).

Puedes descargar esta herramienta desde la página oficial de **JetBrains**:





## 3.6. CALIDAD DEL SOFTWARE

### 3.6.1. NORMAS Y CERTIFICACIONES

La **norma ISO/IEC 25000**, conocida como SQuaRE (Software Product Quality Requirements and Evaluation), crea un conjunto de reglas comunes para evaluar la calidad del producto de software.

#### Familia de normas ISO/IEC 25000:

- ISO/IEC 2500n. División de Gestión de Calidad.
- ISO/IEC 2501n. División de Modelo de Calidad.
- ISO/IEC 2502n. División de Medición de Calidad.
- ISO/IEC 25030n. División de Requisitos de Calidad.
- ISO/IEC 25040n. División de Evaluación de Calidad.
- ISO/IEC 25050–25099. Estándares de extensión SQuaRE.

Se pueden consultar con más detalle esta normativa en las siguientes webs:

BUSCA EN LA WEB

1. [iso25000.com/](http://iso25000.com/)  
2. [www.iso.org/standard/64764.html](http://www.iso.org/standard/64764.html)

 1  2 



### 3.6.2. MEDIDAS DE CALIDAD DEL SOFTWARE

#### Optimización de código

<https://youtu.be/w-RuXx4mFMs>



Al principio de los años ochenta, Maurice Halstead<sup>1</sup> desarrolla un conjunto de normas y métricas que se sustentan en el cálculo de palabras clave y variables del código.

Esta métrica está basada en contar los operadores y operandos que hay en un programa:

- $n_1$ : número de operadores únicos que aparecen en un programa.
- $n_2$ : número de operandos únicos que aparecen en un programa.
- $N_1$ : número total de ocurrencias de operadores.
- $N_2$ : número total de ocurrencias de operandos.

A partir de aquí, podemos realizar diferentes cálculos en nuestros códigos:

- Longitud:  $N = N_1 + N_2$ . Es la medida del tamaño del programa. Cuanto mayor sea  $N$ , más complejo es el programa y, por tanto, mayor esfuerzo hay que dedicar al mantenimiento. Esta medida nos detalla mejor el tamaño de nuestro código que el simple conteo de líneas de código.
- Volumen:  $N * \log_2(n)$ , donde  $n = n_1 + n_2$ . Si, por ejemplo, tenemos dos programas con la misma longitud ( $N$ ), pero uno de ellos tiene mayor número de operandos y opera-

<sup>1</sup> Institute of Electrical and Electronics Engineers Inc (1995). *Computer pioneers by J. A. N. Lee*. <https://history.computer.org/pioneers/halstead.html>

dores únicos, será más difícil de entender y mantener. Por tanto, tendrá mayor volumen.

- Dificultad:  $D = ((n_1 * N_2) / (n_2 * 2))$ . La dificultad es directamente proporcional al número de operadores únicos de un programa. Si los mismos operandos se utilizan varias veces dentro del programa, será más propenso a tener errores.
- Esfuerzo:  $E = V * D$ . El esfuerzo por entender un programa o mantenerlo es proporcional al volumen del programa y a su nivel de dificultad.
- Nivel:  $L = 1 / D$ . El nivel de un programa es inversamente proporcional a su dificultad. Un programa de bajo nivel es más propenso a tener errores que uno de alto nivel.

Veamos estas métricas con un ejemplo:

```

if (N > 1) {
A = B * N;
System.out.println("El resultado es: " + A);
}
n1 = 8 (if, (), >, {}, =, *, system.out.println, +)
n2 = 4 (N, 1, A, B)
n = n1 + n2 = 12
N1 = 9 (if, (), >, {}, =, *, system.out.println, (), +)
N2 = 6 (N, 1, A, B, N, A)
N = N1 + N2 = 15
V = N * log2(n) = 15 * log2(12) = 53,77
D = (n1 * N2) / (n2 * 2) = (8 * 6) / (4 * 2) = 6
E = V * D = 54,77 * 6 = 328,62
L = 1 / D = 1 / 6 = 0,16

```

Otro tipo de métricas sirven para valorar el mantenimiento de nuestro producto:

- Número de solicitudes para el mantenimiento correctivo: si hay un aumento en el reporte de fallos, quiere decir que se han introducido más errores que los que se han arreglado. Por lo tanto, el mantenimiento será más costoso.
- Tiempo requerido para hacer un análisis del producto: indica el número de componentes del sistema que se ven afectados. Si el número es alto, el mantenimiento será costoso.
- Tiempo requerido para poder implementar los cambios: tiempo para modificar el aplicativo y su documentación. Si este tiempo es alto, el mantenimiento será más costoso.
- Número de peticiones de cambio: un aumento de esas solicitudes hará de nuestro aplicativo muy difícil de mantener.

ponte a prueba

**Una de las métricas de Halstead es la medida del esfuerzo.**

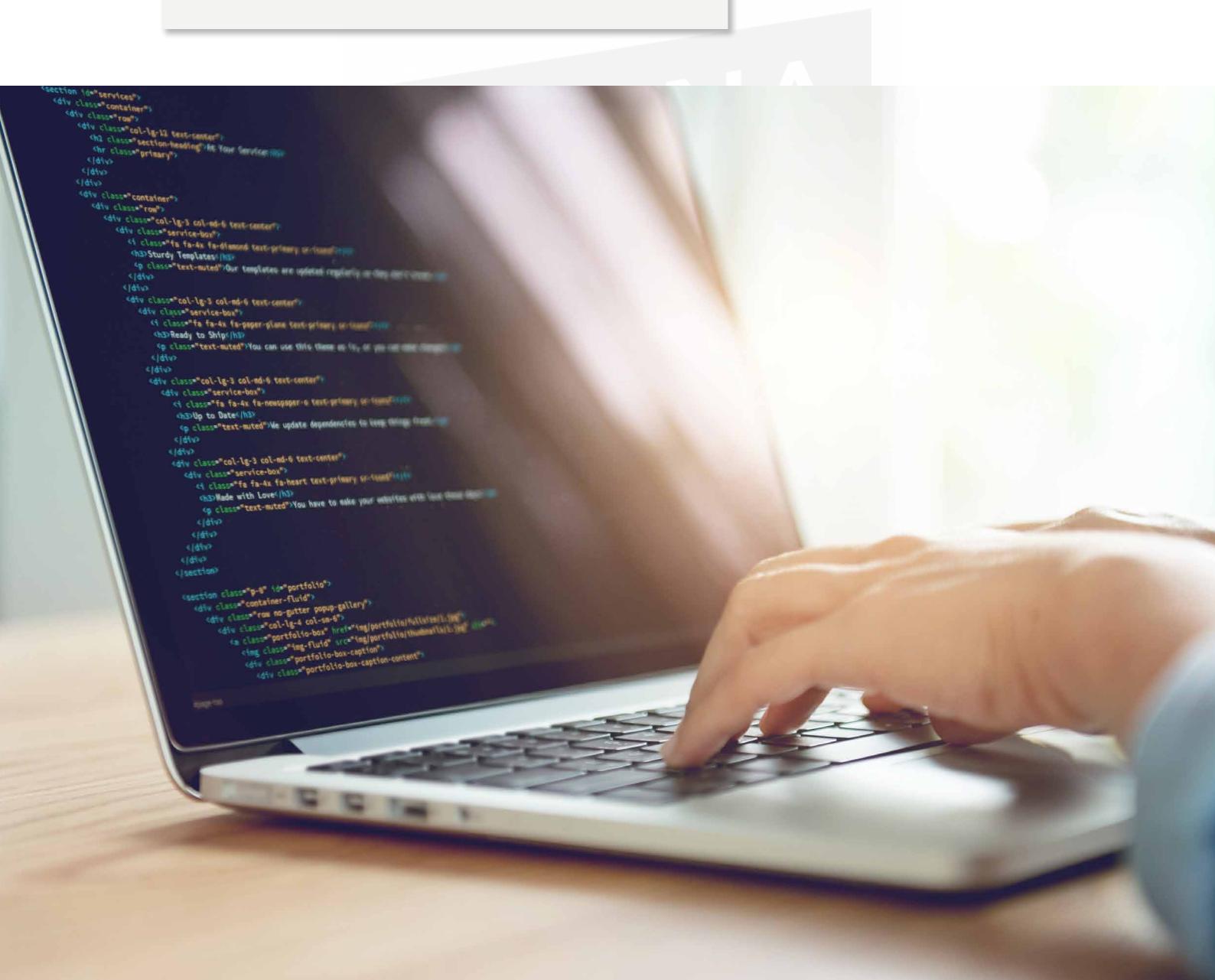
- a) Verdadero
- b) Falso

**La fórmula  $N = N_1 + N_2$ , donde  $N_1$  es el número total de operadores y  $N_2$  es el número total de operandos, ¿qué calcula?**

- a) La dificultad de un código
- b) El volumen de un código
- c) El esfuerzo de un código
- d) La longitud de un código

```
<section id="services">
  <div class="container">
    <div class="row">
      <div class="col-lg-12 text-center">
        <h2 class="section-heading">Our Services</h2>
        <hr class="primary">
      </div>
    </div>
    <div class="container">
      <div class="row">
        <div class="col-lg-3 col-md-6 text-center">
          <div class="service-box">
            <i class="fa fa-diamond text-primary fa-4x"></i>
            <h3>Sturdy Templates</h3>
            <p>Our templates are updated regularly as they are used.</p>
          </div>
        </div>
        <div class="col-lg-3 col-md-6 text-center">
          <div class="service-box">
            <i class="fa fa-paper-plane text-primary fa-4x"></i>
            <h3>Ready to Ship!</h3>
            <p>You can use this theme as is, or you can alter it to your needs.</p>
          </div>
        </div>
        <div class="col-lg-3 col-md-6 text-center">
          <div class="service-box">
            <i class="fa fa-newspaper-o text-primary fa-4x"></i>
            <h3>Up to Date!</h3>
            <p>We update dependencies to keep things fresh.</p>
          </div>
        </div>
        <div class="col-lg-3 col-md-6 text-center">
          <div class="service-box">
            <i class="fa fa-heart text-primary fa-4x"></i>
            <h3>Made with Love!</h3>
            <p>You have to take your website seriously, but this theme doesn't.</p>
          </div>
        </div>
      </div>
    </div>
  </div>
</section>
```

.../index.html





# 4 DOCUMENTACIÓN Y OPTIMIZACIÓN

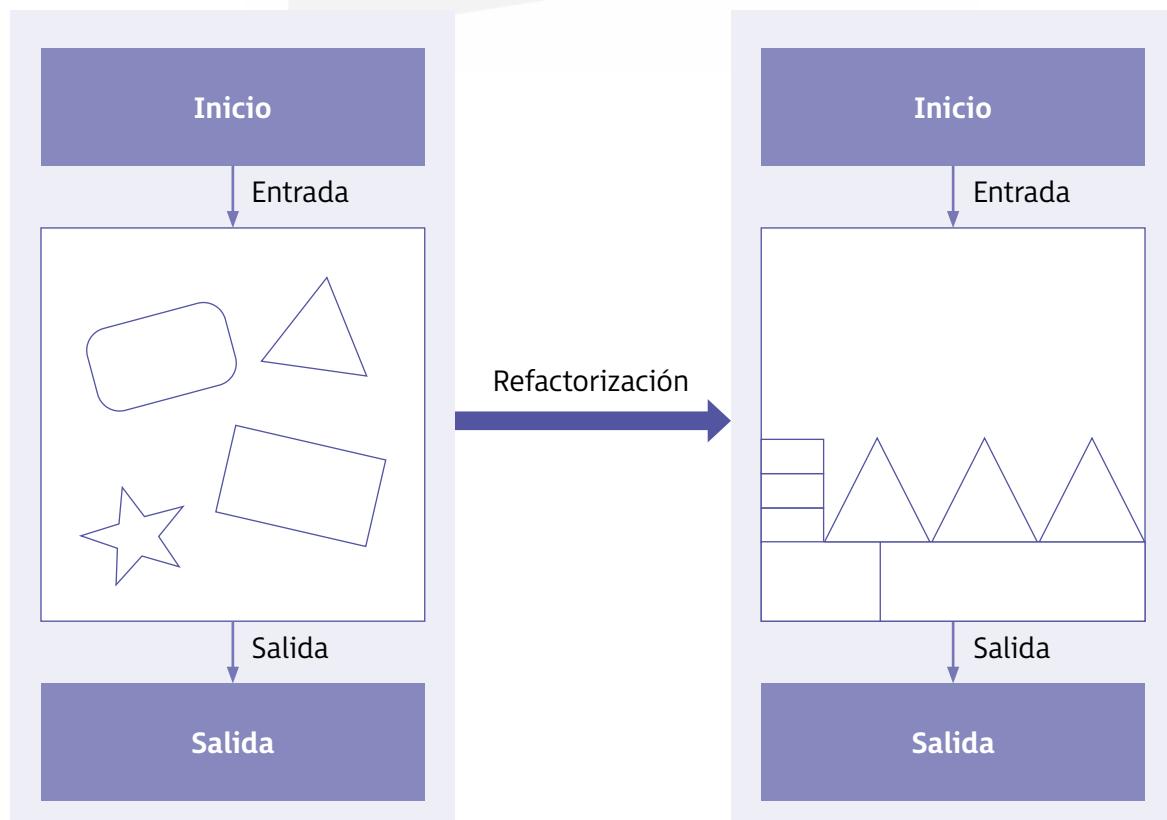
En esta segunda parte, abordaremos el concepto de refactorización y por qué es necesario refactorizar. Trataremos el control de versiones y para qué se utiliza en la creación de proyectos de software. Y, por último, usaremos herramientas cliente y servidor y aprenderemos a documentar las clases Java usando Javadoc.

## 4.1. REFACTORIZACIÓN. CONCEPTO. LIMITACIONES. PATRONES DE REFACCIÓN MÁS USUALES

### CONCEPTO

La **refactorización** nos va a permitir optimizar un código que se ha escrito previamente, realizando cambios en la estructura interna sin que afecten al comportamiento final del producto.

La refactorización tiene como objetivo limpiar el código para que se entienda mejor y se pueda modificar de forma más fácil, lo que nos va a permitir una mejor lectura y comprensión de lo que se realiza. Esta modificación no alterará su ejecución ni los resultados.



### 4.1.1. CUÁNDO REFACTORIZAR. *BAD SMELLS*

La refactorización la deberemos ir haciendo a medida que desarollamos el software. En 2003, Mario G. Piattini<sup>1</sup> y Félix Óscar García analizaron los síntomas que indican la necesidad de refactorizar. Por su parte, el ingeniero de software británico Martin Fowler y otros expertos diagnosticaron los ***bad smells*** (malos olores), es decir, pequeños indicios que indican que el sistema no funciona como es debido. Los síntomas para refactorizar el código son los siguientes:

- **Código duplicado** (*duplicated code*). Esta será la principal razón para realizar la refactorización. Si encontramos algún código repetido, deberemos unificarlo.
- **Métodos muy largos** (*long method*). Los métodos largos normalmente pueden estar compuestos de métodos más pequeños, por lo que deberemos dividirlos para que, además, puedan reutilizarse.
- **Clases muy grandes** (*large class*). Si una clase es grande, tendrá muchas responsabilidades al tener demasiados métodos y atributos. Por ello, deberemos crear clases más pequeñas y que estén bien delimitadas.
- **Lista de parámetros extensa** (*long parameter list*). Las funciones deben tener el mínimo de parámetros posible o, del contrario, tendremos un problema de encapsulación de datos. Si un método requiere de muchos parámetros, deberemos crear una clase objeto con esa cantidad de datos.
- **Cambio divergente** (*divergent change*). Una clase se puede modificar por diferentes motivos. Estos no tienen por qué estar relacionados y cabe la posibilidad de poder eliminar o dividir dicha clase en el caso, por ejemplo, de que esté realizando demasiadas tareas.
- **Cirugía a tiro pistola** (*shotgun surgery*). Cambios adicionales realizados después de modificar una clase para compatibilizar el cambio.
- **Envidia de funcionalidad** (*feature envy*). Ocurre cuando un método usa más elementos de otra clase que de la suya propia. Se resolverá pasando ese método a la clase que usa más.

---

1 R. Villarroel, E. Fernández-Medina, J. Trujillo, M. Piattini. (2005) *Un profile de UML para diseñar almacenes de datos seguros*. [https://www.researchgate.net/profile/Mario\\_Piattini/publication/3454984\\_A\\_UML\\_profile\\_for\\_designing\\_secure\\_data\\_warehouses/links/0deec5391f6203f2a500000.pdf](https://www.researchgate.net/profile/Mario_Piattini/publication/3454984_A_UML_profile_for_designing_secure_data_warehouses/links/0deec5391f6203f2a500000.pdf)

- **Clase de solo datos** (*data class*). Clase que solo tiene atributos y métodos de acceso. No debería ser lo habitual.
- **Legado rechazado** (*refused bequest*). Subclases que usan características de superclase, lo que puede inducir a un error en la jerarquía de clases.

El proceso de refactorización posee algunas ventajas, entre las que están la sencillez de mantenimiento en el diseño del sistema y el incremento de la facilidad en la lectura y en el código fuente.

Las bases de datos y las interfaces son áreas conflictivas para la refactorización. El cambio de base de datos tendría como consecuencia la migración de la estructura y de los datos.



### ponte a prueba

**Es mejor realizar un método o clase lo más extenso posible para cubrir todos los posibles casos y pruebas.**

- Verdadero
- Falso

**Si tenemos que realizar un cambio en un módulo debido a que cambian los requisitos y este cambio afecta a todos los módulos de sistema, ¿qué *bad smell* encontramos?**

- Cirugía a tiro de pistola
- Código duplicado
- Cambio divergente
- Ninguna de las opciones es la correcta

01

```

        error_mod.use_y = False
        error_mod.use_z = True

[ (1+x+y+2a)-(3a+3g+x) ] selection at the end -ad
    ob.select= 1
    error_mod.select=1
    E=mc2context.scene.objects.active
    "Selected" + str(modifier)
    error_mod.select = 0 1lim h>0
    bpy.context.selected_obi
    obta.objects[one.name].se
    1lim h>0

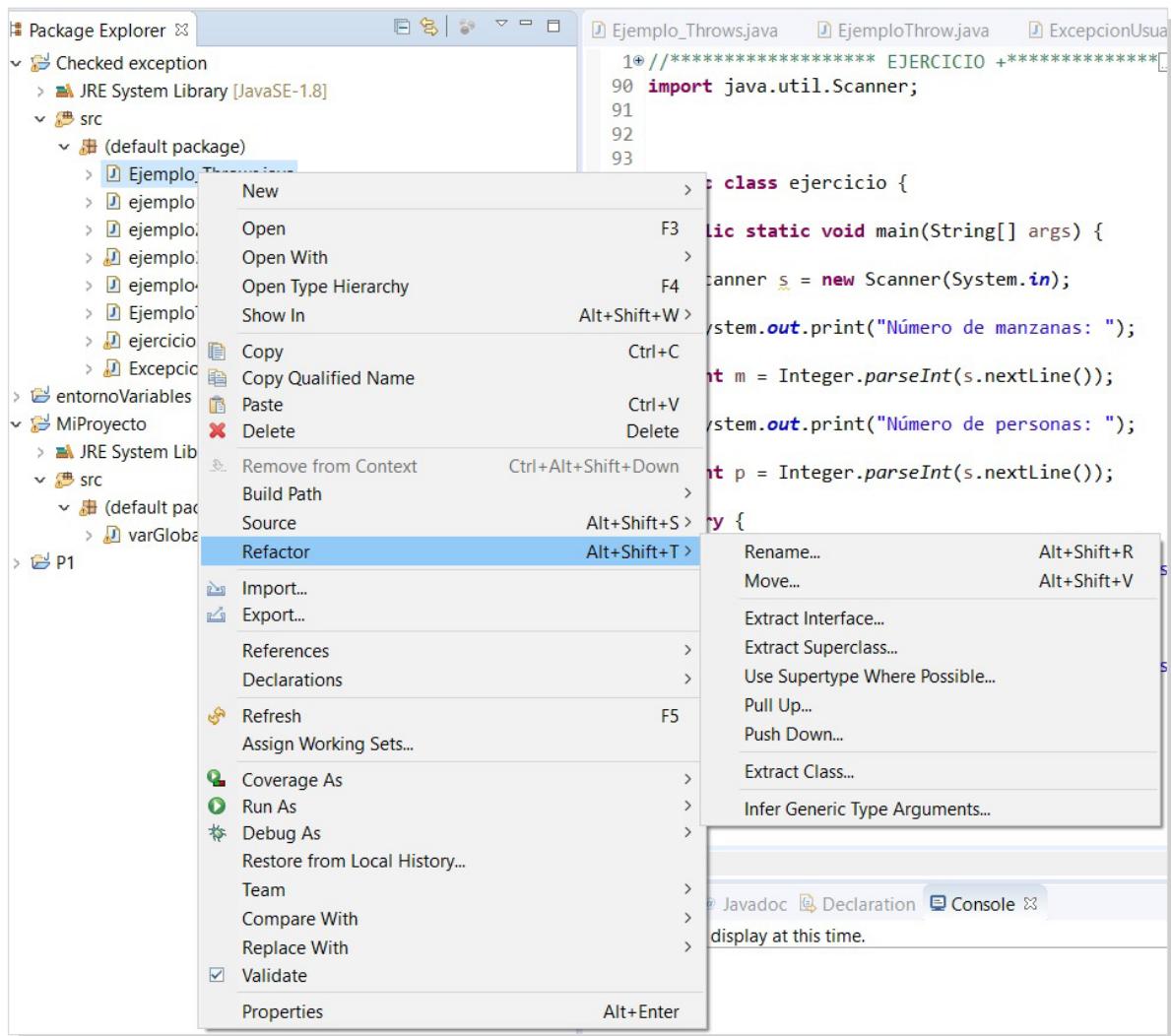
2+....+2a....+a 1+x+y+2a
    1+x+y+2a+21 int("please select exact")
    1lim h-->0 2+....+2a+....+a
    OPERATOR CLASSES 45 4a 3
    1+x+y+2a+21 {x-12-y+n...}
    1lim h-->0 {x-12-y+n...}
    x=0 xn {x-12-y+n...}

    types.Operator):
        X mirror to the selected
        object_mirror_mirror_x"
        for X" (1+x+y+2a)-(3a+3g+x)
        1+x+y+2a+21
        error [ 5+x+k+2a+21
        is not
    
```

## 4.1.2. REFACTORIZACIÓN EN ECLIPSE

Con el IDE Eclipse, podemos refactorizar nuestro código.

Podemos ir a la clase que queremos factorizar, clicamos con el botón derecho del ratón y nos saldrá un menú dentro del cual tendremos la opción factorizar.



### Métodos y herramientas de refactorización

Mediante el uso de distintas herramientas plantearemos elementos para refactorizar y estas nos mostrarán las posibles soluciones, en las que podremos observar el resultado antes y después de la refactorización. También se les llama **patrones de refactorización** o **catálogos de refactorización**.

Para refactorizar, seleccionamos el elemento y pulsamos el botón derecho del ratón, elegimos **Refactor** y seleccionamos **Método de Refactorización**. Los elementos más comunes serán los siguientes:

- **Rename.** Cambia el nombre de cualquier identificador de Java. Es de las opciones más utilizadas y, una vez ejecutada, se modifican las referencias a ese identificador.
- **Move.** Se mueve la clase de un paquete a otro, se moverá el archivo .java y se cambiarán todas las referencias.
- **Extract Interface.** Nos va a permitir escoger los métodos de una clase para crear una *interface*. Una interfaz es una plantilla que define los métodos, pero no los desarrolla. Serán las clases de la interfaz la encargada de desarrollarlos.
- **Extract Superclass.** Permite extraer una superclase. Si ya utilizaba una, la extraída será la nueva superclase. Se podrán seleccionar los métodos y atributos que van a formar parte de la nueva superclase.
- **Pull Up.** Permite pasar un método de una subclase (o clase hija) a una superclase.
- **Pull Down.** Permite pasar un método de una superclase a una clase hija o subclase.
- **Extract Constant.** Convierte en una constante un número o una cadena. Se mostrará el estado antes y después de refactorizar. El objetivo es modificar el valor del literal en un único lugar.
- **Extract Local Variable.** Se asigna una expresión a una variable local. La misma expresión a otro método no se modifica.
- **Convert Local Variable to Field.** Convierte una variable local en un atributo privado de la clase. Tras la refactorización, todos los usos de la variable local se sustituyen por el atributo.
- **Extract Method.** Convierte un bloque de código en un método. No debe llevar llaves abiertas. Este patrón es muy útil cuando detectamos *bad smells* en métodos muy largos o en bloques de código que se repiten.
- **Change Method Signature.** Permite cambiar la firma de un método, es decir, el nombre y los parámetros que tiene.
- **Inline.** Nos ajusta una referencia a una variable o método con la línea en la que se utiliza para conseguir así una única línea de código.
- **Member Type to Top Level.** Permite convertir una clase anidada en una clase de nivel superior con su propio archivo de Java.
- **Convert Anonymous Class to Nested.** Permite convertir una clase anónima a una clase anidada de la clase que la contiene. Una clase anónima se caracteriza por:



- Utilizar la palabra *new* seguida de la definición entre llaves.
- Usar la palabra *new* seguida del nombre de la clase que hereda (sin *extends*) y la definición de la clase entre llaves.
- Utilizar la palabra *new* seguida del nombre de la interfaz (sin *implements*) y la definición de la clase anónima entre llaves.

Los aspectos más importantes que se deben tener en cuenta para refactorizar son:

- Se reestructura el código sin que haya cambios significativos en él.
- Los IDE (como Eclipse) soportan la refactorización. El IDE conoce las relaciones entre los métodos y las clases, permitiendo hacer cambios.
- Antes de realizar una refactorización, hay que definir correctamente los casos de prueba.

### 4.1.3. REFACTORIZACIÓN Y PRUEBAS

Para poder realizar la factorización, es muy importante contar con una buena colección de casos de prueba que nos validen nuestro sistema:

- Es recomendable que sean casos de prueba automáticos.
- Reportar los casos de fallo y errores que se encuentren.
- Deben poder ser ejecutados de forma independiente.

Lo primero que debemos hacer a la hora de refactorizar es ejecutar las pruebas para comprobar que, después de refactorizar, el sistema no haya cambiado y tengamos los mismos resultados. Después habría que analizar los cambios que se quieren aplicar y, finalmente, aplicarlos.

Hay que destacar que optimizar el código no es refactorizar. Es verdad que ambos métodos modifican el código sin cambiar su funcionalidad, pero optimizar suele implicar introducir dificultad al código.

#### Importancia de las pruebas a la hora de refactorizar

Las pruebas unitarias juegan un papel muy importante para el desarrollo del software, no solo por el hecho de encontrar errores en nuestro código, sino que:

- Nos previenen de las pruebas de regresión (recordemos que son pruebas que consisten en volver sobre el código cuando se hayan realizado cambios). Unas pruebas robustas en los primeros estados nos ayudan a reducir el tiempo en detección de errores.
- Facilitan refactorizar. Las pruebas unitarias nos dan seguridad suficiente para poder refactorizar y eliminar el riesgo de errores en el código.
- Mejoran cómo está diseñada la implementación. Si el código es complejo de probar, indica que hay que refactorizar.

#### Ventajas de refactorizar:

- Facilita la comprensión de nuestro código, sobre todo a aquellos programadores que se incorporaron al proceso posteriormente. Si el código fuente es complejo, su lectura reduce la productividad en el análisis de este.
- Favorece la detección de errores debido a que es más fácil de comprender y, por tanto, mejora su robustez.
- Provoca mayor rapidez en la programación.
- Al reducir la complejidad del código, la calidad se ve incrementada durante el proceso.

Las zonas más conflictivas a la hora de refactorizar son las interfaces y las bases de datos, por lo que refactorizar también presenta algunas desventajas:

- El aplicativo suele estar fuertemente cohesionado con el sistema de la base de datos, por lo que una migración de esta última implica cambios estructurales y de datos.
- El cambio de interfaz no tiene que conllevar problemas siempre y cuando se tenga acceso al código fuente. Si no, la refactorización sería problemática.

#### 4.1.4. PATRONES DE DISEÑO

Lo primero es definir qué es un patrón de diseño. Un patrón es una solución que se puede replicar en distintos desarrollos de software. Se consideran descripciones de cómo poder afrontar un problema en diferentes situaciones.

Estos patrones explican todos los problemas que se encuentran en el diseño y se debate cuál es la mejor solución.

Gracias a estos patrones, podemos identificar las clases del diseño, cómo interactúan entre ellas y la distribución de la carga de trabajo.

Esta es la estructura de un patrón:

- Nombre: descripción resumida del problema y las posibles soluciones.
- El problema: describimos el problema detalladamente teniendo en cuenta:
  - Los requisitos a la hora de hacer el aplicativo.
  - Las restricciones.
  - Qué tipo de propiedades y características debe tener la solución.
- La solución: no describe una solución implementada como tal, sino que es una plantilla para llenar en situaciones concretas. Nos proporciona una solución abstracta y general.
- Consecuencias: ventajas e inconvenientes.

Los tipos de patrones que nos podemos encontrar son:

- **Patrones estructurales:** nos detallan cómo se van a relacionar unos objetos con otros. Por ejemplo, el patrón *adapter* nos permite comunicar dos clases con interfaces diferentes a través de un objeto intermedio.
- **Patrones de comportamiento:** nos permiten detallar las responsabilidades entre los objetos. Un ejemplo sería el

patrón *iterator* nos permite movernos por elementos de forma secuencial sin necesidad de entrar en su implementación.

- **Patrones creacionales:** ayudan a la creación de nuevos objetos sin necesidad de entrar en la implementación del resto del aplicativo, como es el caso del patrón *singleton*, que delimita una a una el número de las instancias de una clase y concede un acceso global a todo nuestro sistema.

## 4.2. CONTROL DE VERSIONES. ESTRUCTURA DE LAS HERRAMIENTAS DE CONTROL DE VERSIONES

**Control de versiones**  
<https://youtu.be/JMnbRjde57Q>



El control de versiones implica la capacidad de poder recordar todos los cambios que se han realizado tanto en la estructura de directorios como en el contenido de los archivos. Puede ser muy **útil para recuperar carpetas, archivos** o algún proyecto en un momento dado del desarrollo. Es necesario saber qué cambios se hacen, quién los hace y cuándo se realizan.

Veamos algunos **términos útiles** con relación al manejo del control de versiones:

- **Repositorio.** Lugar donde se almacenan los datos y los cambios realizados.
- **Revisión o versión.** Una revisión es una versión concreta de los datos almacenados. La última versión se identifica como la cabeza o **HEAD**.
- **Etiquetar o rotular (tag).** Las etiquetas se crean para localizar o recuperar en cualquier momento una versión concreta del desarrollo.
- **Tronco (trunk).** Línea principal del desarrollo del proyecto.
- **Rama o ramificar (branch).** Copias de carpetas, archivos o proyectos. Se pueden crear ramas para la creación de nuevas funcionalidades o comprobación de errores.
- **Desplegar (checkout).** Copia del proyecto, archivos y carpetas en el repositorio del equipo local.
- **Confirmar (commit o check-in).** Se realiza cuando se confirman los cambios realizados en local para integrarlos al repositorio.

- **Exportación (export).** Es similar al *checkout*, pero no se vincula la copia con el repositorio.
- **Importación (import).** Subida de carpetas y archivos al repositorio.
- **Actualizar (update).** Se realiza cuando se desea integrar los cambios realizados en el repositorio de la copia del trabajo local.
- **Fusión (merge).** Se unen cambios realizados sobre uno o varios archivos en una única revisión. Se suele realizar cuando existen varias ramas y es necesario unir los cambios realizados.
- **Conflicto.** Suele ocurrir cuando un usuario hace un *checkout* de un archivo y otro usuario no actualiza y realiza cambios sobre el mismo archivo. Cuando envía los cambios realizados, existe un conflicto entre ambos archivos, por lo que se deberán realizar los cambios o elegir uno de ellos.
- **Resolver conflicto.** Actuación del usuario para atender varios conflictos.

Para trabajar con el control de versiones habrá que crear primero una copia local con *checkout*, realizar las modificaciones y, por último, subir las modificaciones con *commit*. Si ya está vinculada la copia, habrá que hacer *update* para que se haga sobre la última versión.

#### 4.2.1. REPOSITORIO

Durante el desarrollo de un proyecto, es fundamental el uso de una herramienta multiplataforma de código abierto que garantice el control de versiones. Esta herramienta usará una base de datos central llamada repositorio que contendrá archivos cuyas versiones e historias son controladas. Este repositorio actuará como servidor de archivos y recordará cada cambio realizado.

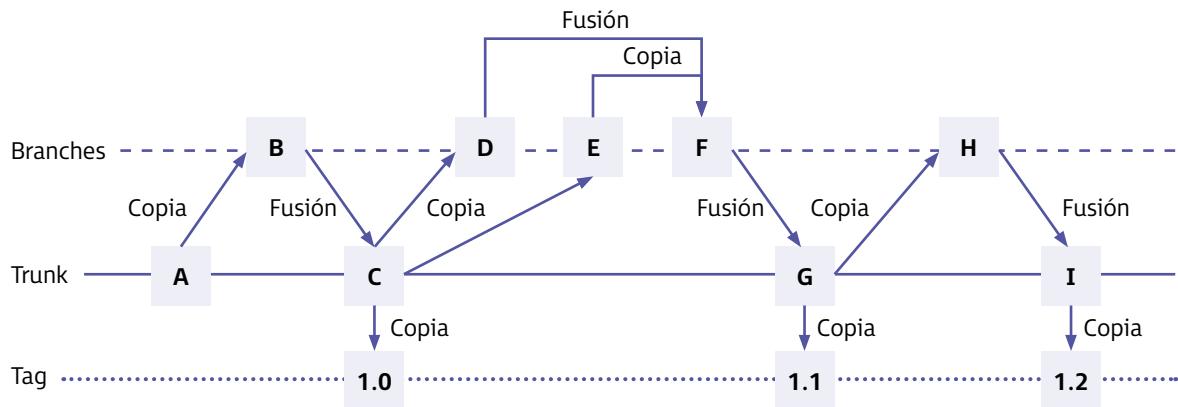
La herramienta de control de versiones es especialmente importante cuando un proyecto lo realizan varias personas, pues será básico llevar un control y un orden para el correcto desarrollo de este. El proyecto tendrá que verse como un árbol con su tronco (**trunk**), que será la línea principal; sus ramas (**branches**), las cuales añaden nuevas funciones o corrigen errores; y sus etiquetas (**tags**), que marcan situaciones importantes o versiones acabadas.

Así, la estructura con sus funciones quedará:

- **Trunk (tronco):** se guardan las carpetas del proyecto. Aquí estará la versión básica, o sea, la línea principal.
- **Tags (etiquetas):** copia del proyecto, carpeta o archivo para obtener una versión que no se modifique. Serán

copias del tronco y son útiles para crear versiones ya finalizadas.

- **Branches (ramas):** desarrolla versiones que serán publicadas. Es una copia del tronco que será modificada para conseguir un producto final distinto al original. Serán modificaciones de versiones cerradas.



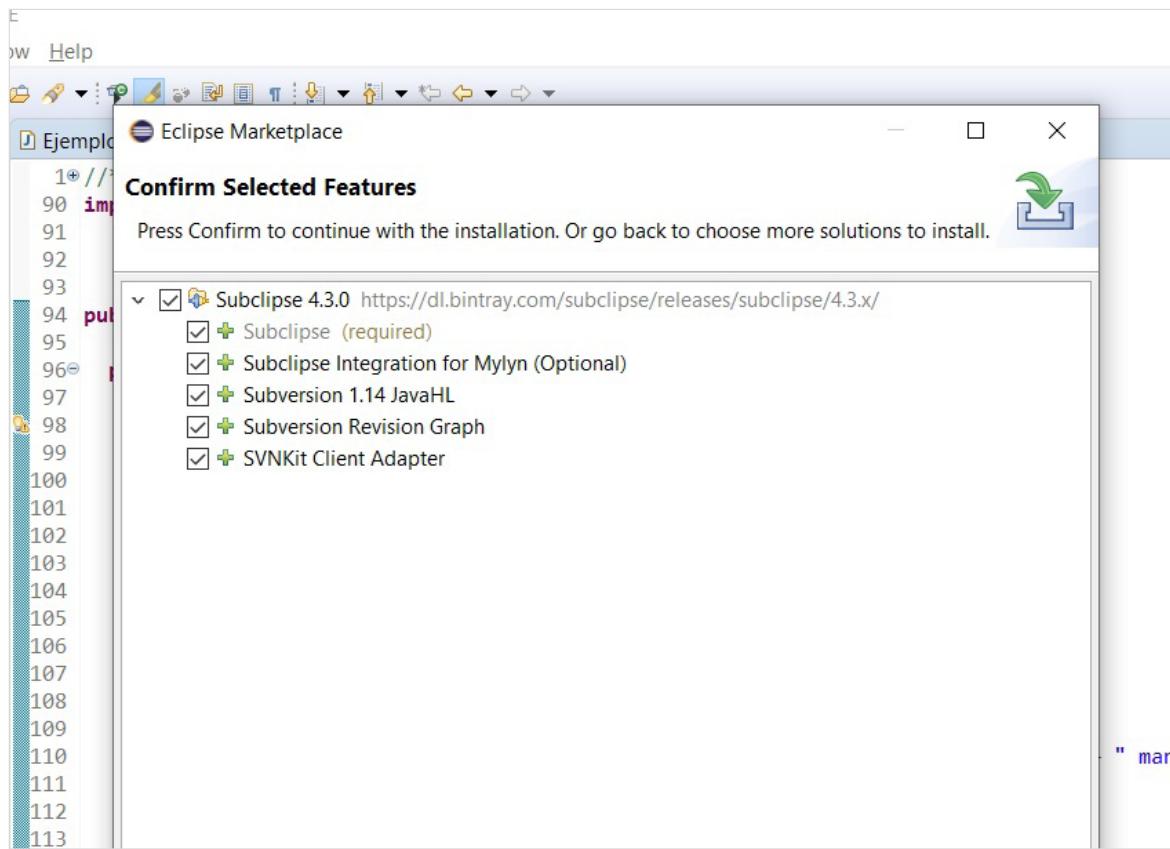
#### 4.2.2. TIPOS DE HERRAMIENTAS DE CONTROL DE VERSIONES

- **GIT** (<https://git-scm.com/>) es una herramienta de código libre que está diseñada tanto para pequeños proyectos como para proyectos de gran envergadura.

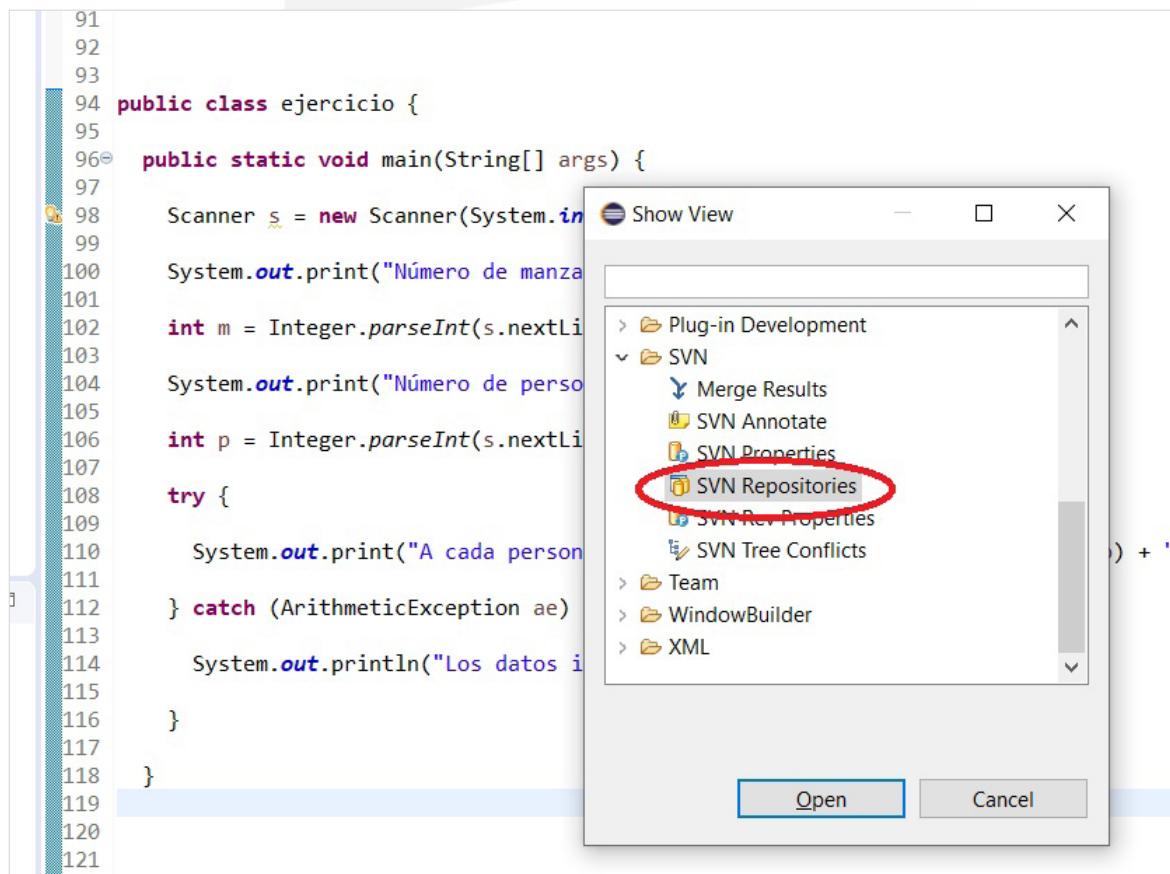
**Ramificación y fusión:** una de las características que distingue a este control de versiones es su modelo de ramificación. GIT te permite trabajar múltiples ramas que pueden ser independientes entre sí y:

- Hacer cambios de contexto sin conflictos. Podemos crear una rama para probar un determinado código. Podemos volver a donde bifurcamos esa rama y, si funciona, parchear.
- Establecer líneas de código que se basan en roles. Podemos tener una rama donde solamente ese código irá a producción y otra rama donde se realizarán pruebas.
- Realizar una división del trabajo basada en funciones. Cada programador estará trabajando en una funcionalidad del aplicativo. Una vez finalizada, se eliminará esta rama y se fusionará al proyecto principal.

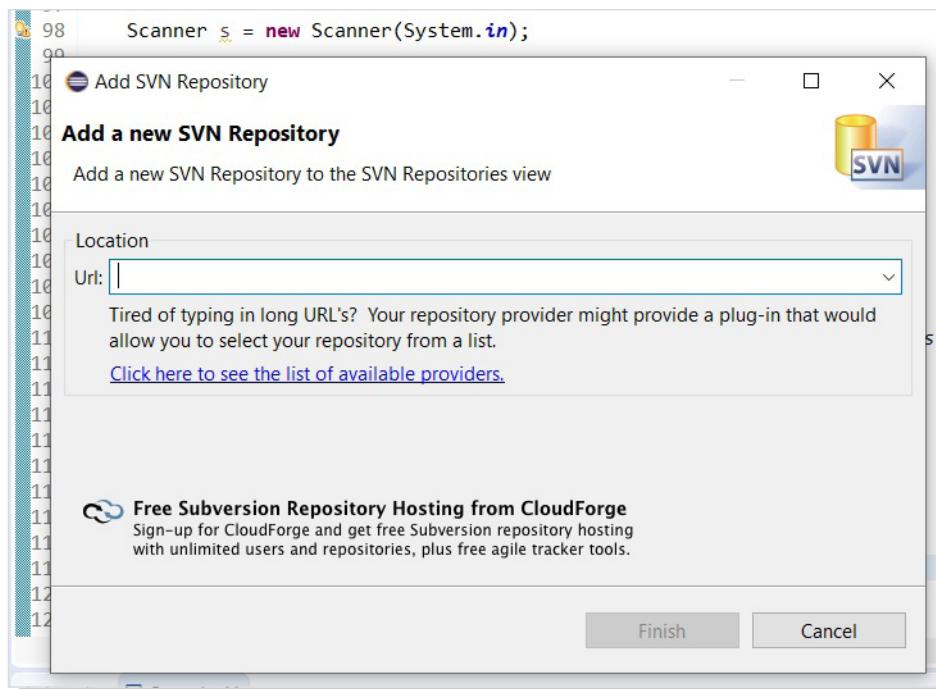
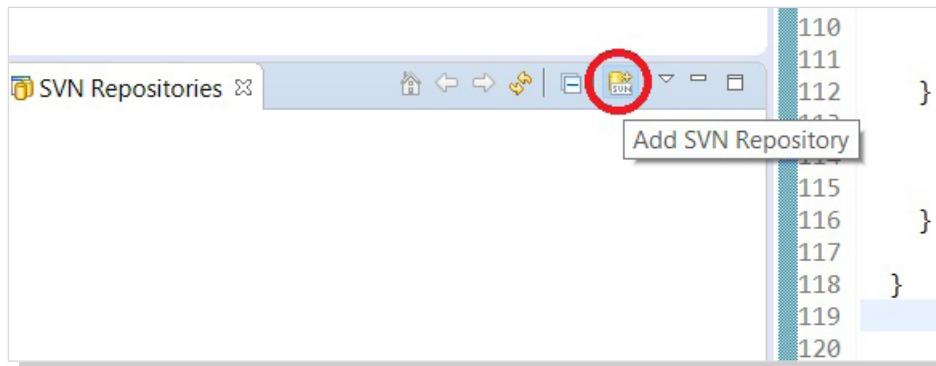
- **Subversión en Eclipse:** al utilizar Eclipse como cliente de VisualSVN, tendremos que realizar la instalación del plugin Subversive SVN. Desde *Eclipse Marketplace* (menú *Help*), buscamos *Subclipse* e instalaremos la versión correspondiente. Pulsamos en *Instalar*, seleccionamos y confirmamos los elementos para instalar y aceptamos los términos de la licencia para comenzar la instalación.



Una vez que tenemos instalado el *plugin*, desde la ventana del menú vamos a la opción *Show View -> Others -> SVN* y mostramos la sección *SVN Repositories*.

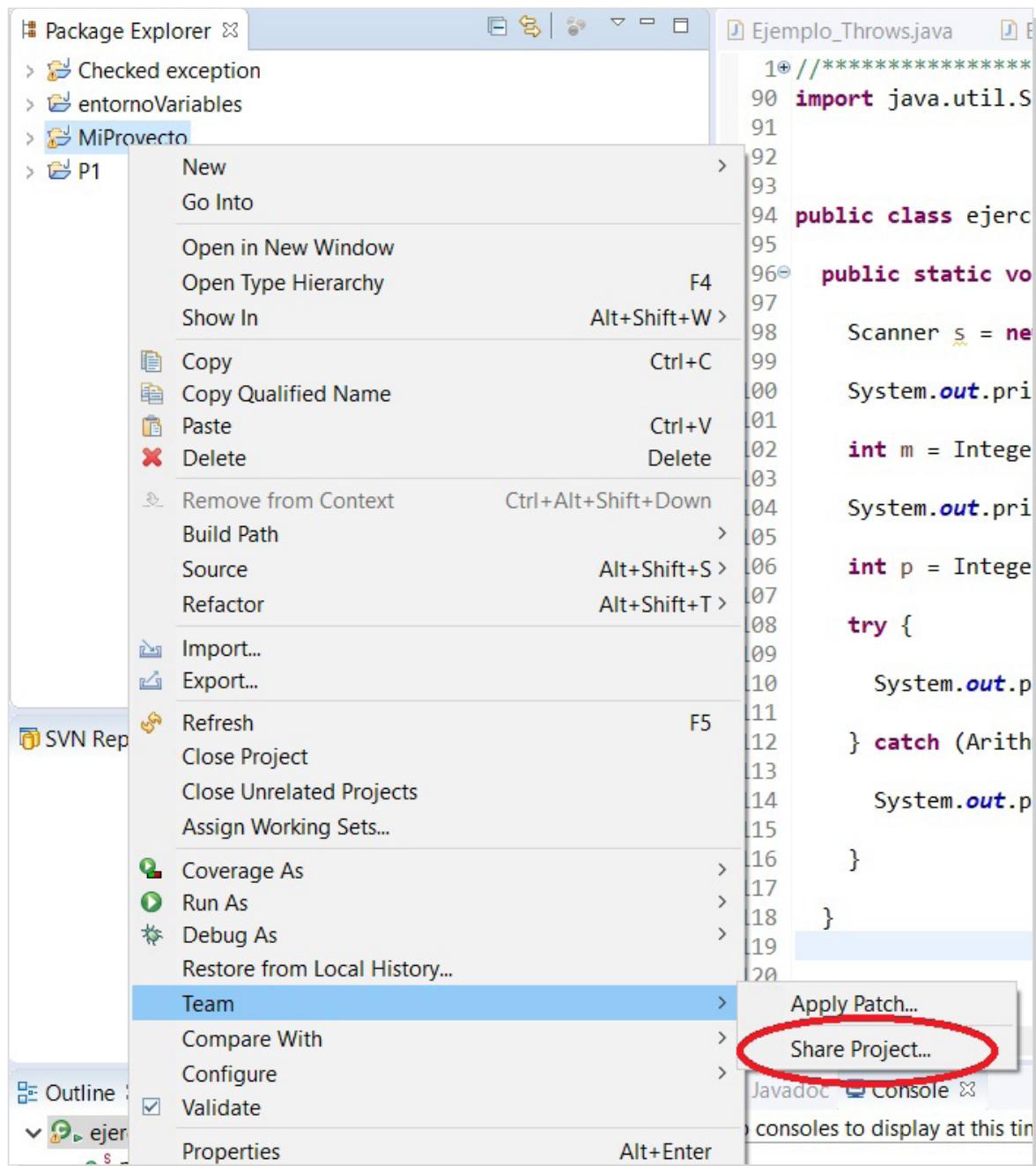


Dentro de esta sección, añadiremos el repositorio de nuestro servidor añadiendo la URL correspondiente.

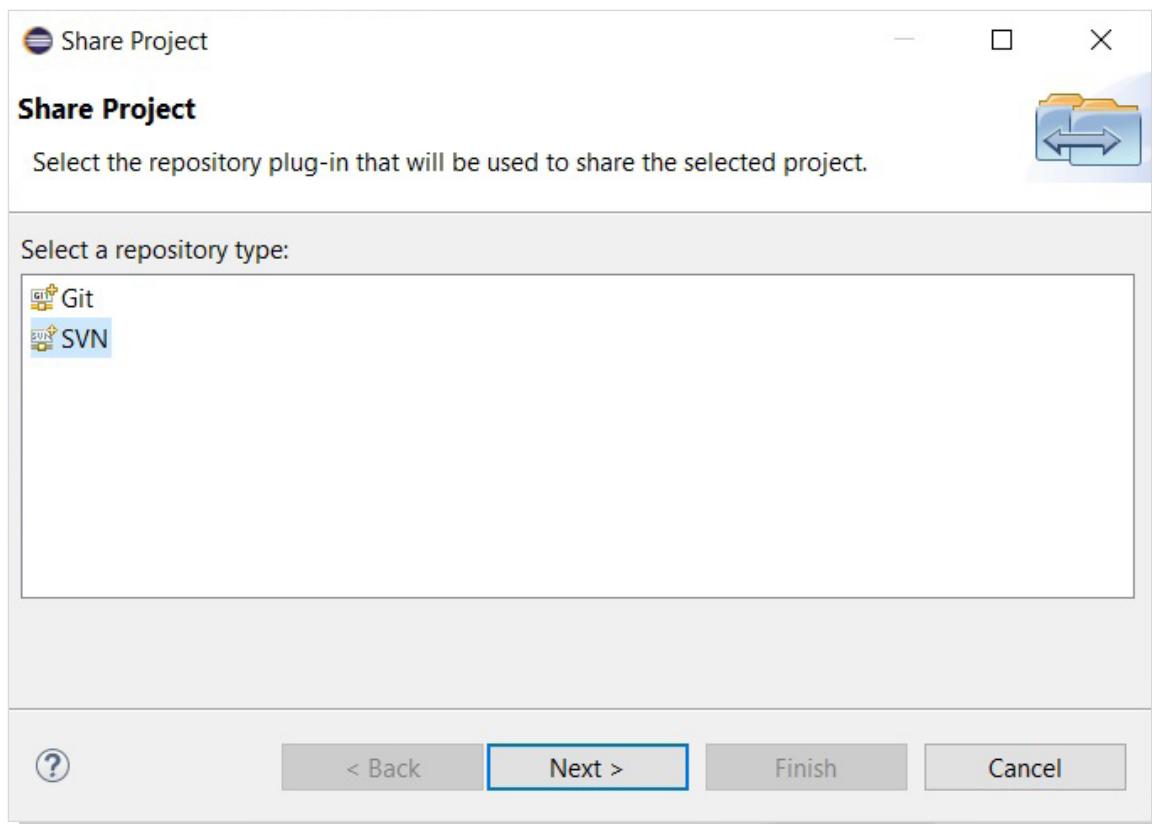


### Añadir un proyecto al repositorio

Cuando ya tengamos todo instalado, vamos a nuestro proyecto, clicamos con el botón derecho y vamos a la opción del menú **Team -> Share Project**.



Elegimos qué repositorio queremos (en este caso, SVN). Una vez que elegimos nuestro repositorio, solo queda seleccionar el servidor y especificar nuestra carpeta donde se encontrará la línea principal de nuestro proyecto (*trunk*).



## Operaciones con SVN Eclipse

A la hora de realizar las operaciones con Eclipse, veremos que podremos realizar las mismas operaciones que con cualquier otro cliente. Al posicionarnos sobre el proyecto o sobre un nodo, veremos las opciones de SVN.

Si lo que queremos es obtener una **copia** del trabajo, pulsaremos sobre **Check Out**, que almacenará la copia en la carpeta actual. En cambio, si elegimos **Check Out As**, la almacenará en la carpeta que elijamos. Para ver el historial de revisiones pulsaremos sobre **Show History**.

En la perspectiva Java se realizan los cambios en el proyecto. Si se realiza algún cambio, aparecerá un símbolo ">" en los elementos asociados.

Si visualizamos ahora el menú contextual sobre el elemento veremos las opciones de SVN.

Para poder ver los cambios, accederemos a la pestaña **Synchronize** de la barra de botones o también desde la perspectiva **Team Synchronizing**.

Desde la barra de botones podremos hacer *update*, validar o ver los conflictos. Si hay **cambios salientes**, aparece una flecha negra hacia fuera; si aparece el signo "+" es que el archivo es nuevo. Lo que habría que hacer es validar este archivo.

## Solución de conflictos

Un conflicto se producirá cuando dos usuarios modifiquen el mismo archivo del repositorio y las mismas líneas del archivo. Uno confirmará los cambios y el otro lo confirma a continuación, momento en que el servidor detectará la existencia de un conflicto, ya que se requiere validar los cambios en una copia que no ha sido actualizada con la versión del repositorio, que fue validada por el primer usuario.

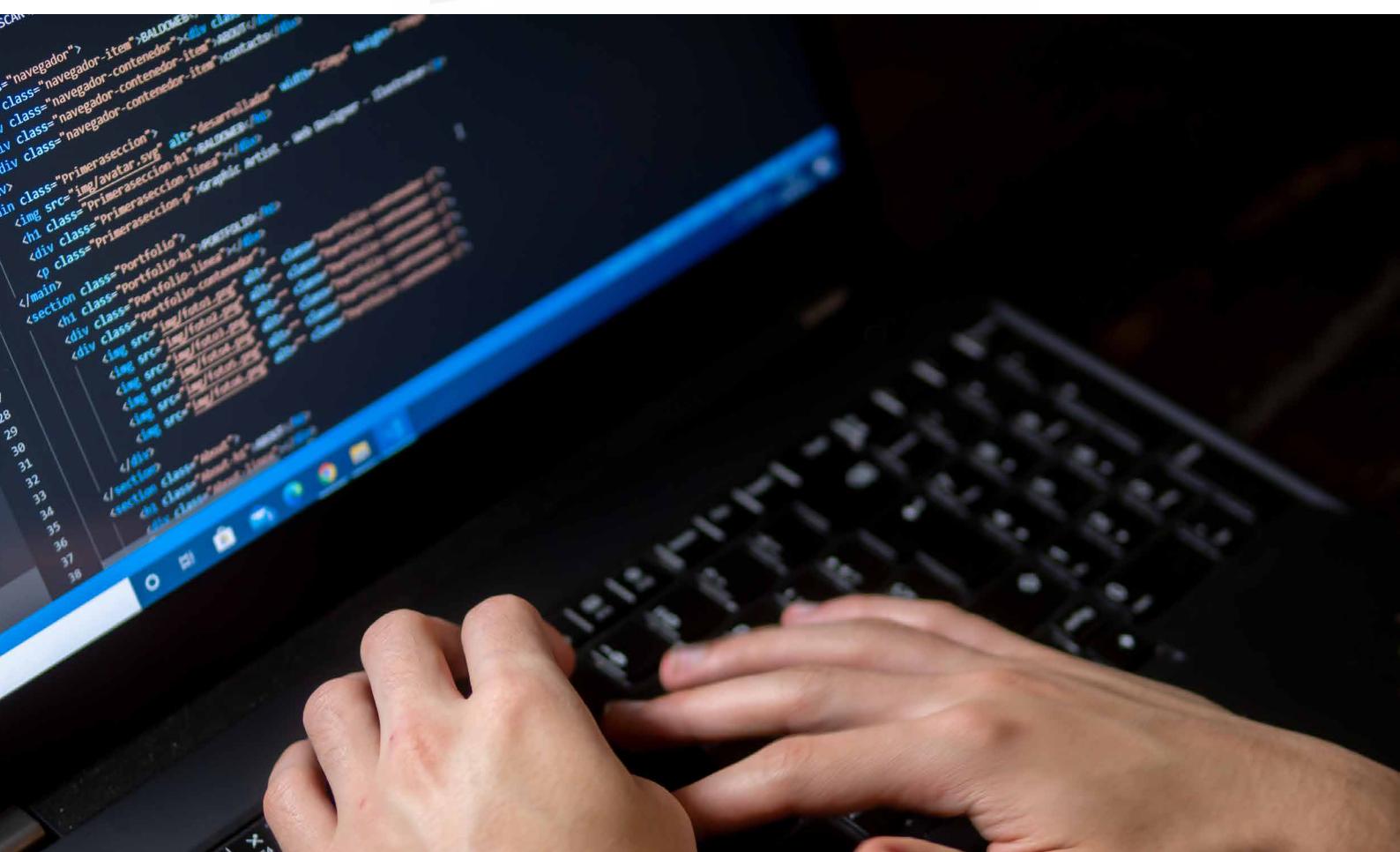
Para resolver un conflicto, una vez recogidos los cambios, será necesario hacer **Mark as Merged**. Si no se hace, Subversive sigue creyendo que hay conflicto, por lo que no dejará validar.



ponte a prueba

En SVN, el tronco es la línea principal del desarrollo del proyecto.

- a) Verdadero
- b) Falso



## 4.3. DOCUMENTACIÓN. USO DE COMENTARIOS. ALTERNATIVAS

La documentación es el texto escrito que acompaña a los proyectos. Es un requisito importante en un proyecto comercial, ya que el cliente querrá que se documenten las distintas fases del proyecto.

Podemos distinguir entre los siguientes tipos de documentación:

- **Documentación de las especificaciones:** sirve para comprobar que tanto las ideas del desarrollador como las del cliente son las mismas, ya que, de lo contrario, el proyecto no será aceptable. Según la norma IEEE 830, que recoge varias recomendaciones para la documentación de software, esta documentación deberá contener:
  - **Introducción:** se explican los fines y objetivos del software.
  - **Descripción de la información:** descripción detallada, incluyendo hardware y software.
  - **Descripción funcional:** detalla cada función del sistema.
  - **Descripción del comportamiento:** explica cómo se comporta el software ante acciones externas e internas.
  - **Criterios de validación:** documento sobre el límite de rendimiento, los tipos de pruebas, la respuesta esperada del software y las consideraciones especiales.
- **Documentación del diseño:** en este documento se describe toda la estructura interna del programa, formas de implementación, contenido de clases, métodos, objetos, etcétera.
- **Documentación del código fuente:** durante el desarrollo del proyecto se debe ir comentando en el código fuente cada parte, para tener una mayor claridad de lo que se quiere conseguir en cada sección.
- **Documentación de usuario final:** documentación que se entrega al cliente en la que se describe cómo usar las aplicaciones del proyecto.

### Documentación del código fuente

La documentación del código del programa también es fundamental para que todo el equipo pueda realizar funciones de actualización y reparación de errores de manera mucho más sencilla.

Esta debe describir lo que se está haciendo y por qué. Hay dos reglas que no se deben olvidar:

- Todos los programas poseen errores y es cuestión de tiempo que se detecten.
- Todos los programas sufren modificaciones a lo largo de su vida.

Al realizar las modificaciones, es necesario que el código esté bien documentado para que otro programador ajeno localice los cambios que quiere realizar.

Al documentarlo, habrá que explicar lo que realiza una clase o un método y por qué y para qué lo hace.

Para documentar proyectos existen muchas herramientas, como pueden ser PHPDoc, PHPDocumentor, Javadoc o JSDoc.

#### 4.3.1. USO DE JAVADOC EN ECLIPSE

Javadoc es una herramienta de Java que sirve para extraer y generar documentación básica para el programador a partir del código fuente en formato HTML. Tendremos que escribir los comentarios siguiendo las recomendaciones de Javadoc, y el código y la documentación se encontrarán dentro del mismo fichero.

Los tipos de comentarios para que genere la documentación son:

- **Comentarios de línea:** comienzan con los caracteres “//” y terminan en la misma línea.
- **Comentarios de bloque:** comienzan con “/\*” y terminan con “\*/”. Pueden contener varias líneas.
- **Comentarios de documentación Javadoc:** se colocan entre delimitadores “/\*\*...\*/”, podrán agrupar varias líneas y cada línea irá precedida por un “\*”. Deberá colocarse antes de la declaración de una clase, un campo, un método o un constructor. Podrá contener etiquetas HTML y los comentarios están formados por una descripción seguida de un bloque de tags.



## Uso de etiquetas de documentación

Las etiquetas de Javadoc van precedidas por "@" y las más utilizadas son:

ETIQUETA	DESCRIPCIÓN
@author	Autor de la clase. Solo para clases.
@version	Versión de la clase. Solo para clases.
@see	Referencia a otra clase.
@param	Descripción del parámetro. Una etiqueta por cada parámetro.
@return	Descripción de lo que devuelve. Solo si no es void.
@throws	Descripción de la excepción que puede propagar. Habrá una etiqueta throws por cada tipo de excepción.
@since	Número de versión del producto.

## Generar la documentación

Casi todos los entornos de desarrollo incluyen un botón para poder configurar Javadoc. Para hacerlo desde Eclipse, abrimos el menú *Project* y elegimos el botón *Generate Javadoc*. En la siguiente ventana nos pedirá la siguiente información:

- En *Javadoc Command* se indicará dónde se encuentra el fichero ejecutable de Javadoc, el javadoc.exe. Pulsamos en *Configure* para buscarlo dentro de la carpeta *workspace*.
- En los cuadros inferiores elegiremos el proyecto y las clases que documentar.
- Elegimos la privacidad de los elementos: con *Private* se documentarán todos los miembros públicos, privados y protegidos.
- Para finalizar, se indica la carpeta de destino en la que se almacenará el código HTML.
- Pulsar en *Next*, en la siguiente ventana poner el título del documento HTML que se genera y elegir las opciones para la generación de las páginas HTML. Como mínimo, se seleccionarán la barra de navegación y el índice.



# 5

## INTRODUCCIÓN AL UML

El lenguaje de modelado unificado (UML) es un lenguaje gráfico para visualizar, especificar y documentar cada una de las partes que comprende el desarrollo de software. Este lenguaje se puede utilizar para modelar sistemas de software, hardware u organizaciones en un entorno real. Para ello utiliza una serie de diagramas en los que se representan distintos puntos de vista de modelado. Podemos decir que UML es un lenguaje que se utiliza para documentar.

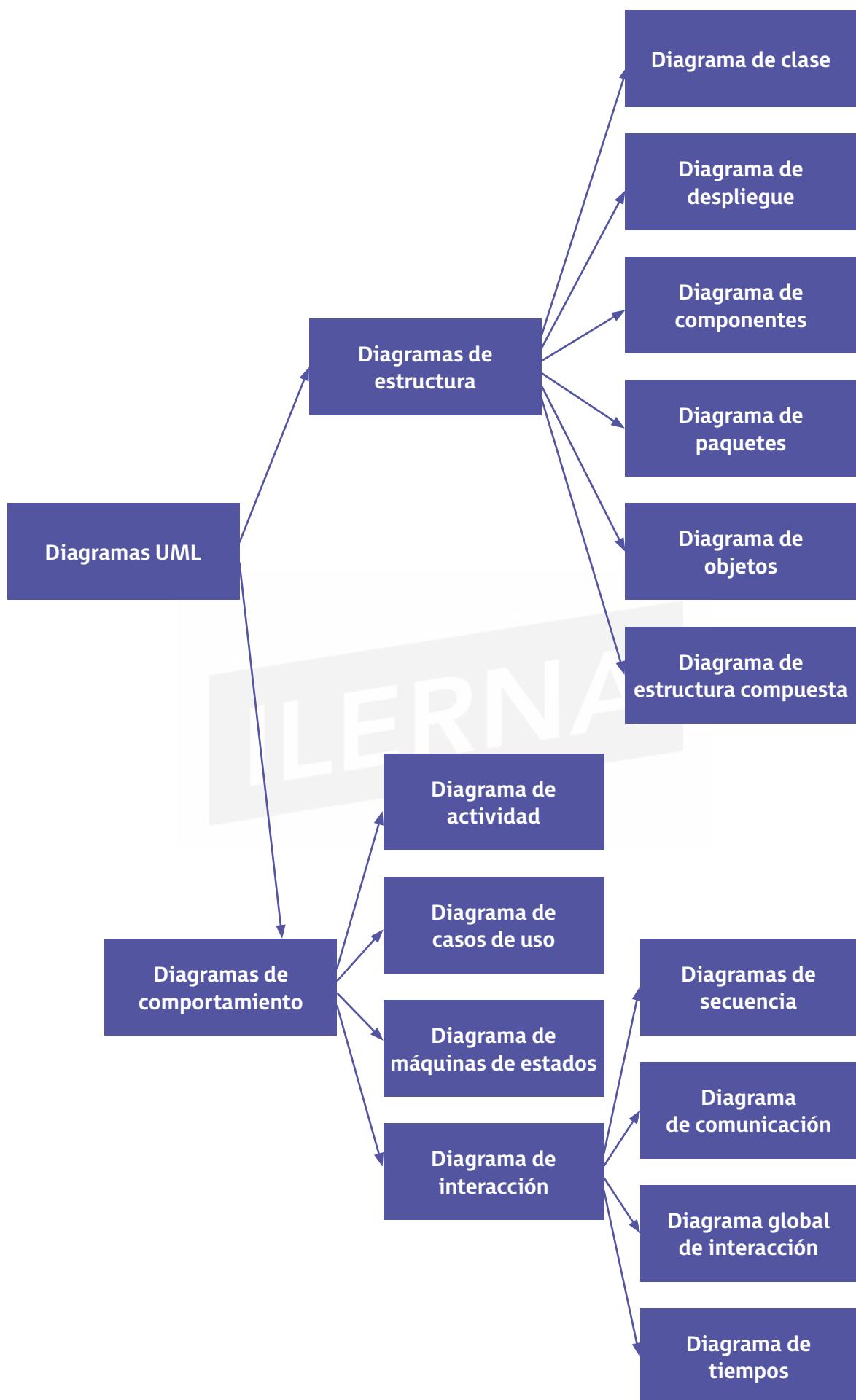
Existen dos grandes versiones de UML:

- UML 1.x: desde finales de los noventa se empezó a trabajar con el estándar UML. En los años sucesivos fueron apareciendo nuevas versiones que introducían mejoras o ampliaban a las anteriores.
- UML 2.x: en torno a 2005 se difundió una nueva versión de UML.

UML 2.0 define 13 tipos de diagramas, divididos en 2 categorías: 6 tipos de diagramas representan la estructura estática de la aplicación o del sistema y 4 tipos de diagramas representan el comportamiento del sistema:

- **Diagramas de estructura (parte estática del modelo):** incluyen el diagrama de clases, diagrama de objetos, diagrama de componentes, diagrama de estructura compuesta, diagrama de paquetes y diagrama de implementación o despliegue.
- **Diagramas de comportamiento (parte dinámica del modelo):** incluyen el diagrama de casos de uso, diagrama de actividad, diagramas de interacción y diagramas de máquinas de estado.
- **Diagramas de interacción:** son diagramas de comportamiento que incluyen diagramas como el diagrama de secuencia, diagrama de comunicación, diagrama de tiempos y diagrama de vista de interacción. Se centran en el flujo de control y de datos entre los elementos del sistema modelado.







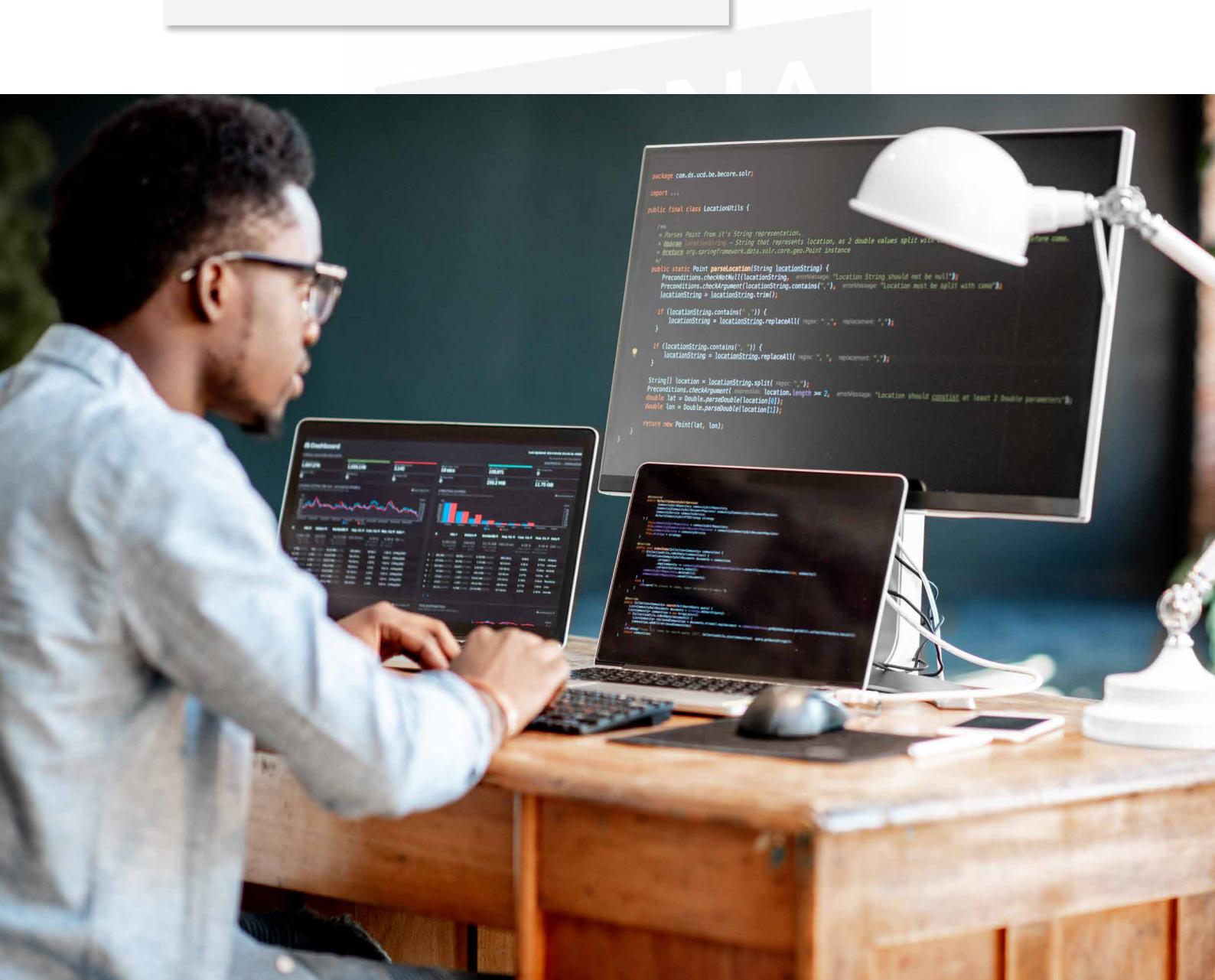
ponte a prueba

### ¿Qué afirmación sobre la UML es correcta?

- a) Nos permite construir y visualizar un sistema de software.
- b) Es un lenguaje de modelización.
- c) Nos permite documentar un sistema de software.
- d) Todas las respuestas son correctas.

### Los diagramas de interacción forman parte de los diagramas de comportamiento

- a) Verdadero.
- b) Falso.





# 6

## ELABORACIÓN DE DIAGRAMAS DE CLASES

En esta primera parte estudiaremos los conceptos básicos de la metodología orientada a objetos: los diagramas de clases. Entendemos por diagrama de clases una representación gráfica y estática de la estructura general de un sistema, mostrando cada una de las clases y sus relaciones representadas en forma de bloques. A lo largo de esta unidad usaremos varias herramientas para crear diagramas de clases y para generar código a partir de diagramas.

## 6.1. OBJETOS

Un objeto es la instancia de una clase. Esta constituido por un conjunto de atributos (valores), métodos (acciones y reacciones a mensajes) e identidad propia (que lo distingue de los demás objetos). La estructura y comportamiento de objetos similares tendrán una clase común. Una clase es un conjunto de objetos con estructura y comportamiento comunes.

Las propiedades de los objetos son claves. Los principios del modelo OO (orientado a objetos) son:

- **Abstracción:** características esenciales de un objeto de tal forma que lo distingue de los demás objetos. Indican qué trabajo va a realizar ese objeto y cómo se comunica con los demás sin necesidad de mostrar cómo se implementan dichas características.
- **Encapsulación:** conjunto de elementos que pertenecen a la misma entidad y al mismo nivel de abstracción. Gracias a esto, permite una mayor cohesión de sus componentes y separa la parte interna inaccesible para otros objetos de la externa, que sí será accesible. Dicho de otra forma, oculta los métodos y atributos a otros objetos, pasando a ser privados.
- **Modularidad:** es la capacidad de un sistema o aplicación para dividirse en pequeños módulos independientes.
- **Jerarquía o herencia:** propiedad que permite que algunas clases tengan propiedades y características de una clase superior. La clase principal se llama clase padre o **superclase**. La nueva clase se denominará clase hija o **subclase**, que heredará todos los métodos y atributos de la superclase. Cada subclase estará formada con objetos más especializados.
- **Polimorfismo:** cuando dos instancias u objetos pertenecientes a distintas clases pueden responder a la llamada a métodos del mismo nombre, cada uno de ellos con distinto comportamiento encapsulado. Ambos responden a una interfaz común (marcada a través del mecanismo de la herencia).

- **Tipificación:** definición precisa de un objeto de forma que objetos de diferentes tipos no se pueden intercambiar.
- **Concurrencia:** propiedad de un objeto que está activo respecto de otro que no lo está.
- **Persistencia:** propiedad de un objeto en virtud de la cual su existencia trasciende en el tiempo y/o el espacio.

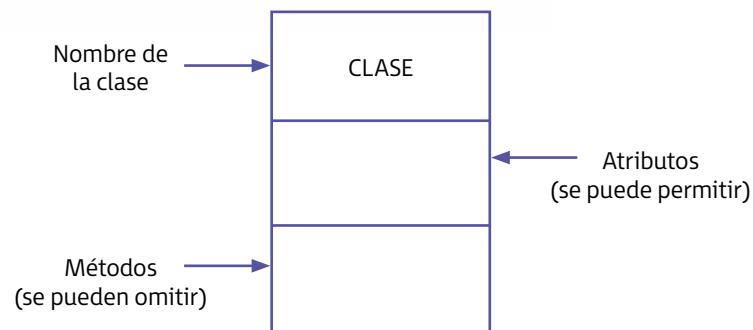
## 6.2. DIAGRAMAS DE CLASES

Como ya hemos comentado, un diagrama de clases es una representación gráfica y estática de la estructura general de un sistema, mostrando cada una de las clases y sus interacciones representadas en forma de bloques. Vamos a poder visualizar las relaciones que existen entre las clases. El diagrama estará compuesto por los siguientes elementos:

- **Clases:** atributos, métodos y visibilidad.
- **Relaciones:** asociación, herencia, agregación, composición y dependencia.

### 6.2.1. CLASES

Son la unidad básica que contendrá toda la información referente a un objeto. A través de ella se podrá modelar el entorno en estudio. Una clase en UML podemos representarla con un rectángulo dividido en tres partes:



- **Parte superior:** nombre de la clase.
- **Parte central:** atributos, que caracterizan la clase (*private, protected, package o public*).
- **Parte inferior:** métodos u operaciones, forma de interactuar del objeto con el entorno (según visibilidad: *private, protected, package o public*).

Al representar la clase, podemos obviar señalar los atributos y métodos. La visibilidad de los métodos debe ser por defecto *public*, y la de los atributos, *private*.

## Atributos y métodos

Los atributos representan las propiedades de la clase, mientras que los métodos son las implementaciones de un determinado servicio de la clase que muestra un comportamiento común.

Cuando los creamos, indicaremos también su visibilidad con el entorno, estando estrechamente relacionada con el encapsulamiento.

Se pueden distinguir los siguientes tipos:

- **Public**: se puede acceder al miembro de la clase desde cualquier sitio. Se representa con el signo "+".
- **Private**: solo se puede acceder al miembro de la clase desde la propia clase. Se representa con signo "-".
- **Protected**: solo se puede acceder a los miembros de la clase desde la misma clase o desde subclases (clases derivadas). Se representa con "#".
- **Package**: los miembros de las clases serán visibles dentro del mismo paquete. Se representa con la tilde "~".

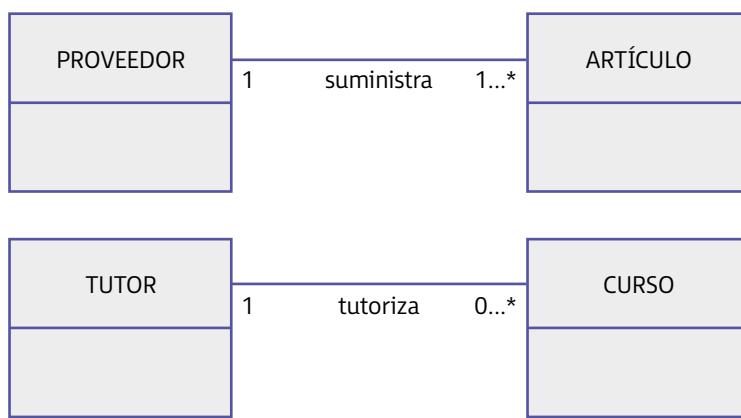
### 6.2.2. RELACIONES

Los objetos estarán vinculados entre sí, y se corresponden con asociaciones entre objetos. En UML, estos vínculos se describen a través de asociaciones al igual que los objetos se describen mediante clases.

Estas relaciones poseen un nombre y una cardinalidad llamada multiplicidad, que representa el número de instancias de una clase que se relaciona con las instancias de otra clase. La asociación es similar a la utilizada en el modelo entidad/relación.

En cada extremo será posible indicar la multiplicidad mínima y máxima para indicar el intervalo de valores al que tendrá que pertenecer siempre la multiplicidad. Se usará la siguiente notación:

Notación	Multiplicidad
0..1	Cero o una vez
1	Una y solo una vez
0..*	De cero a varias veces
1..*	De una a varias veces
M..N	Entre M y N veces



Podremos distinguir los siguientes tipos de relaciones:

### Asociación

Podrá ser **unidireccional** o **bidireccional**, dependiendo de si una conoce la existencia de la otra o no. Cada clase cuenta con un rol, que se indica en la parte superior o inferior de la línea que conecta a ambas clases, y, además, este vínculo también tendrá un nombre, representado con una línea que conecta a ambas clases.

Unidireccional

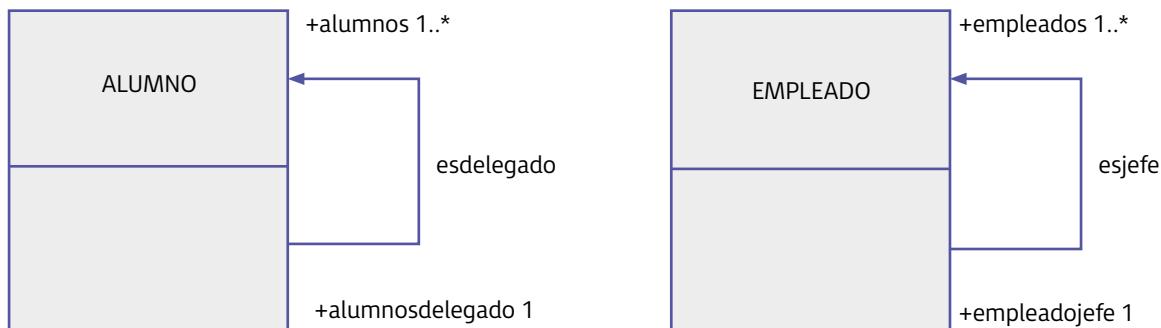
Bidireccional

Una asociación **bidireccional** se puede recorrer en ambos sentidos entre las dos clases. En cambio, en la asociación **unidireccional**, la clase destino no sabrá de la existencia de la clase origen, y la clase origen contendrá un objeto o varios de la clase destino.

Dependiendo de la multiplicidad, podemos pasar de un objeto de una clase a uno o varios de la otra. A este proceso se le llama **navegabilidad**. En la asociación unidireccional, la navegabilidad es solo en un sentido, desde el origen al destino, pero no al contrario.

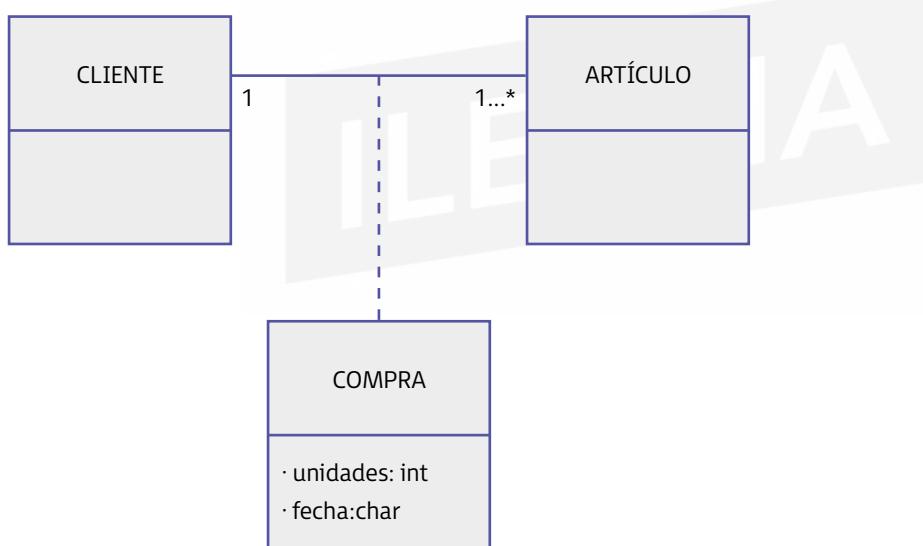


Algunas clases pueden asociarse consigo mismas, creando así una **asociación reflexiva**. Estas asociaciones unen entre sí instancias de una misma clase.



### Clase asociación

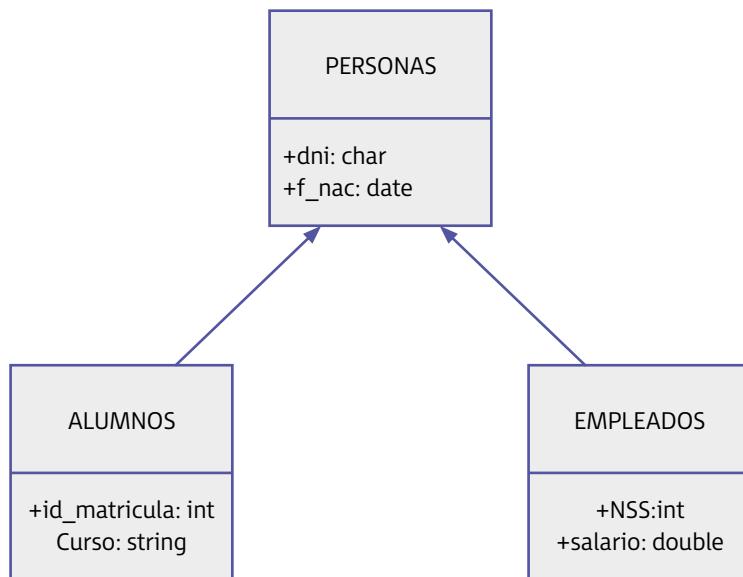
Hay asociaciones entre clases que podrán tener información necesaria para dicha relación, por lo que se creará una clase llamada clase asociación. Recibirá el estatus de clase y las instancias serán elementos de la asociación, así como podrán vincularse con otras clases.



### Herencia (generalización y especialización)

Podremos organizar las clases de forma jerárquica y, gracias a la herencia, seremos capaces de compartir atributos y operaciones comunes con las demás clases a través de una superclase. Las demás clases se conocen como subclases. La subclase hereda los atributos y métodos de la otra. La superclase generaliza a las subclases y las subclases especializan a la superclase.

Para representar esta asociación se usará una →, donde el extremo de la flecha apunta a la superclase.

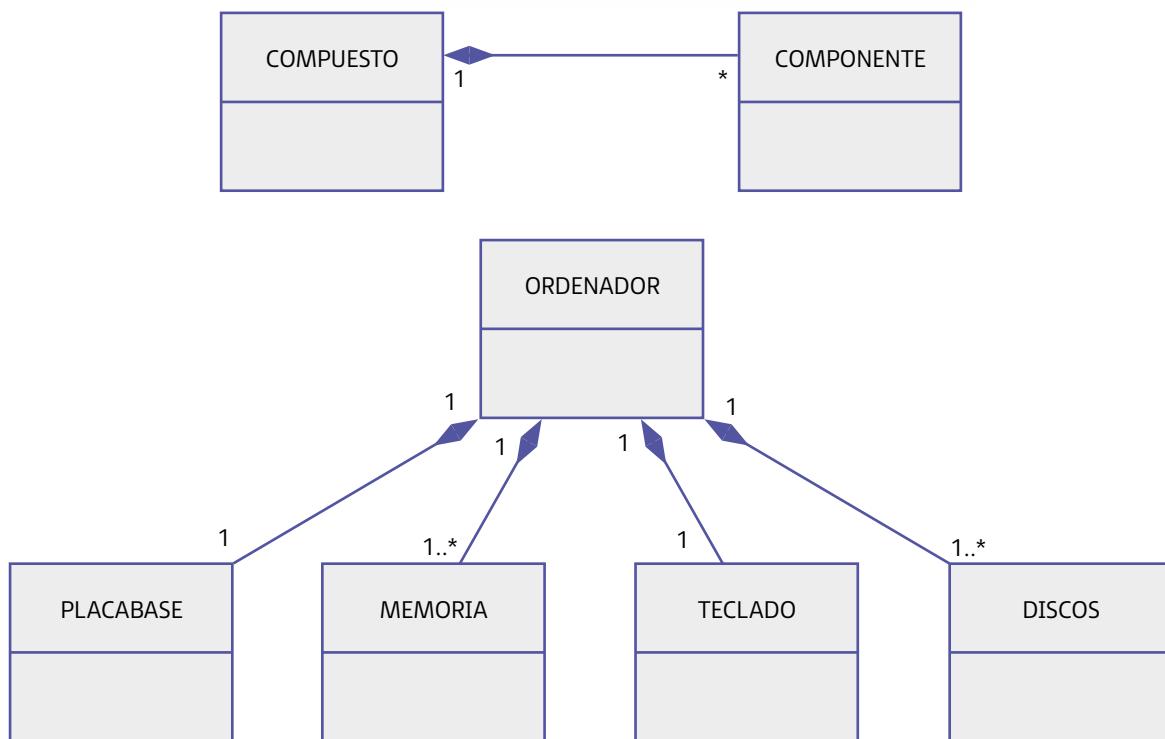


### Composición

La asociación de composición consiste en que un objeto puede estar compuesto por otros objetos. Existirán dos formas: asociación fuerte, conocida como **composición**, y asociación débil, conocida como **agregación**.

En la asociación fuerte, los objetos están constituidos por componentes y la supresión del objeto comporta la supresión de los componentes. La cardinalidad será de uno a varios.

Se representa con una línea con un rombo lleno —————◆—.



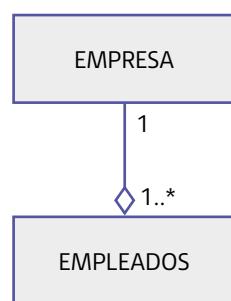


# ILERNA

## Agregación

Es la composición débil, en la cual los componentes pueden ser compartidos por varios compuestos y la destrucción de uno de ellos no implica la eliminación del resto. Se suele dar con más frecuencia que la composición.

Su representación es una línea con un rombo vacío —◇—.



## Dependencia

Es una relación unidireccional donde una clase utiliza a otra:



Características de la relación:

- La clase *impresora* usa a la clase *documento*.
- La clase *impresora* depende de *documento*.
- Dada la dependencia, todo cambio en *documento* podrá afectar a *impresora*.
- La *impresora* conoce la existencia de *documento*, pero al revés no.



ponte a prueba

**¿Qué cardinalidad corresponde a este tipo de relaciones?**

Libro	Ejemplar
ISBN: 1 Texto Título: 1 Texto Año de publicación: 1 Tiempo Idioma: 1..* enum Idioma	Código: 1 Texto Editorial: 1 Texto Año de adquisición: 1 Tiempo

- a) 1 a 1.
- b) 1 a varios.
- c) Varios a varios.
- d) 0 a 1.

**"Debemos de registrar el nombre, apellidos y número de teléfono de una persona en nuestro aplicativo. También debemos modelar las relaciones familiares de progenitor y cónyuge." ¿De qué forma podemos modelar este caso?**

- a) Como una relación de agregación entre una clase "persona" y otra "familiar".
- b) Como una doble relación reflexiva.
- c) Como una relación asociativa entre una clase "persona" y otra "familiar".
- d) Necesitamos más información para poder modelar este caso.

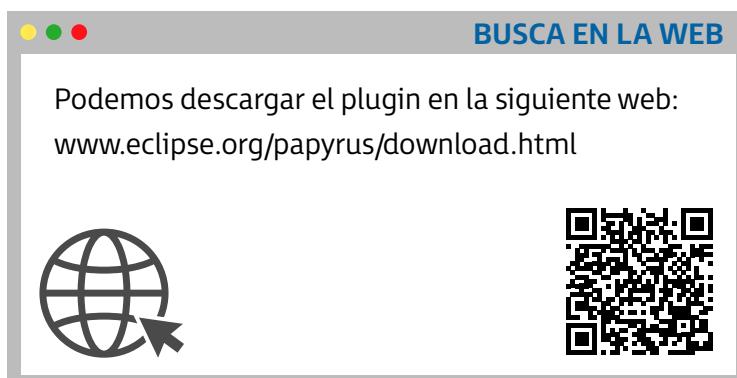
**La clase A depende de la clase B, por lo que A no conoce la existencia de B.**

- a) Verdadero.
- b) Falso.

## 6.3. HERRAMIENTAS PARA EL DISEÑO DE DIAGRAMAS

### 6.3.1. PAPYRUS PARA ECLIPSE

Es una herramienta de modelado UML de código abierto e incluye soporte para todos los diagramas UML. Con este *plugin*, trabajaremos desde Eclipse para realizar los modelados.



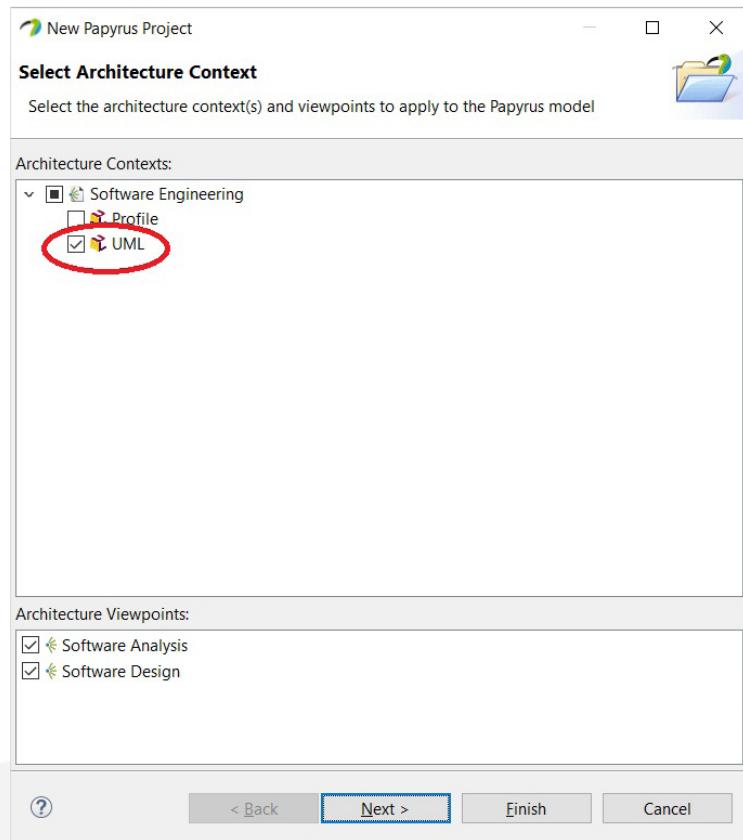
Algunas características de este *plugin* son las siguientes:

- De código abierto basado en el entorno Eclipse.
- Proporciona un entorno integrado para el usuario para editar cualquier tipo de modelo EFM (metamodelos).
- Proporciona editores de diagramas para lenguajes como UML2 y SysML.
- Tiene un soporte de perfil de usuario.
- En términos generales, tiene formas de personalización muy potentes que pueden ser definidas por el usuario.

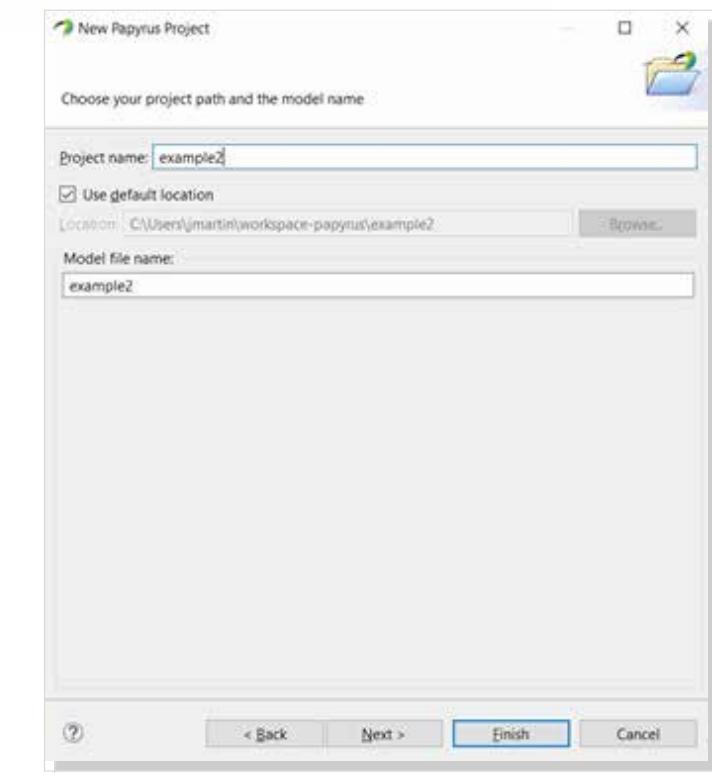
Al descargar el *plugin*, solo tenemos que descomprimirlo en una carpeta y ejecutar el .exe (hay que recordar que el IDE de Eclipse tiene que estar instalado en tu equipo).

Nombre	Fecha de modificación	Tipo	Tamaño
📁 configuration	02/06/2020 13:45	Carpeta de archivos	
📁 features	19/03/2020 17:35	Carpeta de archivos	
📁 p2	19/03/2020 17:34	Carpeta de archivos	
📁 plugins	19/03/2020 17:35	Carpeta de archivos	
📁 readme	19/03/2020 17:35	Carpeta de archivos	
📄 .eclipseproduct	19/12/2019 0:11	Archivo ECLIPSEPR...	1 KB
📄 artifacts	19/03/2020 17:35	Documento XML	476 KB
⚙️ papyrus	19/03/2020 17:37	Aplicación	416 KB
⚙️ papyrus	19/03/2020 17:35	Opciones de confi...	1 KB

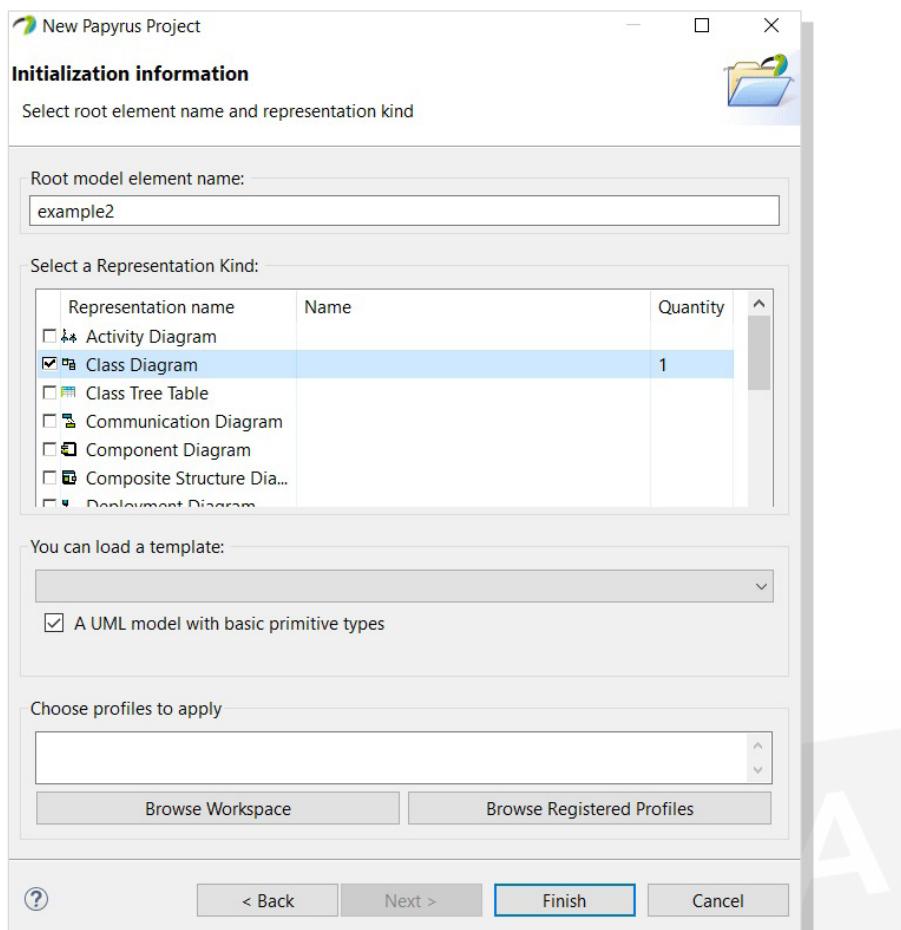
Cuando abrimos Papyrus, nos dará las opciones de qué arquitectura escoger. En este caso, vamos a realizar un diseño en UML.



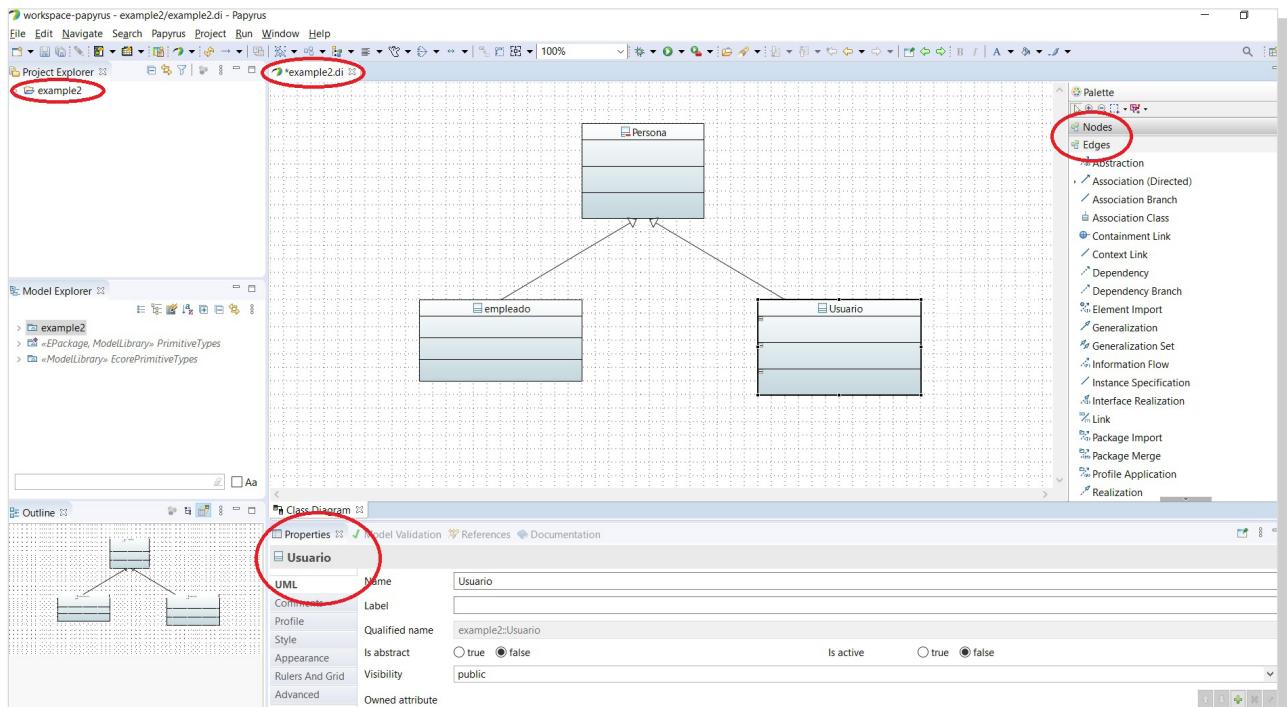
Crearemos nuestro proyecto y elegiremos la ruta donde queremos almacenarlo.



Finalmente, seleccionamos la representación que queremos. En este caso, vamos a realizar un diagrama de clases.



A continuación, veremos un ejemplo del entorno de diseño de UML.



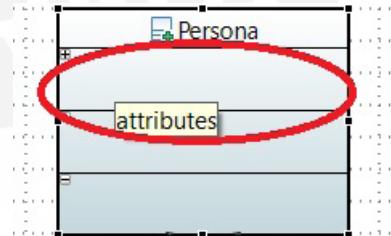
Se distinguirán cuatro paneles principales:

- **Panel principal:** es donde colocaremos nuestros elementos del diagrama que queramos modelar.
- **Panel del proyecto:** referencia a la estructura de nuestro proyecto principal.
- **Panel de propiedades:** en este panel daremos valores a las propiedades de nuestros elementos del diagrama.
- **Panel de nodes y edges:** en la parte de *nodes*, podemos elegir el tipo de elemento de nuestro diagrama. Por ejemplo, en un diagrama de clases, podemos elegir crear una clase interfaz, o un paquete. En la parte de *edges* elegiremos qué tipo de relación tendrán nuestros elementos. Por ejemplo, en un diagrama de casos de uso, tendremos relaciones de <<include>> o de <<extend>>.

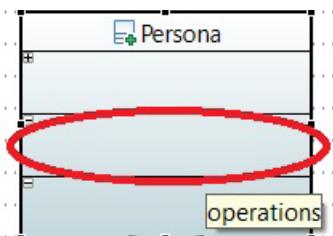
### Creación de diagramas de clase

A la hora de crear una clase en UML, podemos dividirla en cuatro partes principales:

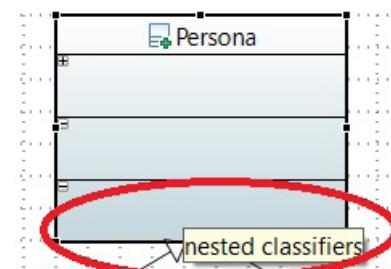
- Nombre de la clase.
- Atributos de la clase que ya hemos comentado que pueden ser públicos, privados, protegidos o *package*.



- Métodos de la clase exactamente igual que los atributos.

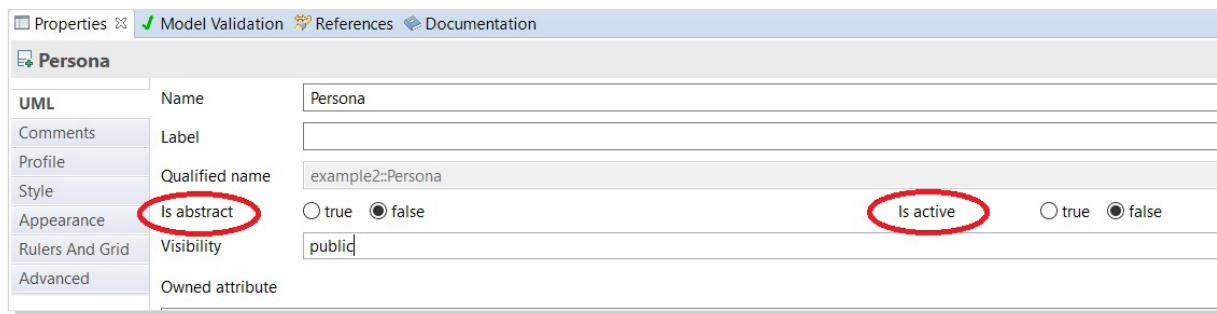


- *Nested classifiers:* limita la visibilidad del clasificador definido en la clase o interfaz al alcance del *namespace* para poder encapsular la información.



Dentro del panel de propiedades, podemos encontrar los tipos de modificadores:

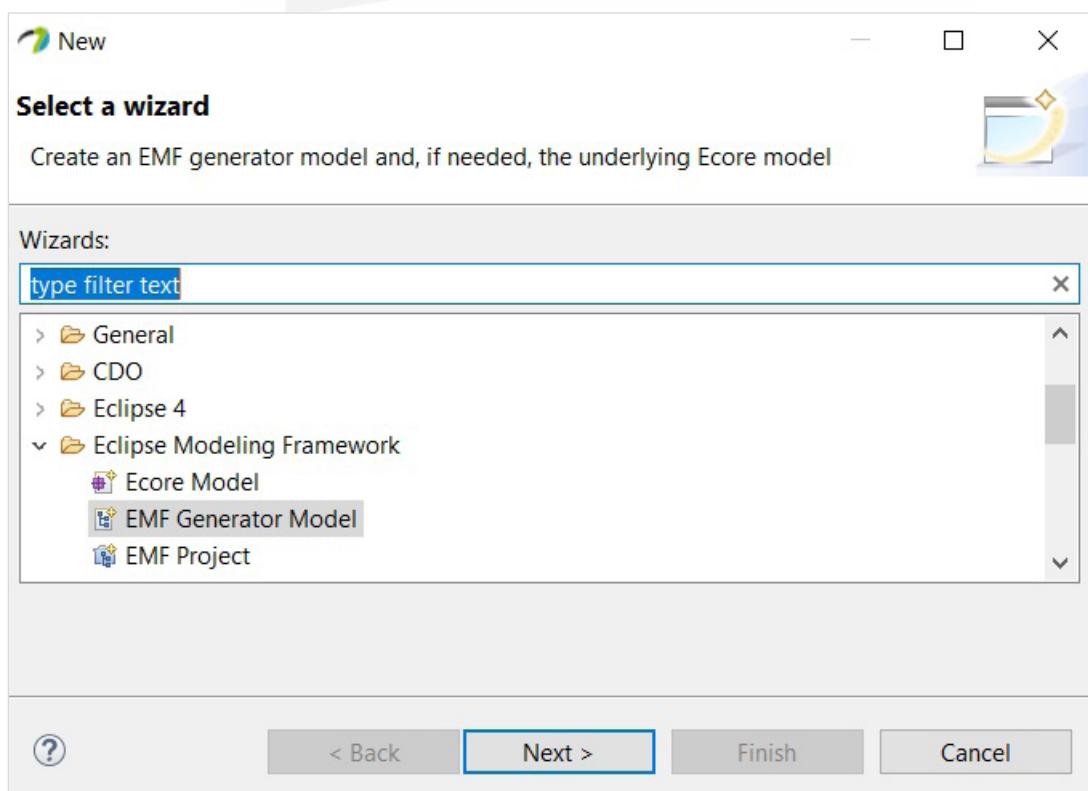
- **Is Abstract**: indica si la clase es abstracta.
- **Is Active**: indica si la clase es activa. Para que lo sea, debe poseer objetos activos, es decir, que realicen uno o más procesos. Si no realizan ninguna actividad, serán objetos pasivos.



### 6.3.2. GENERACIÓN DE CÓDIGO EN PAPYRUS

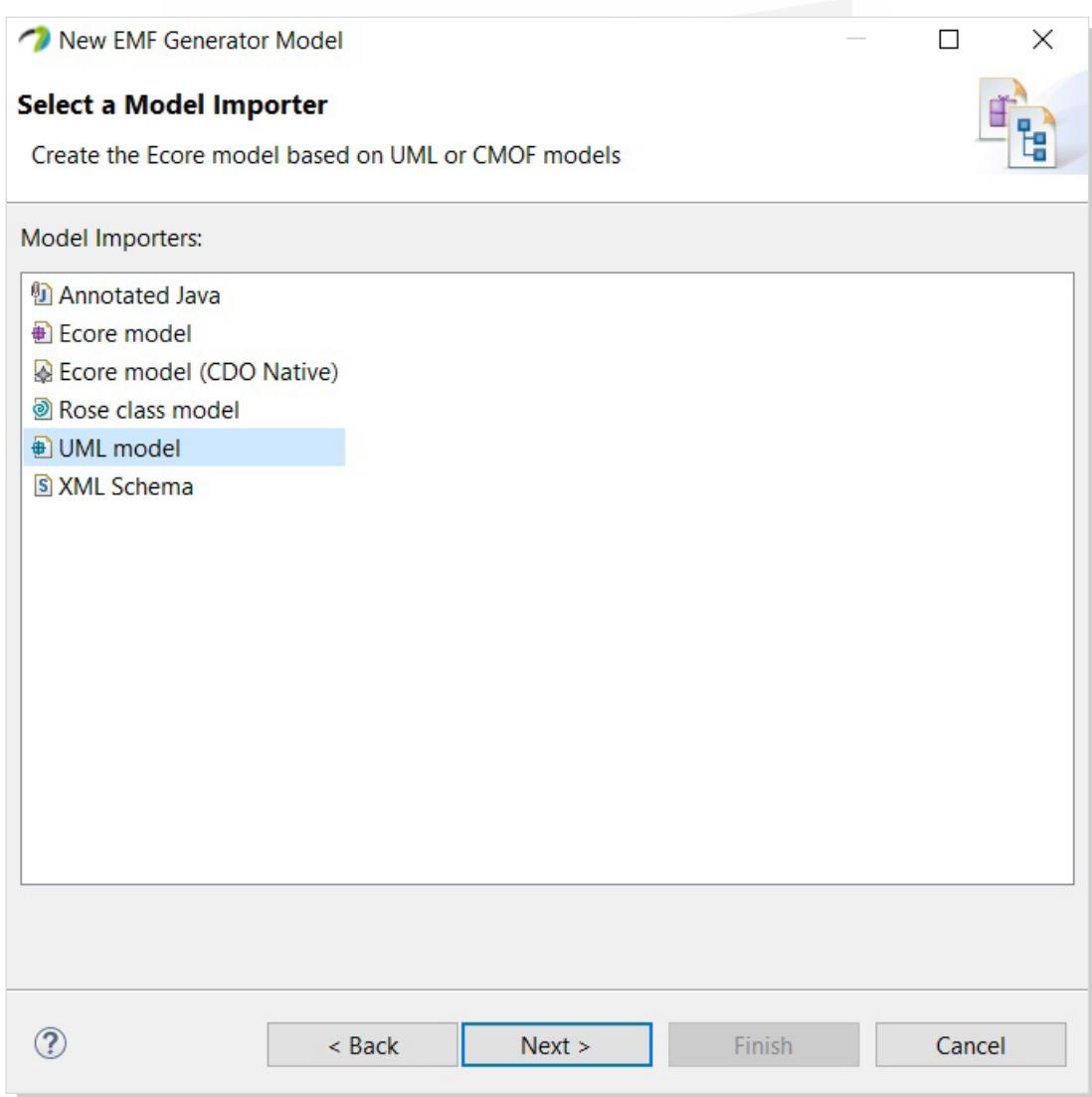
Cuando hemos creado nuestro diagrama de clases, podemos implementar ese diseño a código Java.

Dentro de nuestro proyecto, vamos a nuestro archivo de UML, clicamos en el botón derecho de nuestro ratón y creamos un nuevo elemento. En este caso, seleccionamos Other. Nos saldrá una ventana con varias opciones, elegiremos *EFM Generator Model*.

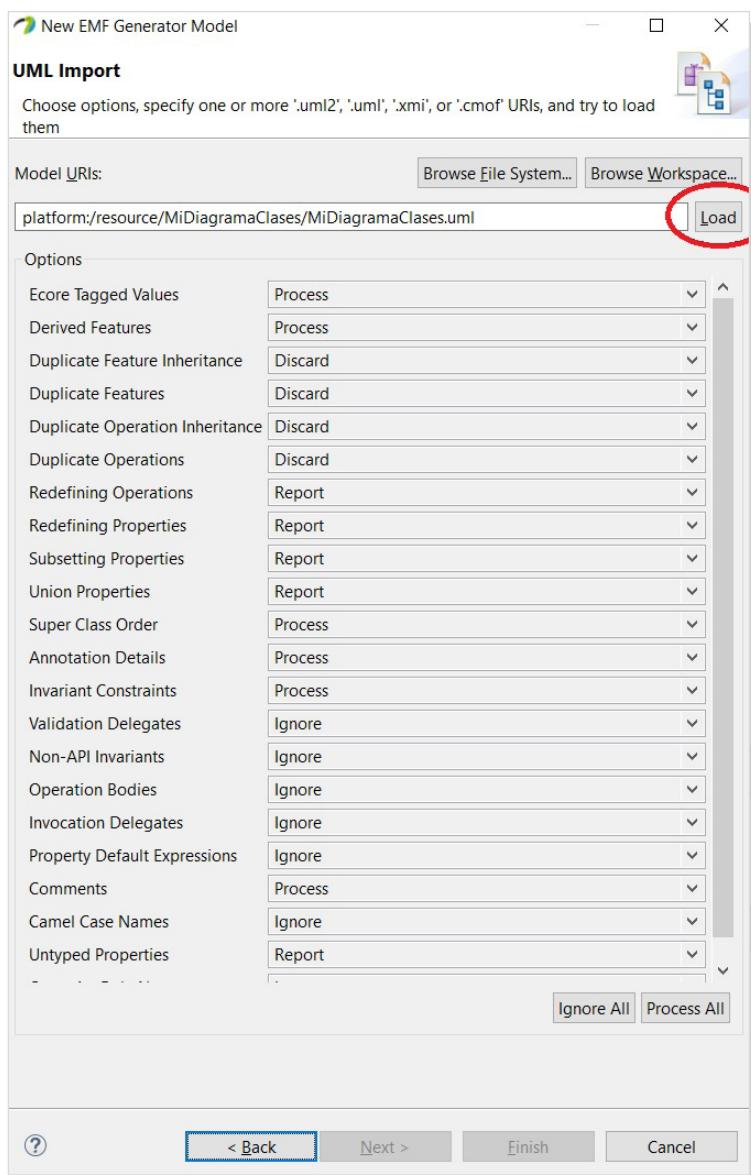




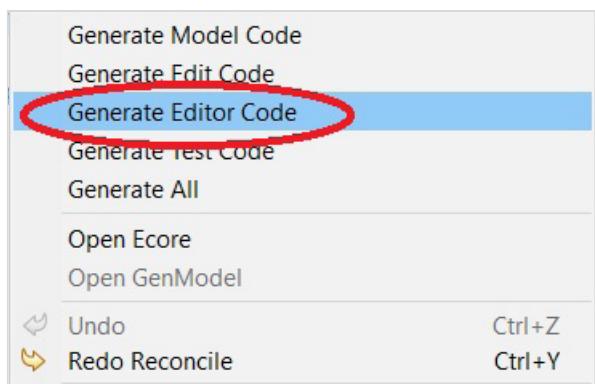
Una vez seleccionado nuestro generador, elegimos nuestro proyecto e importamos nuestro diagrama de UML.



Cargamos nuestro diagrama (*load*) y seleccionamos aquellas opciones que queramos habilitar.



Finalmente, se generará un archivo .genmodel. Cuando pulsemos sobre él con el botón derecho de nuestro ratón, nos aparecerá un menú donde clicaremos para generar nuestro código (*Generate Editor Code*) y la estructura de nuestro proyecto.



**ponte a prueba**

**¿Con cuál de los siguientes programas puedo modelar en UML?**

- a) ArgoUML
- b) StarUML
- c) Modelio
- d) Todas las respuestas son correctas

**Papyrus UML es un entorno de modelado de Eclipse.**

- a) Verdadero
- b) Falso



# 7

## ELABORACIÓN DE DIAGRAMAS DE COMPORTAMIENTO

En este último apartado vamos a centrarnos en cómo modelar lo que sucede en un sistema de software por medio de diagramas de comportamiento.

## 7.1. TIPO. CAMPO DE APLICACIÓN

Los diagramas de comportamiento nos permiten modelar la información que hemos manejado anteriormente con los diagramas de clases. Dentro de estos diagramas podemos encontrar los de casos de uso, actividad, estado e interacción. Los diagramas de interacción incluyen el diagrama de secuencia, de comunicación, de tiempos y de vista global de interacción.

Los diagramas de comportamiento mostrarán, como su propio nombre indica, el comportamiento de un sistema. Se clasifican en:

DIAGRAMA	RESUMEN
Diagrama de casos de uso	Describe el comportamiento del sistema desde el punto de vista de un usuario que interactúa con él.
Diagrama de actividad	Parecido a diagramas de flujo, muestra pasos, puntos de decisión y bifurcaciones.
Diagrama de estado	Muestra estados de un objeto y transiciones de un estado a otro.
Diagrama de secuencia	Muestra interacción de unos objetos con otros.
Diagrama de comunicación	Interacciones entre elementos en tiempo de ejecución.
Diagrama de tiempos	Define comportamiento de objetos en una escala de tiempo.
Diagrama de vista global de interacción	Cooperación entre otros diagramas de interacción.

## 7.2. DIAGRAMAS DE CASOS DE USO. ACTORES, ESCENARIO, RELACIÓN DE COMUNICACIÓN

Los casos de uso van a modelar el sistema desde el punto de vista del usuario. Con esta herramienta vamos a poder obtener los requisitos de software en la fase de análisis de un proyecto.

Tendrá que cumplir los siguientes objetivos:

- Definir los requisitos funcionales y operativos del sistema, diseñando un escenario que nos haga más fácil describir cómo se usará el sistema.
- Proporcionar una descripción clara de cómo el usuario interactúa con el sistema, y viceversa.
- Facilitar una base para la validación de las pruebas.

Se usará un lenguaje sencillo y deberá describir todas las formas de utilizar el sistema, por lo que define todo el comportamiento de este.

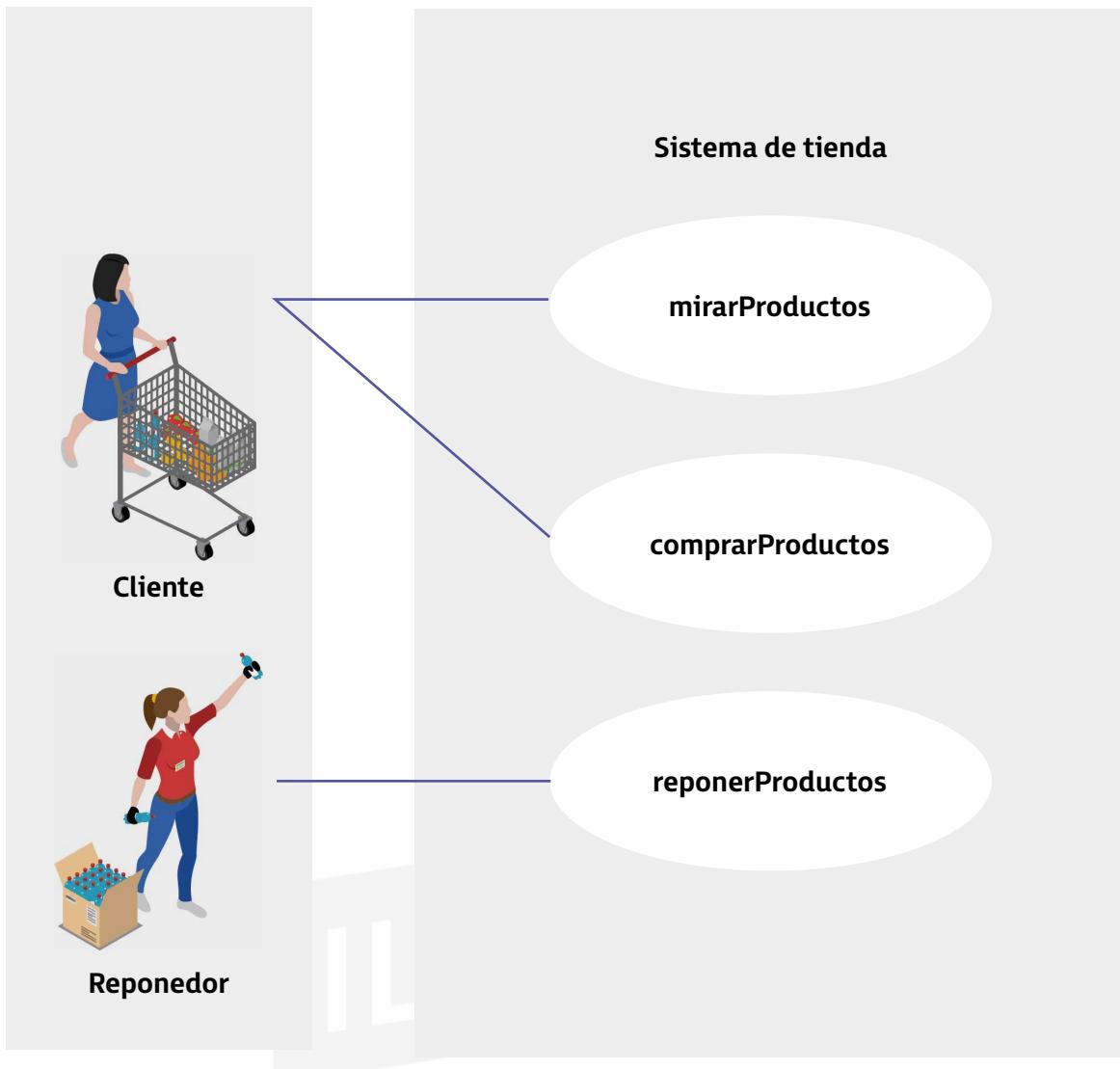
Usando UML podemos crear una representación visual de los casos de uso llamada **diagrama de casos de uso**.

### 7.2.1. ELEMENTOS DEL DIAGRAMA DE CASOS DE USO

Los **componentes** de un diagrama de casos de uso son:

- **Actores:** llamamos actor a cualquier agente que interactúa con el sistema y es externo a él. El actor se representa con un monigote y, en este tipo de diagramas, no se representa la interacción de estos.
- **Casos de uso:** representa una unidad funcional del sistema que realizará una orden de algún agente externo. Será iniciado por un actor y otros actores podrán participar de él. Se representan con un óvalo o eclipse y una descripción textual.
- **Relaciones:** existen varios tipos. La más común es una relación bidireccional entre actores y casos de uso representada con una línea continua.
- También podemos tener un rectángulo que delimita el **sistema**.

Los casos de uso serán iniciados por actores que pueden solicitar o modificar información del sistema. El nombre debe coincidir con el objetivo del actor principal, que será el que normalmente comience el caso de uso.



### 7.2.2. IDENTIFICAR A ACTORES

Los actores son unidades externas que van a interactuar con el sistema. Normalmente son personas, pero pueden ser otros sistemas o incluso dispositivos. Para poder interactuar con el sistema, hay que conocer la información de cada elemento para saber qué y quién interactúa con el sistema y qué rol tendrá cuando se interactúe con él. Es necesario tener en cuenta los siguientes puntos a la hora de definir los actores:

- Serán siempre externos al sistema.
- Van a interactuar directamente con el sistema.
- Representan roles de personas o elementos que desempeñan en relación con el sistema.
- Necesitan un nombre que describa la función que desempeñan.
- Una misma persona o elemento puede desempeñar varios roles como actores distintos.

### 7.2.3. IDENTIFICAR CASOS DE USO

Para identificarlos, será necesario entender el funcionamiento del sistema y lo que quiere hacer. Para ello tendremos que buscar los actores que participan y saber cómo lo usarán.

Para documentar los casos de uso, podremos hacerlo a través de una plantilla que nos describa lo que hace el actor y lo que ocurre cuando se interactúa con dicho sistema. Una plantilla sencilla podrá ser:

- **Nombre** del caso de uso.
- **ID** del caso de uso.
- **Pequeña descripción** de lo que se espera del caso de uso.
- **Actores implicados**. Existen principales y secundarios. Los primeros activan el sistema, y los segundos usan el caso de uso una vez iniciado.
- **Precondiciones**. Serán las condiciones que se deberán cumplir antes de que empiece el caso de uso.
- **Curso normal**. Pasos del caso de uso para que finalice correctamente.
- **Poscondiciones**. Las condiciones que se deberán cumplir cuando finalice el caso de uso.
- **Alternativas**. Errores o excepciones.

A continuación, veremos la plantilla a llenar de un caso de uso:

Identificador de Caso Uso	[CU_XX]
Nombre	[Nombre del caso de uso]
Descripción	[Descripción deñ caso de uso]
Actores	[Actores responsables de realizarlo o ejecutarlo]
<b>Secuencia normal</b>	
Actor	Software
1. Acción 1	
	2. Acción 2
	...
	N-1. Actividad N-1
	N. El caso de uso termina.
Excepciones	Software
<b>CU relacionados</b>	
CU relacionados	[CU_XX]
Precondición	[Estado del sistema antes de la ejecución del CU]
Post condición	[Estado del sistema después de la ejecución del CU]

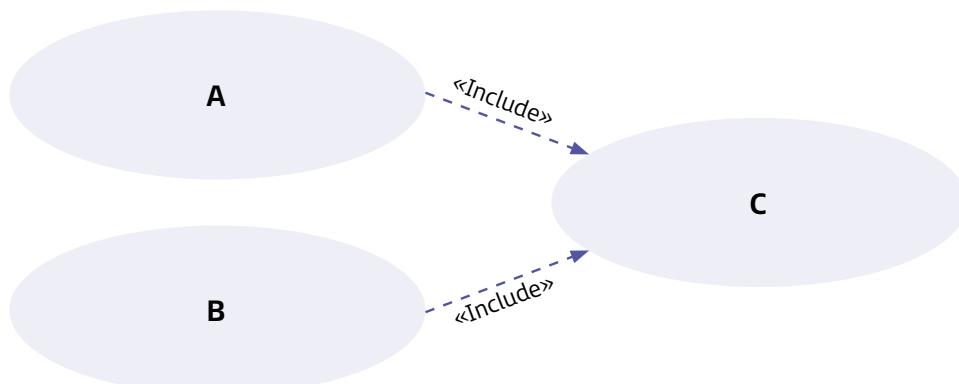
## 7.2.4. RELACIONES EN UN DIAGRAMA DE CASOS DE USO

Podremos encontrar varios tipos de relaciones:

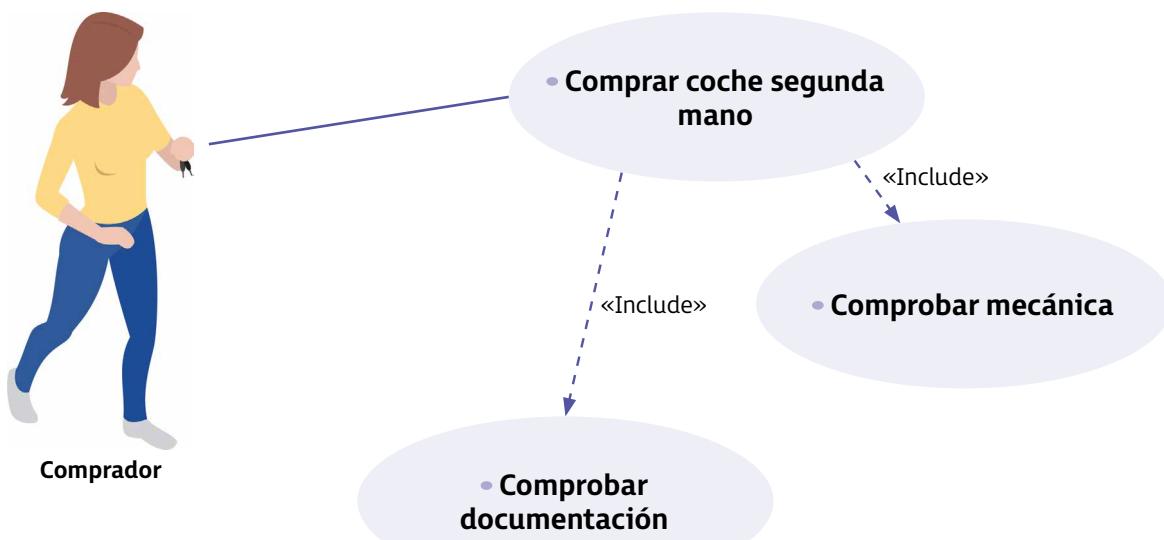
RELACIÓN	FUNCIÓN	NOTACIÓN
Asociación	Línea de comunicación entre actor y caso de uso.	_____
Extensión «<<extend>>	Indica el comportamiento de un caso de uso que se puede extender a otro.	<<extend>> ----->
Generalización de casos de uso	Generalización entre clases. El caso de uso <i>hijo</i> hereda el comportamiento y significado del <i>padre</i> .	_____>
Inclusión «<<include>> o «uses»»	Permite que un caso de uso base incluya el comportamiento de otro caso de uso.	<<include>> ----->

Para aclarar el funcionamiento de cada relación, vamos a especificar el funcionamiento de cada una:

- **Include:** un caso de uso base incorpora explícitamente el comportamiento de otro en algún lugar de su secuencia. La relación de inclusión sirve para enriquecer un caso de uso con otro y compartir una funcionalidad común entre varios casos de uso. Si tenemos, por ejemplo, dos casos de uso A y B que poseen una serie de pasos en común, podrán ponerse esos pasos en un mismo caso de uso C, y X e Y lo incluyen para usarlo.



Un ejemplo sería el de un comprador que quiere adquirir un vehículo de segunda mano, pero, para ello, antes comprueba que la mecánica está bien y que la documentación sea correcta.



- **Extend:** se usará esta relación cuando un caso de uso extiende la funcionalidad de otro caso de uso. Si tenemos un caso de uso B, extenderá la funcionalidad del caso de uso A añadiendo algunos pasos.

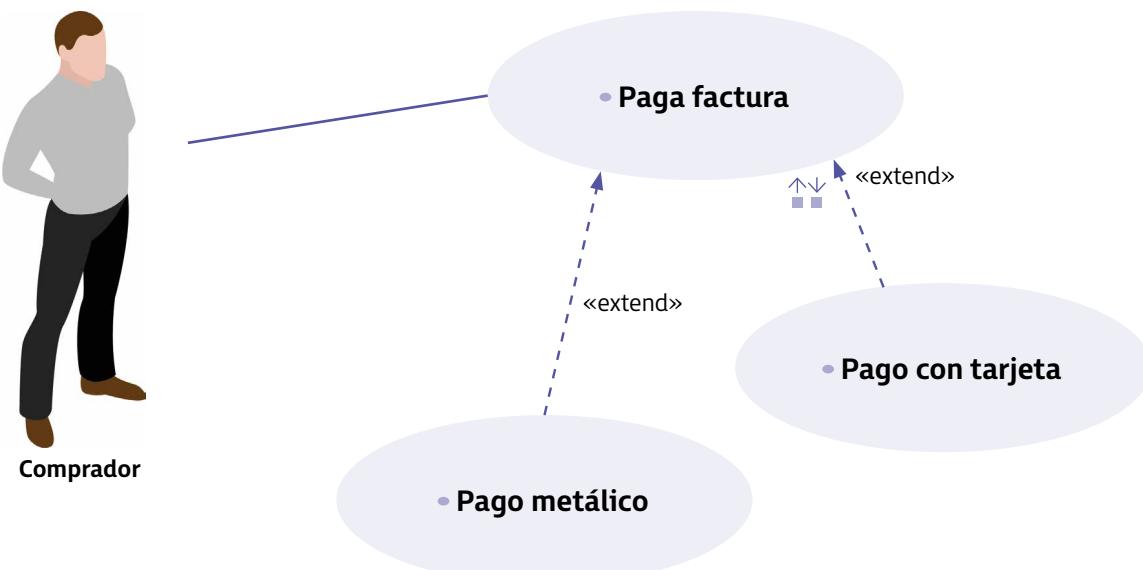
A

B

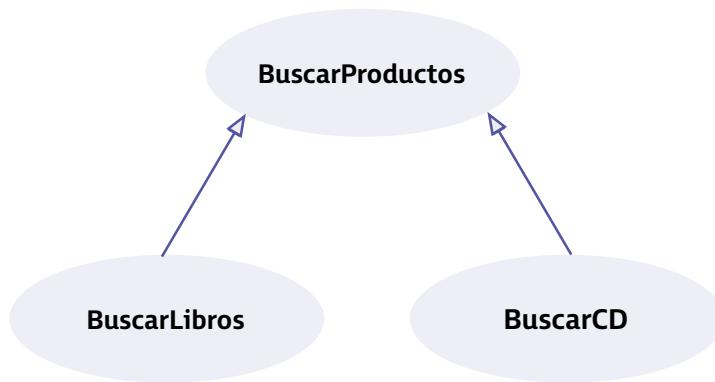
«extend»

En este supuesto, el caso de uso extendido no sabe nada del caso de uso que lo extiende. Por tanto, el caso de uso de extensión (B) no será indispensable que ocurra, y, si lo hace, ofrece un valor extra (extiende) al objetivo original del caso de uso base.

Por ejemplo, un cliente quiere hacer el pago de una factura y puede hacerlo, o bien en metálico, o bien pagando con tarjeta.



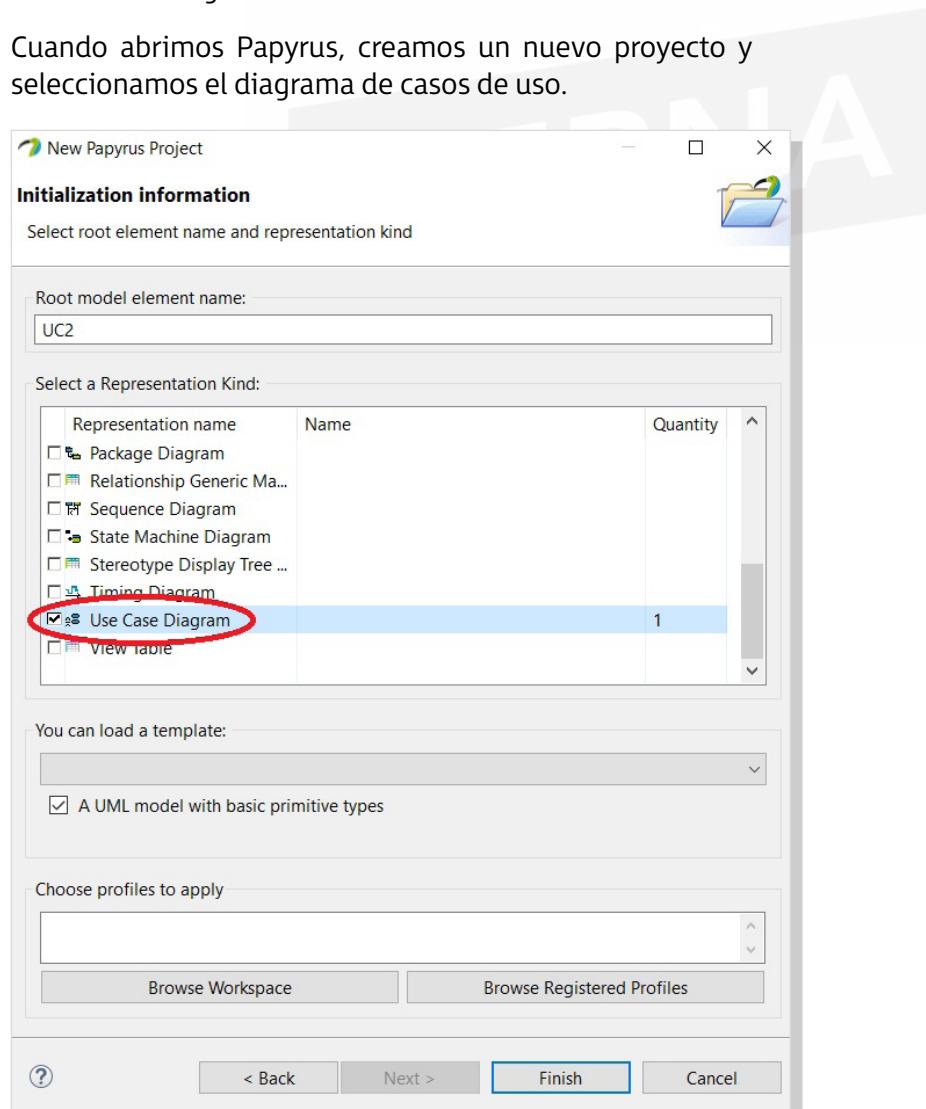
- **Generalización:** la relación de generalización se usa cuando se poseen uno o más casos de uso que son especificaciones de un caso de uso más general.



### 7.2.5. HERRAMIENTAS PARA LA CREACIÓN DE DIAGRAMAS: PAPYRUS

Anteriormente, hemos visto cómo se instala este programa y sus características. En este apartado vamos a abordar la creación de diagramas.

Cuando abrimos Papyrus, creamos un nuevo proyecto y seleccionamos el diagrama de casos de uso.



ponte a prueba.

### ¿Qué hace la relación <<extend>>?

- a) Especifica un caso de uso extendido de otro.
- d) Incluye un caso baso dentro de otro caso de uso.
- c) Extiende una generalización de un caso de uso.
- d) Extiende una especificación de otro caso de uso.

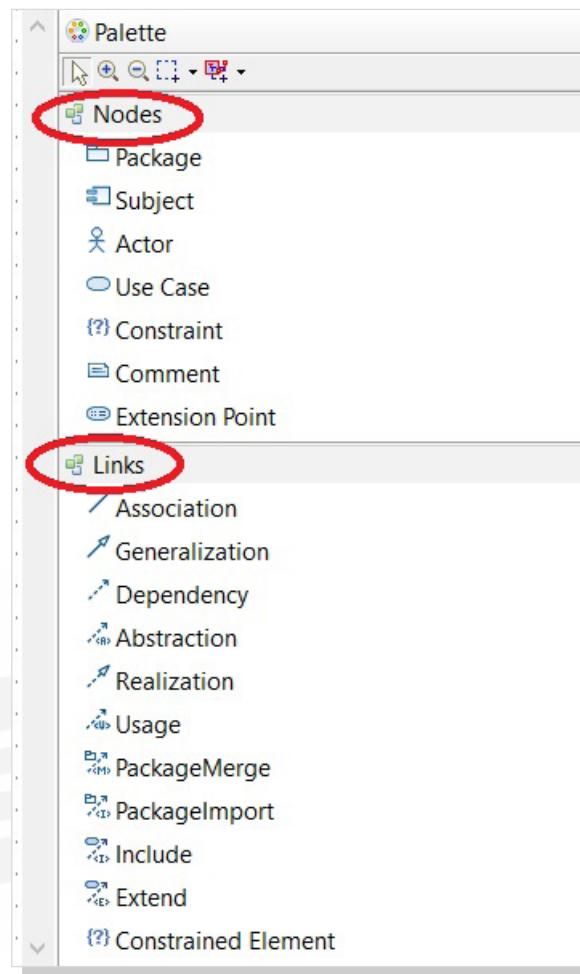
**"Una vez realizado la compra de un producto, nuestro usuario solicita pedir la factura". ¿Cómo relacionaríamos ambos casos de uso?**

- a) "Pedir factura" extiende de "compra producto".
- b) "Compra producto" está incluido en "pedir factura".
- c) "Compra producto" es una clase heredada de "pedir factura".
- d) Ninguna de las opciones es la correcta.

**Podemos tener actores que no sean personas en los casos de uso.**

- a) Verdadero.
- b) Falso.

En el panel por defecto de la parte derecha tenemos el apartado *nodes*, donde elegimos los elementos de nuestro diagrama (actores, casos de uso, etcétera) y el apartado *links*, donde elegimos la relación de estos elementos (<<extend>>, <<include>>, entre otras).



## 7.3. DIAGRAMAS DE INTERACCIÓN

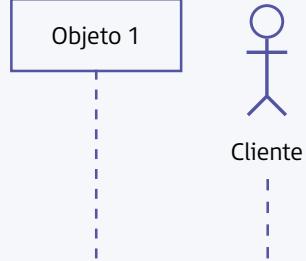
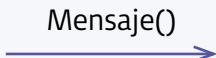
### 7.3.1. DIAGRAMAS DE SECUENCIA. LÍNEA DE VIDA DE UN OBJETO, ACTIVACIÓN, ENVÍO DE MENSAJES

El diagrama de secuencia nos mostrará gráficamente los eventos que fluyen de los actores del sistema. Para la realización de estos diagramas, partimos de los casos de uso. El diagrama tendrá dos dimensiones: la dimensión vertical, que representa el tiempo, y la dimensión horizontal, que representa los roles de la interacción.

Cada objeto será representado por un rectángulo en la parte superior del diagrama, y cada uno tendrá una línea vertical llamada línea de vida, la cual describe la interacción a lo largo del tiempo. Cada línea vertical tendrá flechas hori-

izontales que mostrarán la interacción, y encima de ellas habrá un mensaje. Es importante ver la secuencia porque nos indicará el orden en que van ocurriendo los eventos. A veces, la secuencia puede ir acompañada de una descripción del curso normal de eventos del caso de uso.

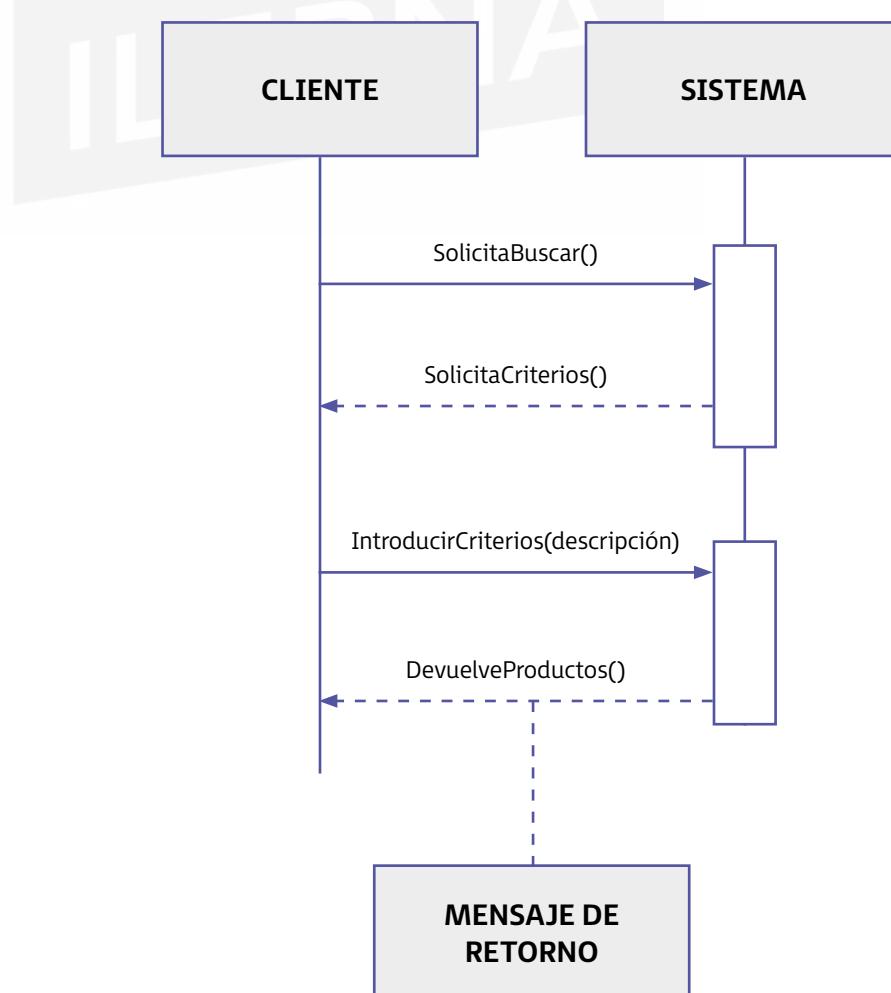
Los **elementos principales** serán:

SÍMBOLO	FUNCIÓN	NOTACIÓN
<b>Marco</b>	Da borde visual al diagrama de secuencia. A la izquierda del marco se escribe la etiqueta <b>sd</b> seguida de un nombre.	
<b>Línea de vida</b>	Representa a un participante durante la interacción.	
<b>Actor</b>	Representa el papel desempeñado por un usuario.	
<b>Mensaje</b>	Mensaje síncrono (queda a la espera de respuesta antes de seguir con su actividad).	
	Mensaje asíncrono (sigue su actividad independientemente, sin esperar a la respuesta).	
	Mensaje de retorno.	
Activación	Opcionales. Representa el tiempo durante el cual se ejecuta una función. Se suele poner cuando está activo un método.	

Los mensajes representan la comunicación entre participantes y se dibujarán con una flecha que irá dirigida desde el participante que los envía hasta el que los ejecuta. Tendrán un nombre, acompañado o no de parámetros.

- **Mensaje síncrono:** cuando se envía un mensaje a un objeto, no se recibe el control hasta que el objeto receptor ha finalizado la ejecución.
- **Mensaje asíncrono:** cuando el emisor que envía un mensaje asíncrono continúa con su trabajo después de ser enviado, es decir, no espera a que el receptor finalice la ejecución del mensaje. Su utilización la podemos ver en sistemas multihilos donde se producen procesos concurrentes.
- **Mensaje de retorno:** representa un mensaje de confirmación. Su uso es opcional.

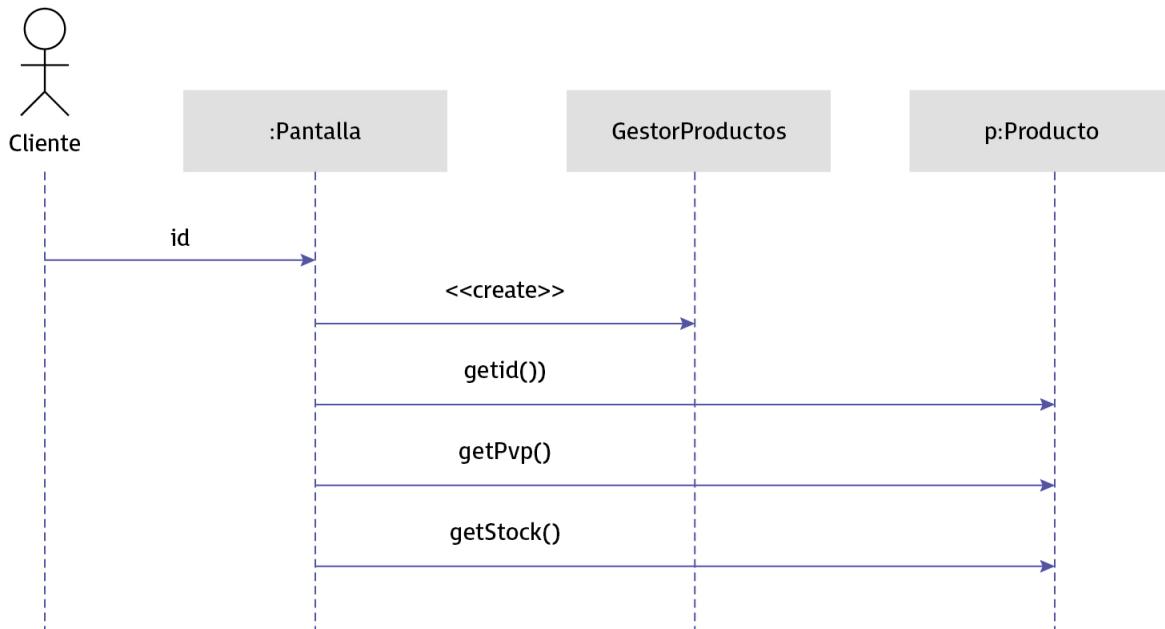
En estos diagramas hemos visto solamente eventos que fluyen desde el cliente hacia el sistema. En la dimensión horizontal, solo se incluían esos dos roles. Dado que trabajamos en un sistema orientado a objetos en el que tenemos varias clases y varios objetos que se comunican unos con otros, los objetos se representarán en la dimensión horizontal.



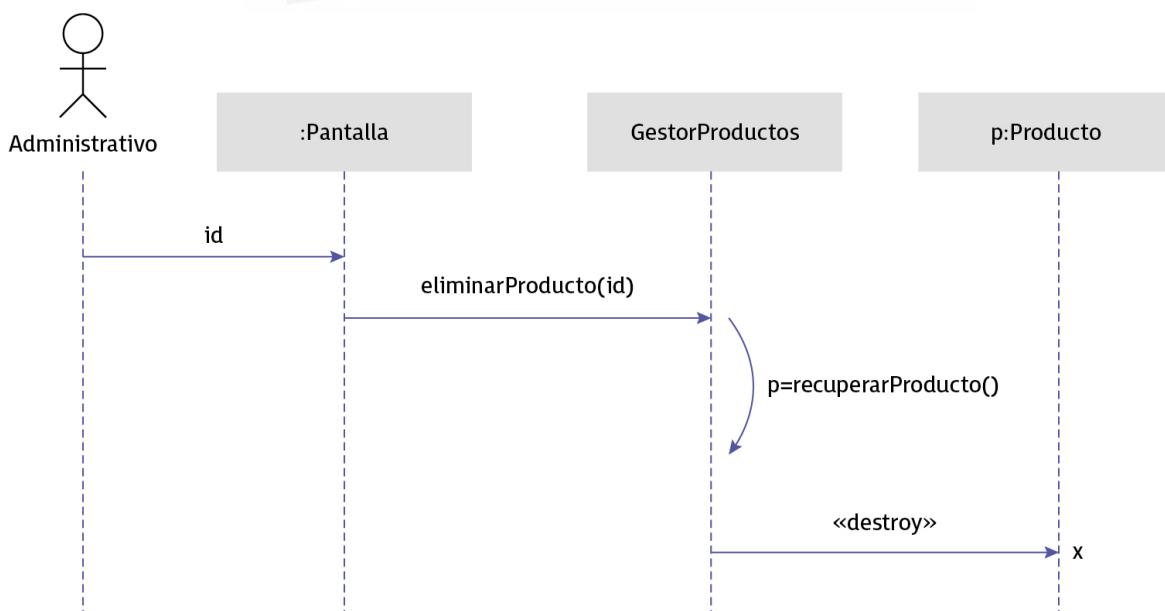
## Creación y destrucción de objetos

En el diagrama de secuencia podemos ver la creación y destrucción de objetos.

La creación se representa mediante un mensaje que termina en el objeto que será creado. Este mensaje puede llevar la identificación <<create>>.



La destrucción finalizará la línea de vida del objeto y se representa mediante una "X" grande en su línea de vida. El mensaje puede llevar la identificación <<destroy>>.



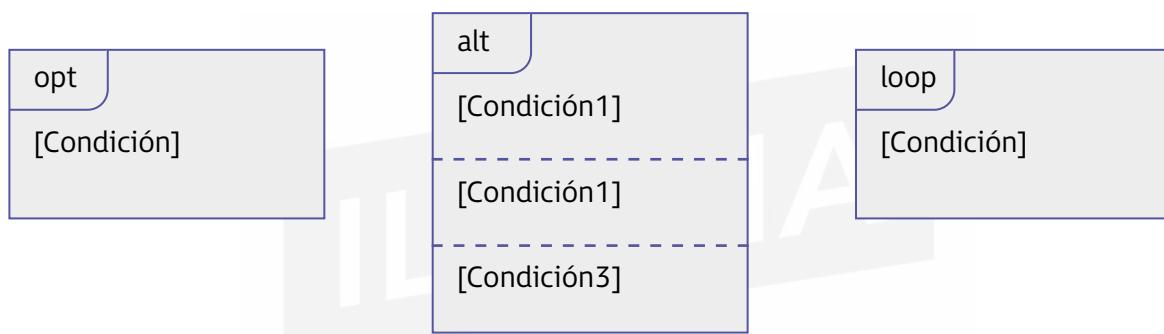
La **autodelegación** o **mensaje reflexivo** es un mensaje que un objeto se envía a sí mismo. Por ello, la flecha del mensaje de vuelta regresa a la misma línea de vida.

## Alternativas y bucles

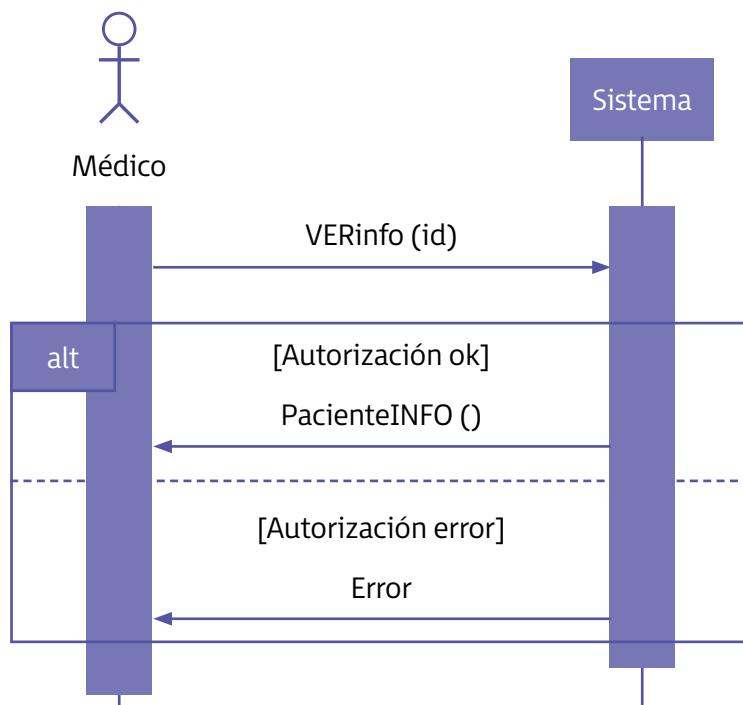
En estos tipos de diagramas podemos introducir extensiones para dar soporte a bucles y alternativas. Podemos llamarlas **fragmentos combinados**, y las hay de varios tipos, entre ellos, las alternativas y los bucles.

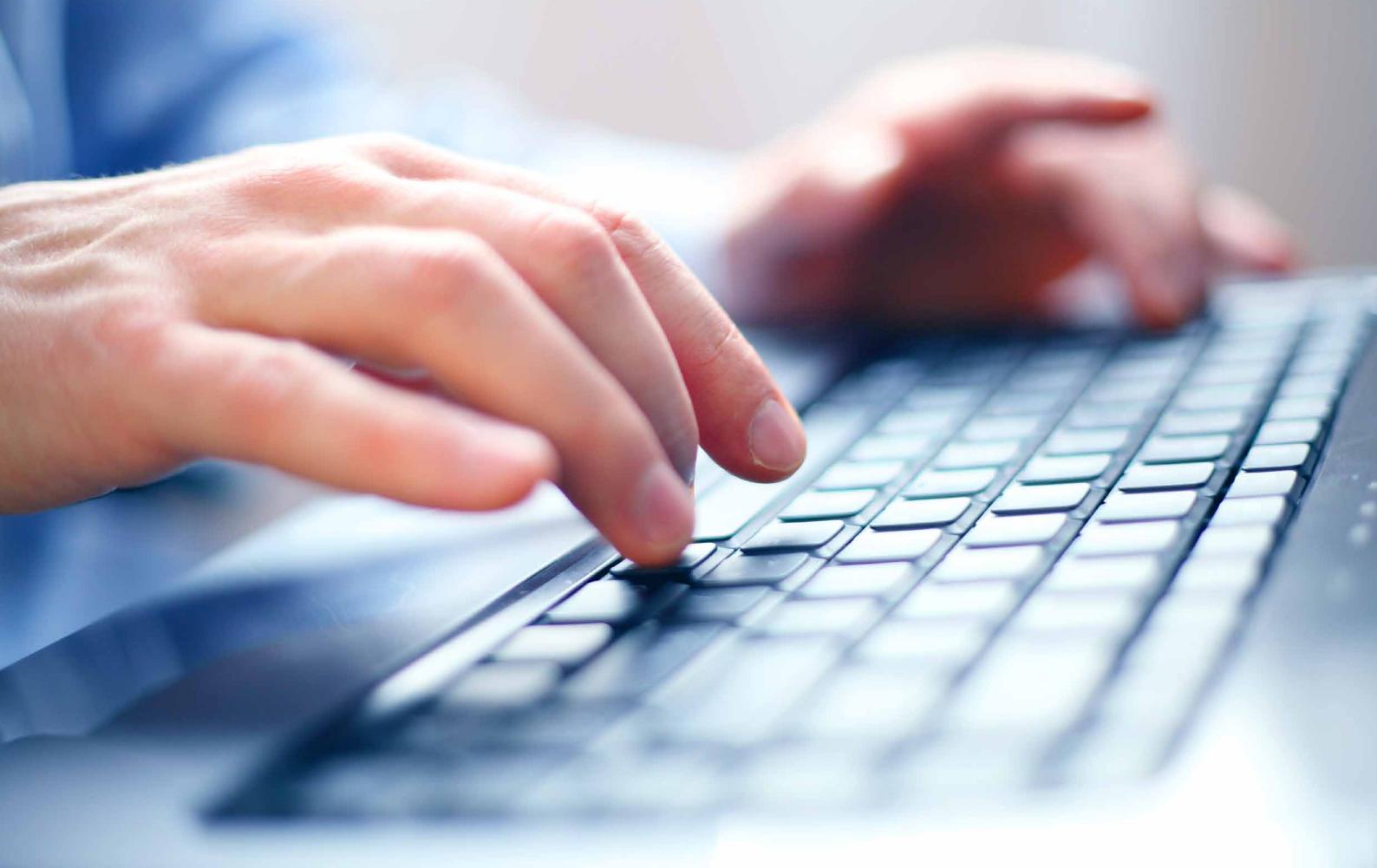
Se pueden representar mediante un marco o caja los eventos que se repiten. Podemos modelar varias alternativas:

- Usando operador ***opt*** seguido de una condición. Si esa condición se cumple, el contenido se ejecuta.
- Usando operador ***alt*** seguido de varias condiciones y al final la palabra clave *else*. Se dividirá en varias zonas según las condiciones que haya. La parte *else* se ejecutará si no se ejecuta ninguna condición.
- Si queremos representar un bucle, lo haremos con el operador ***loop*** seguido de una condición. Lo que está encerrado en el marco se ejecutará siempre que dicha condición se cumpla.



Aquí mostramos un ejemplo utilizando el operador ***alt***.





Un médico solicita al sistema los datos médicos de un paciente:

- Si la solicitud es correcta, el sistema devolverá esos datos.
- Si no, el sistema arrojará un error.



### **ponte a prueba**

#### **Cuando enviamos mensaje síncrono de una clase a otra, ¿qué ocurre?**

- a) La clase que envía el mensaje sigue trabajando y no espera a que el receptor termine su ejecución.
- b) La clase receptora, debe finalizar su ejecución y destruirse.
- c) La clase emisora, debe finalizar su ejecución y destruirse.
- d) La clase que envía el mensaje no recibe el control hasta que la clase receptora ha finalizado la ejecución.

### 7.3.2. DIAGRAMAS DE COMUNICACIÓN. OBJETOS, MENSAJES

En el diagrama de comunicación, cada objeto se representa con una caja, las relaciones entre los objetos con líneas y los mensajes con flechas que indican la dirección. Este diagrama muestra los objetos junto con los mensajes que se envían entre ellos. Se representará la misma información que en el diagrama de secuencia.

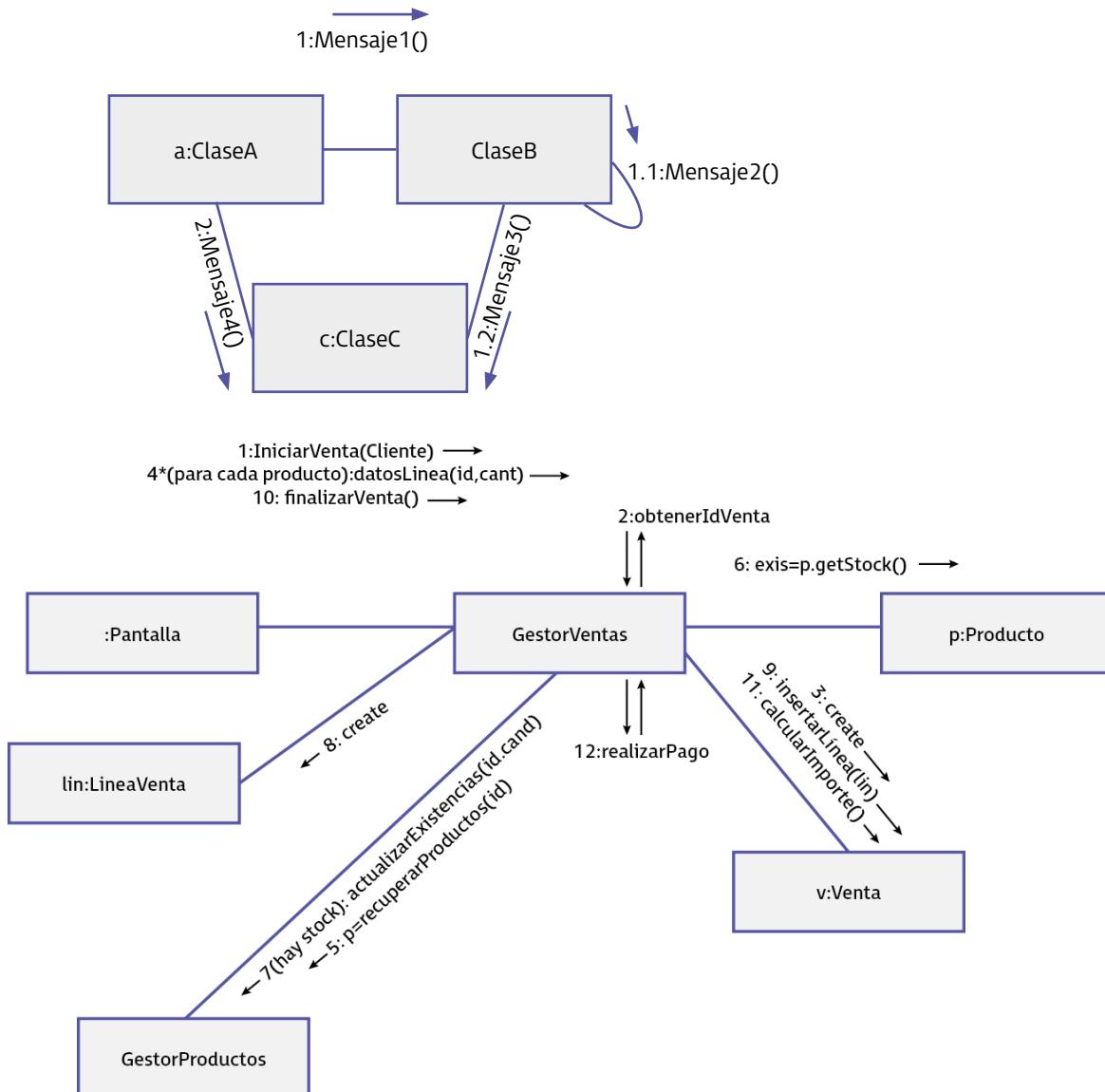
Sin embargo, este diagrama, a diferencia del anterior, se centrará en el contexto y la organización general de los objetos que envían y reciben mensajes. También muestra los objetos y sus relaciones, es decir, los mensajes que se envían entre sí.

El diagrama de colaboración posee los siguientes **elementos**:

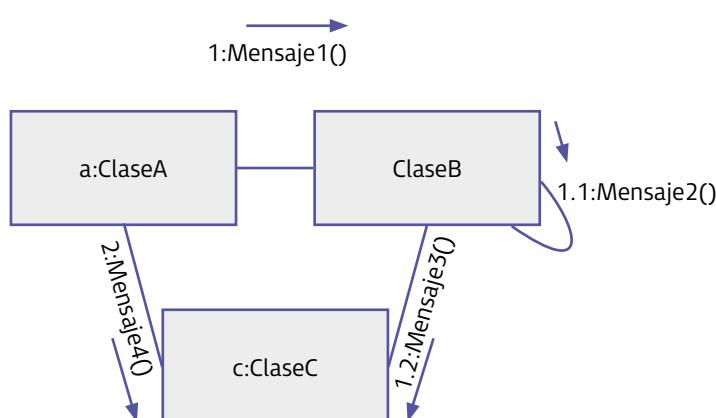
SÍMBOLO	FUNCIÓN	NOTACIÓN
<b>Objetos o roles</b>	Se representan con un rectángulo que contiene el nombre y la clase del objeto en el siguiente formato objeto:Clase.	objeto:Clase
<b>Enlaces</b>	Líneas del grafo que conectan ambos objetos. Se podrán mostrar muchos mensajes en un mismo enlace, pero cada uno con un número de secuencia único.	
<b>Mensajes</b>	Se representan mediante una flecha dirigida con un nombre y un número de secuencia.	1:Mensaje() →
<b>Número de secuencia</b>	Indica el orden de un mensaje dentro de la iteración (repetición). Comenzará con el 1 y se incrementará conforme se envíen mensajes.	
<b>Iteración</b>	Se representa colocando un "*" después del número de secuencia y una condición encerrada entre corchetes.	
<b>Alternativa</b>	Se indican con condiciones entre corchetes. Los caminos alternativos tendrán el mismo número de secuencia, seguido del número de subsecuencia.	
<b>Anidamiento</b>	Se puede mostrar el anidamiento de mensajes con números de secuencia y subsecuencia.	

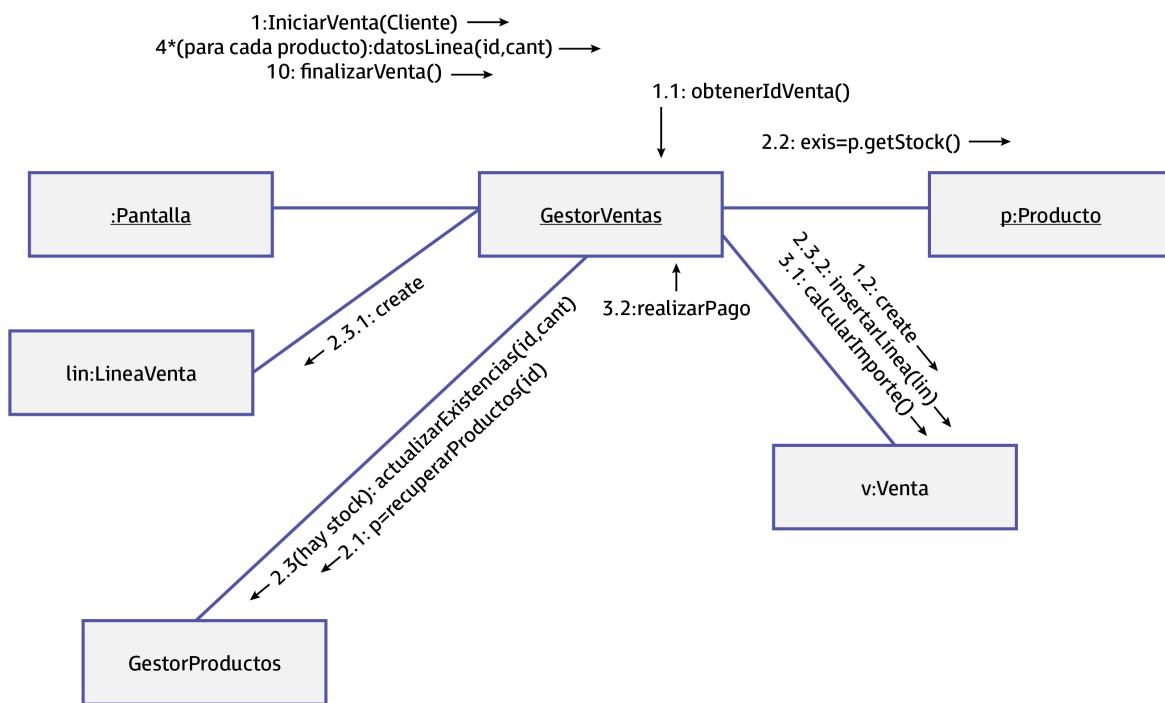
Para numerar los mensajes, se pueden usar varios esquemas:

- **Numeración simple:** comienza en 1 y se incrementa una unidad, y no hay nivel de anidamiento.



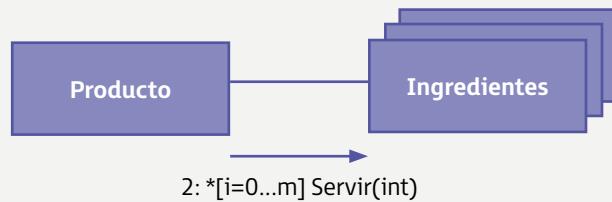
- **Numeración decimal:** tiene varios subíndices para indicar anidamiento de operaciones.





ponte a prueba

¿Qué tipo de mensaje está siendo enviado de una clase a otra?



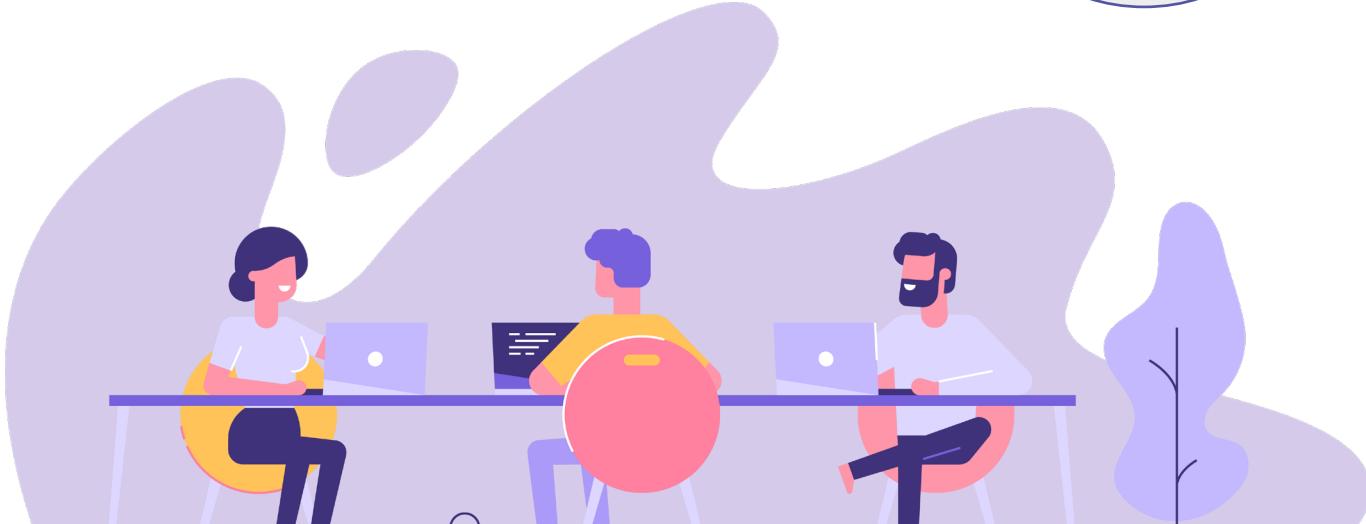
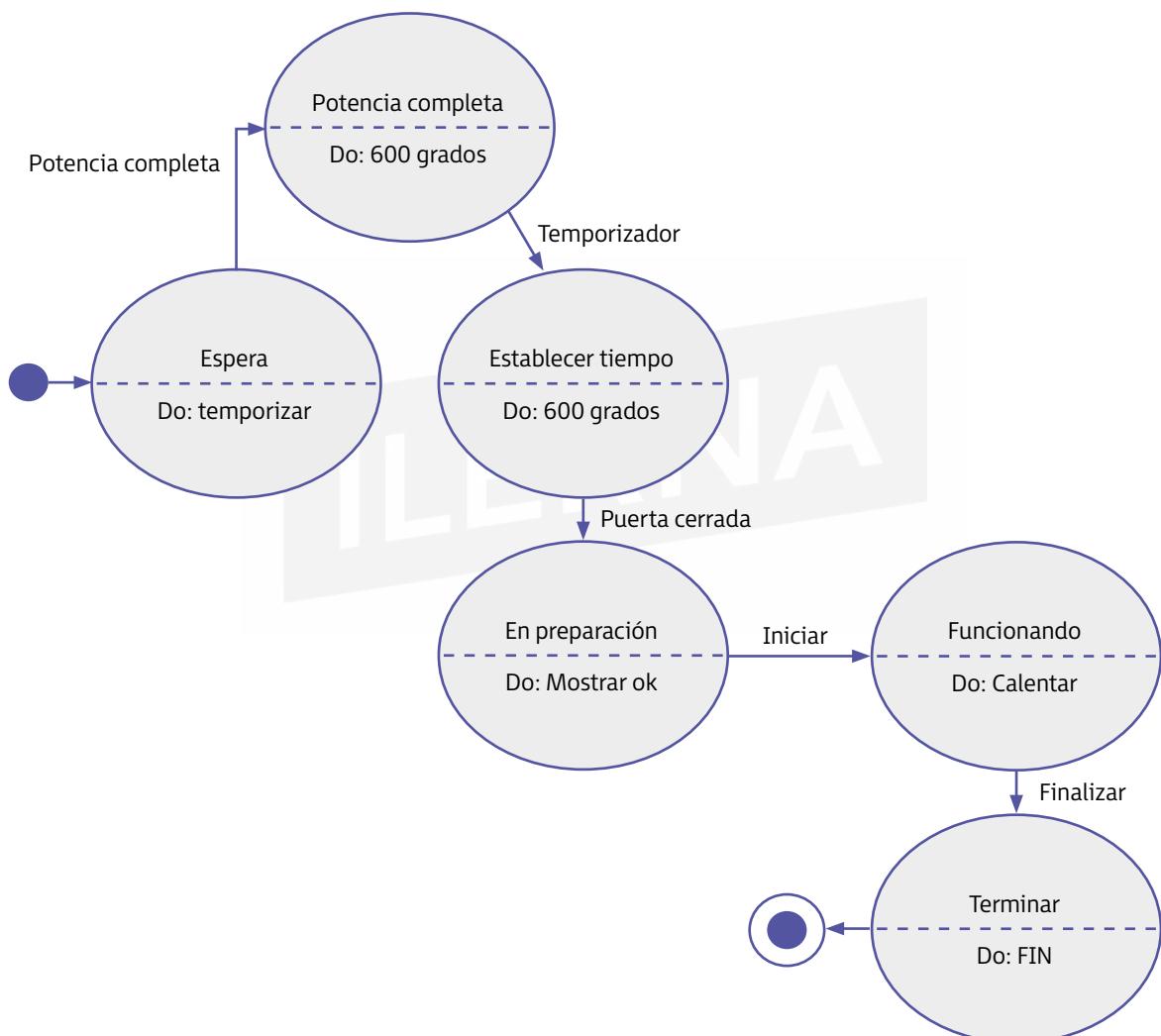
- a) Un anidamiento
- b) Un mensaje iterativo
- c) Un mensaje secuencial
- d) Un mensaje de interfaz

## 7.4. DIAGRAMAS DE ESTADOS. ESTADOS, EVENTOS, SEÑALES, TRANSICIONES

En UML podemos realizar un modelado basado en eventos creando diagramas de estado. Estos diagramas indican cómo evoluciona el sistema a través de las transiciones de un estado a otro.

Suelen incluir información adicional acerca de cómo se ha realizado la transición de un estado a otro.

Vamos a ver un ejemplo:



ESTADO	DESCRIPCIÓN
<b>Espera</b>	El microondas espera la entrada del usuario.
<b>Potencia completa</b>	La potencia se marca a 600 grados.
<b>Establecer tiempo</b>	Tiempo establecido por el usuario.
<b>En preparación</b>	Muestra por pantalla que todo es correcto para empezar a calentar.
<b>Funcionando</b>	El microondas empieza a calentar.
<b>Terminar</b>	Finaliza la acción de calentar del microondas.

ACCIÓN	DESCRIPCIÓN
<b>Potencia completa</b>	El usuario acciona el botón de potencia completa.
<b>Temporizador</b>	El usuario acciona uno de los botones del temporizador.
<b>Puerta cerrada</b>	La puerta del microondas está cerrada.
<b>Iniciar</b>	El usuario acciona el botón <i>iniciar</i> .
<b>Finalizar</b>	El usuario acciona el botón <i>parar</i> .

**ponte a prueba**

**¿Cuál de las siguientes afirmaciones sobre la máquina de estados es correcta?**

- a) Almacena el estado de un objeto en un instante en el tiempo.
- b) El inicio se marca con un círculo relleno.
- c) Ilustra los distintos escenarios de un caso de uso.
- d) Todas las opciones son correctas.



## 7.5. DIAGRAMAS DE ACTIVIDADES. ACTIVIDADES, TRANSICIONES, DECISIONES Y COMBINACIONES

Estos diagramas nos ayudan a modelar el flujo de control de las distintas actividades. Desde un punto de vista de diseño, el diagrama es un conjunto de líneas y nodos. Desde el punto de vista más conceptual, muestra cómo va pasando el control de unas clases a otras, colaborando para conseguir un fin determinado.

Los elementos de un diagrama de actividades son:

- Estados de actividad.
- Estados de acción.
- Transiciones.
- Bifurcaciones.

### 7.5.1. ESTADOS DE ACTIVIDAD Y DE ACCIÓN

Un estado de acción se representa a través de un rectángulo.

Podemos tener dos tipos de acciones:

- Acciones simples.

Preparar un pedido

- Acciones compuestas por una expresión.

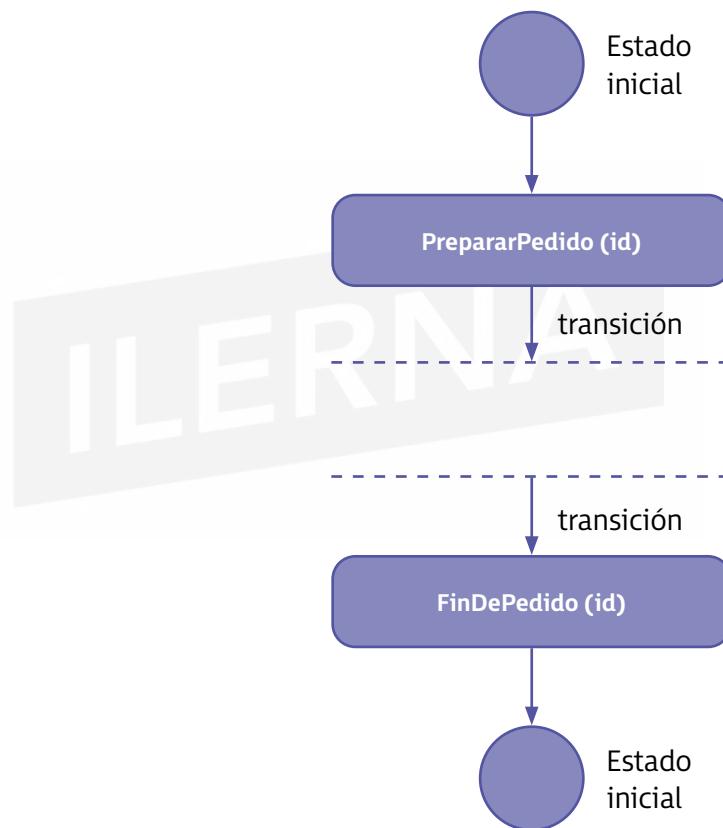
Contador := contador \* 10

Un estado de actividad también se representa mediante un rectángulo y se puede descomponer en más subactividades. Dichos estados pueden ser interrumpidos y, por tanto, tardar en completarse. Podemos encontrar acciones adicionales, como de entrada (*entry*) o de salida (*exit*).

PrepararPedido (id)  
entry/SacarPrimerPedido(id)

### 7.5.2. TRANSICIONES

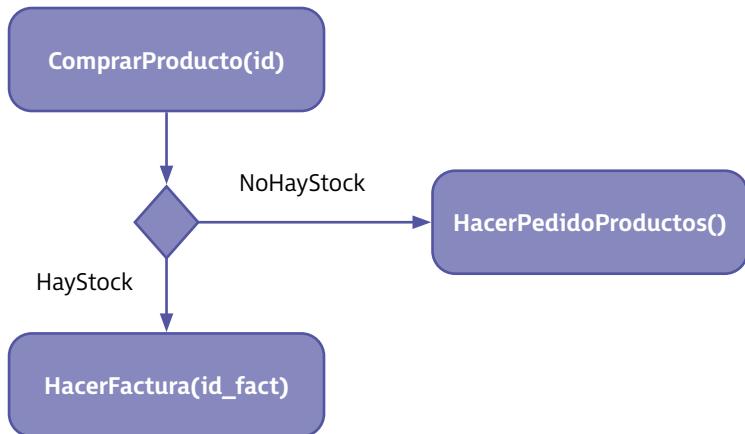
Muestran el paso de un estado a otro tanto de una actividad como de una acción. El flujo de control debe empezar y terminar, por lo que lo indicamos con dos disparadores de inicio y fin.



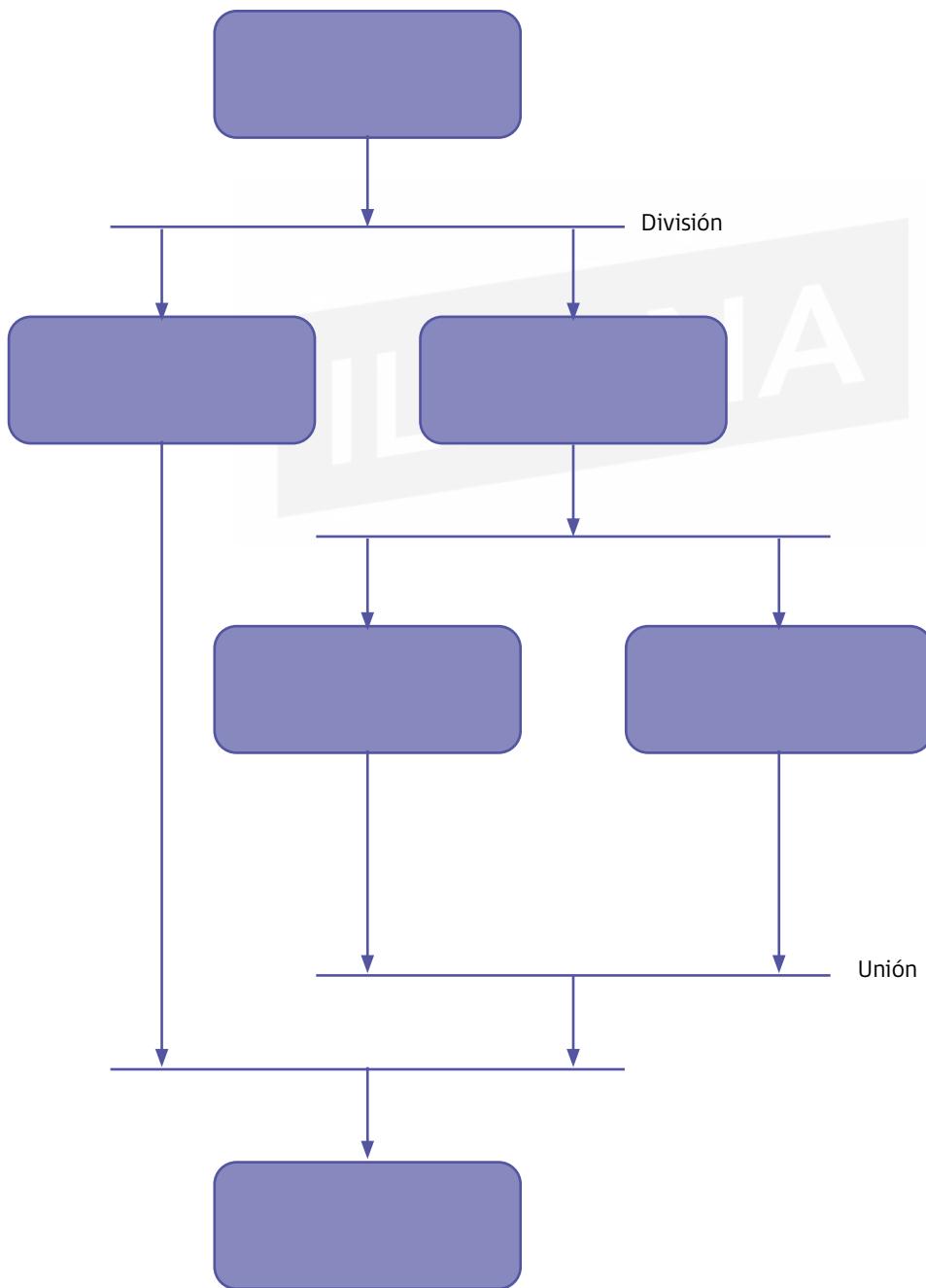
### 7.5.3. BIFURCACIONES Y CAMINOS

El flujo de control no suele ser secuencial, sino que puede presentar varios caminos. Para ello, utilizaremos como símbolo el rombo para las bifurcaciones. En cada transición de salida, se indicará una expresión booleana que se evaluará cada vez que se llegue a la bifurcación.

Las expresiones deben ser excluyentes y se deben visualizar todos los casos posibles.

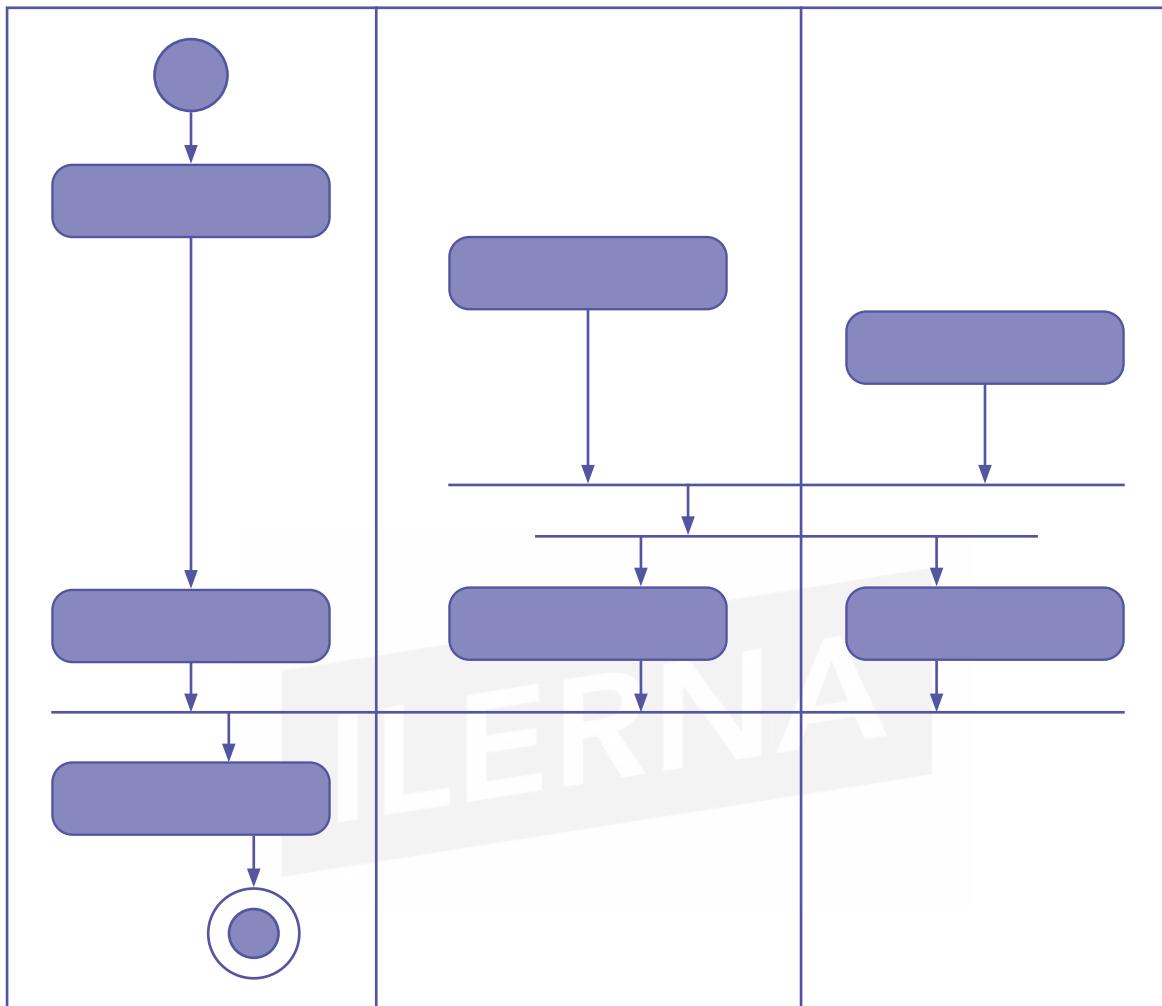


Puede que las tareas sean concurrentes y que se tengan que dividir o unir.



#### 7.5.4. CALLES

Cuando se representan varios tipos de diagramas de actividades de diferentes organizaciones o departamentos, podemos modelarlos en diferentes calles. Cada calle tiene su propio nombre y grupo.



ponte a prueba

**¿Qué componentes son básicos en un diagrama de actividades?**

- a) Nodos de decisión
- b) Flujos de control
- c) Nodo terminal
- d) Todas las opciones son correctas

**Un flujo de control tiene que ser siempre secuencial.**

- a) Verdadero
- b) Falso

## BIBLIOGRAFÍA / WEBGRAFÍA

---

- ❑ Brey, B. B. (2006). *Microprocesadores Intel. Arquitectura, programación e interfaz*. Prentice Hall, 7.<sup>a</sup> Ed.
- ❑ Grau, X. F.; Segura, I. S. (2008). *Desarrollo orientado a objetos en UML*.
- ❑ Pressman, R. S. (2010). *Ingeniería del software. Un enfoque práctico*. McGrawHill, 7.<sup>a</sup> Ed.
- ❑ Ramos Martín, A. (2014). *Entornos de desarrollo*. Madrid: Garceta.
- ❑ Ciencia de software de Halstead: [http://eprints.uanl.mx/9991/1/5\\_Edgar\\_Dominguez\\_et\\_al\\_Ensenanza\\_de.pdf](http://eprints.uanl.mx/9991/1/5_Edgar_Dominguez_et_al_Ensenanza_de.pdf)
- ❑ García Ortega, Víctor H.; Sosa Savedra, Julio C.; Ortega S., Susana; Tovar, Rubén H. (2009) *Microprocesador didáctico de arquitectura RISC implementado en un FPGA*. E-Gnosis. <https://www.redalyc.org/pdf/730/73012215014.pdf>
- ❑ Institute of Electrical and Electronics Engineers Inc (1995). *Computer pioneers by J. A. N. Lee*. <https://history.computer.org/pioneers/halstead.html>
- ❑ MacTutor History of Mathematics Archive: [https://mathshistory.st-andrews.ac.uk/Biographies/Von\\_Neumann/](https://mathshistory.st-andrews.ac.uk/Biographies/Von_Neumann/)
- ❑ R.Villarroel,E.Fernández-Medina,J.Trujillo,M.Piattini.(2005)*Un profile de UML para diseñar almacenes de datos seguros*. [https://www.researchgate.net/profile/Mario\\_Piattini/publication/3454984\\_A\\_UML\\_profile\\_for\\_designing\\_secure\\_data\\_warehouses/links/0deec5391f6203f2a5000000.pdf](https://www.researchgate.net/profile/Mario_Piattini/publication/3454984_A_UML_profile_for_designing_secure_data_warehouses/links/0deec5391f6203f2a5000000.pdf)



## solucionario

### 1.1. El software del ordenador

¿En qué tipo de método de distribución estaría el siguiente software?

b) Shareware.

El software libre puede ser vendido.

a) Verdadero.

*En algunos casos los programas libres se distribuyen gratuitamente, y en otras ocasiones por un precio determinado.*

### 1.2.1. Programa y componentes del sistema informático

¿Qué tipo de lenguaje de programación es Python?

a) Alto nivel.

¿Qué función realiza la ALU?

d) Todas las opciones son correctas.

### 1.4. Tipos de lenguajes de programación. Clasificación y características de los lenguajes más difundidos

¿Qué capacidad (en bits) tiene el registro EAX?

c) 32 bits.

¿Cuál de los siguientes lenguajes no son de alto nivel?

d) Ensamblador.

### 1.5. Fases del desarrollo de una aplicación: análisis, diseño, codificación, pruebas, documentación, mantenimiento y explotación

¿Qué desventaja tiene el modelo en espiral?

c) Es difícil evaluar los riesgos.

Un cliente pide que se realice una base de datos de su web. ¿Qué modelo de desarrollo es el más adecuado?

a) Modelo en cascada con realimentación.

*Es un caso muy definido y los requisitos son estables.*

En el modelo en V, las pruebas se representan en la parte derecha y en la parte izquierda, las especificaciones del sistema.

a) Verdadero.

### 1.5.1 Análisis

En la fase de análisis, realizamos los diagramas de clases para modelar el sistema.

b) Falso.

*En la etapa de diseño es donde realizamos el modelado de UML.*

En la fase de análisis, capturamos los requisitos no funcionales.

a) Verdadero.

### 1.5.2. Diseño

¿Qué es el pseudocódigo?

c) Una representación de nuestros algoritmos.

¿Cuál de las siguientes representaciones son utilizadas para la fase de diseño?

d) B y C son correctas.

### 1.5.3. Codificación

¿Cómo debe comenzar un archivo en java?

c) Nombre de la clase, la información de la versión, la fecha y el aviso de derechos de autor.

Es recomendable que las clases tengan una longitud de líneas de código superior a 80 caracteres.

b) Falso.

*Es recomendable menos de 80 líneas. Si tiene más, probablemente podemos hacer la clase más modular (“divide y vencerás”).*

### 1.5.4. Pruebas

Estamos realizando las pruebas de un método que realiza el factorial de un número. Estamos introduciendo el número 4 y nos da como salida 24. ¿Qué pruebas estamos llevando a cabo?

b) Prueba de caja negra.

### 1.6. Metodologías ágiles

¿En qué metodología se trabaja por “sprints”?

a) SCRUM.

¿Qué caracteriza la metodología “programación extrema”?

d) Todas las respuestas son correctas.



## solucionario

### 2.2. Instalación de un entorno de desarrollo

Con el entorno de desarrollo de Eclipse podemos modelar en UML.

a) Verdadero.

*Podemos instalar plugins que nos permite modelar en UML.*

¿Qué caracteriza a la herramienta MySQL Workbench?

d) Todas las respuestas son correctas.

¿Qué funcionalidades nos proporciona la herramienta CASE?

d) Todas las respuestas son correctas.

### 3.2.1. Prueba de unidad

¿Qué realiza la siguiente instrucción en Junit?

```
assertTrue(String mensaje, boolean expression)
```

a) Comprueba que la expresión se evalúe a true. Si no es true y se incluye el string, al producirse, error se lanzará un mensaje.

### 3.2.2. Prueba de integración

¿Qué es una integración big bang?

b) Una prueba donde integramos todos los módulos sin niveles establecidos.

### 3.2.3. Prueba de validación

En las pruebas beta, el desarrollador se encuentra presente junto con el cliente.

b) Falso.

*En las pruebas alfa, trabajan conjuntamente el desarrollador y el cliente.*

En las pruebas omega, el desarrollador está presente junto con el cliente.

b) Falso.

*No existen las pruebas omega.*

### 3.3.1. Prueba del camino básico

Según el siguiente grafo, ¿cuántas regiones tiene?

b) 3.

Aquellos programas con una complejidad mayor de 50 son programas de alto riesgo y poco testeables.

a) Verdadero.

### 3.3.2. Partición o clases de equivalencia

¿Cuáles son los dos tipos en los que podemos dividir las clases de equivalencia?

b) Válidas y no válidas.

### 3.3.3. Análisis de valores límite

Si estamos testeando un módulo que tiene de rango de entradas [0-5], ¿qué valores deberíamos probar?

b) -1, 0, 5, 6.

### 3.6. Calidad del software

Una de las métricas de Halstead es la medida del esfuerzo.

a) Verdadero.

La fórmula  $N = N_1 + N_2$ , donde  $N_1$  es el número total de operadores y  $N_2$  es el número total de operandos, ¿qué calcula?

d) La longitud de un código.

### 4.1.1. Cuándo refactorizar. Bad smells

Es mejor realizar un método o clase lo más extenso posible para cubrir todos los posibles casos y pruebas.

b) Falso.

*Cuando un método es demasiado extenso, probablemente, se pueda subdividir en métodos más pequeños (refactorización).*

Si tenemos que realizar un cambio en un módulo debido a que cambian los requisitos y este cambio afecta a todos los módulos de sistema, ¿qué bad smell encontramos?

a) Cirugía a tiro de pistola.

### 4.2. Control de versiones. Estructura de las herramientas de control de versiones.

En SVN, el tronco es la línea principal del desarrollo del proyecto.

a) Verdadero.

### 5. Introducción al UML

¿Qué afirmación sobre la UML es correcta?

d) Todas las respuestas son correctas.

Los diagramas de interacción forman parte de los diagramas de comportamiento.

a) Verdadero.



## solucionario

### 6.2.2. Relaciones

¿Qué cardinalidad corresponde a este tipo de relaciones?

- b) 1 a varios.

"Debemos de registrar el nombre, apellidos y número de teléfono de una persona en nuestro aplicativo. También debemos modelar las relaciones familiares de progenitor y cónyuge."

¿De qué forma podemos modelar este caso?

- b) Como una doble relación reflexiva.

La clase A depende de la clase B, por lo que A no conoce la existencia de B.

- b) Falso.

*Al A depender de B, es A quien conoce la existencia de B; no al revés.*

### 6.3 Herramientas para el diseño de diagramas

¿Con cuál de los siguientes programas puedo modelar en UML?

- d) Todas las respuestas son correctas.

Papyrus UML es un entorno de modelado de Eclipse.

- a) Verdadero.

### 7.2. Diagramas de casos de uso. Actores, escenario, relación de comunicación.

¿Qué hace la relación <>extend>>?

- a) Especifica un caso de uso extendido de otro.

"Una vez realizado la compra de un producto, nuestro usuario solicita pedir la factura". ¿Cómo relacionaríamos ambos casos de uso?

- a) "Pedir factura" extiende de "compra producto".

Podemos tener actores que no sean personas en los casos de uso.

- a) Verdadero.

### 7.3.1. Diagramas de secuencia. Línea de vida de un objeto, activación, envío de mensajes.

Cuando enviamos mensaje síncrono de una clase a otra, ¿qué ocurre?

- d) La clase que envía el mensaje no recibe el control hasta que la clase receptora ha finalizado la ejecución.

### 7.3.2. Diagramas de comunicación. Objetos, mensajes.

¿Qué tipo de mensaje está siendo enviado de una clase a otra?

- b) Un mensaje iterativo.

### 7.4. Diagramas de estados. Estados, eventos, señales, transiciones

¿Cuál de las siguientes afirmaciones sobre la máquina de estados es correcta?

- d) Todas las opciones son correctas.

### 7.5. Diagramas de actividades. Actividades, transiciones, decisiones y combinaciones

¿Qué componentes son básicos en un diagrama de actividades?

- d) Todas las opciones son correctas.

Un flujo de control tiene que ser siempre secuencial.

- b) Falso.

*Un flujo de control puede presentar caminos alternativos.*