

Módulo 5

Entornos de desarrollo

```

function updatePhotoDescription() {
    if (descriptions.length > (page * 9) + (currentimage.substring(0) - 1)) {
        document.getElementById("bigimageDesc").innerHTML = descriptions[page * 9 + (currentimage.substring(0) - 1)];
    }
}

function updateAllImages() {
    var i = 1;
    while (i < 10) {
        var elementId = "foto" + i;
        var elementIdBig = "bigimage" + i;
        if (page * 9 + i - 1 < photos.length) {
            document.getElementById(elementId).src = "images/" + photos[page * 9 + i - 1];
            document.getElementById(elementIdBig).src = "images/" + photos[page * 9 + i - 1];
        } else {
            document.getElementById(elementId).src = "images/" + photos[photos.length - 1];
            document.getElementById(elementIdBig).src = "images/" + photos[photos.length - 1];
        }
        i++;
    }
}

```

UF1. DESARROLLO DE SOFTWARE.....	3
1. Desarrollo de software.....	3
1.1. El Software del ordenador	3
1.2. Concepto de programa informático	6
1.3. Código fuente, código abierto y código ejecutable; máquinas virtuales.....	8
1.4. Tipos de lenguajes de programación. Clasificación y características de los lenguajes más difundidos.....	12
1.5. Fases del desarrollo de una aplicación: análisis, diseño, codificación, pruebas, documentación, mantenimiento y explotación.....	19
1.6. Proceso de obtención de código a partir del código fuente; herramientas implicadas.....	46
2. Instalación y uso de entornos de desarrollo.	52
2.1. Funciones de un entorno de desarrollo.	52
2.2. Instalación de un entorno de desarrollo.	53
2.3. Herramientas para el modelado de datos	58
UF2. OPTIMIZACIÓN DE SOFTWARE	67
1. Diseño y realización de pruebas.....	67
1.1. Planificación de pruebas.	67
1.2. Estrategias de pruebas de software	68
1.3. Pruebas de código: cubrición, valores límite y clases de equivalencia.....	71
1.4. Pruebas unitarias con JUNIT.....	82
2. Documentación y optimización.	85
2.1. Refactorización. Concepto. Limitaciones. Patrones de refacción más usuales.	85
2.2. Control de versiones. Estructura de las herramientas de control de versiones.	89
2.3. Documentación. Uso de comentarios. Alternativas.	99
UF3. INTRODUCCIÓN AL DISEÑO ORIENTADO A OBJETOS	103
1. Elaboración de diagramas de clases.	103
1.1. Objetos	105
1.2. Diagramas de clases.....	106
1.3. Herramientas para el diseño de diagramas	114
2. Elaboración de diagramas de comportamiento.....	119
2.1. Tipo. Campo de aplicación.....	119
2.2. Diagramas de casos de uso. Actores, escenario, relación de comunicación.	120
2.3. Diagramas de secuencia. Línea de vida de un objeto, activación, envío de mensajes.	125
2.4. Diagramas de colaboración. Objetos, mensajes.....	129
BIBLIOGRAFÍA.....	132

UF1. Desarrollo de Software

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.

Empezaremos a ver qué es el software del ordenador y el significado y los componentes de un programa, así como el ciclo de vida de dicho software. También veremos los distintos tipos de lenguajes de programación que podremos utilizar, sus características y las distintas clasificaciones. Por último, abordaremos las fases que deberemos pasar para desarrollar la aplicación y estudiaremos cómo es el código fuente.

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 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. En cambio, el ordenador posee otra parte lógica llamada **software**, encargada de dar instrucciones al hardware y hacer ejecutar 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**. Además, si tenemos en cuenta la **licencia** por la cual se distribuye, se clasifica en: **software libre**, **software propietario** y **software de dominio público**.

	<p>BASADAS EN EL TIPO DE TRABAJO QUE REALIZAN</p> <ul style="list-style-type: none"> • De sistema • De aplicación • De programación • De uso específico • Multimedia
	<p>BASADAS EN EL MÉTODO DE DISTRIBUCIÓN</p> <ul style="list-style-type: none"> • Shareware • Freeware • Adware
	<p>TENIENDO EN CUENTA LA LICENCIA</p> <ul style="list-style-type: none"> • Software libre • Software propietario • Software de dominio público

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, etc.
- **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, etc.
- **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

Según esta clasificación, distinguimos tres tipos de software:

- **Shareware:** donde los usuarios pueden pagar y después descargar el aplicativo desde internet. Por ejemplo, PowerDVD.
- **Freeware:** donde los usuarios Software pueden descargar el aplicativo de forma gratuita, pero que mantiene los derechos de autor. Por ejemplo, Avast.
- **Adware:** es un aplicativo donde se ofrece publicidad incrustada, incluso en la instalación del mismo. 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 tres 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 a otros usuarios.
 - Libertad para poder mejorar el programa y publicar las modificaciones realizadas.
- **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. Pueden darse dos variantes vistas anteriormente: *freeware* y *shareware*.

- **Software de dominio público:** este software no pertenece a ningún propietario y carece de licencia, por lo que todo el mundo lo puede utilizar. Incluso podremos realizar una oferta para adquirirlo bajo el código fuente de dominio público.

La licencia que más se usa en el software libre es la licencia **GPL** (*GNU General Public License – Licencia Pública General*) que nos dejará usar y cambiar el programa, con el único requisito que se hagan públicas las modificaciones realizadas.

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

<http://www.gnu.org/licenses/license-list.es.html#SoftwareLicenses>

1.2. Concepto de programa informático

Un **programa informático** es un grupo de instrucciones que están escritas en un lenguaje de programación sobre el que se aplican una serie de datos para resolver un problema. Es decir, el ordenador necesita que esté en lenguaje máquina y para ello tendremos que usar un **compilador**. Una vez hecho esto tendremos que procesar todas las instrucciones pasándolas a la memoria principal.

1.2.1. Programa y componentes del sistema informático

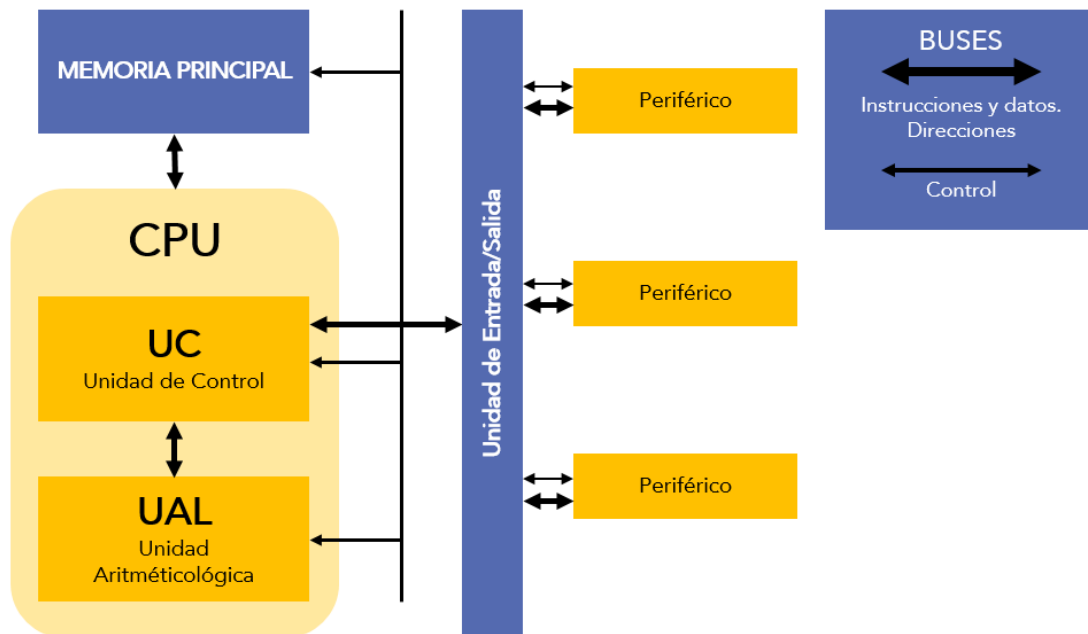
Si queremos iniciar un programa necesitaremos recursos hardware del ordenador, como son el procesador, la memoria RAM, dispositivos E/S, etc. Las instrucciones para inicializar el programa se cargan en la memoria principal y se ejecutarán en la CPU (en inglés, *Central Processing Unit*).

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 los que almacenan la información temporal, 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 contenidos en la posición apuntada por el RDM.
- **Decodificador de instrucción (DI):** extrae y analiza el código de la instrucción contenida en el RI y, además, genera las señales para que se ejecute correctamente la acción.

- **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.

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 pueden ser los diagramas de flujo o el pseudocódigo.

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

1.3.1. Tipos de código

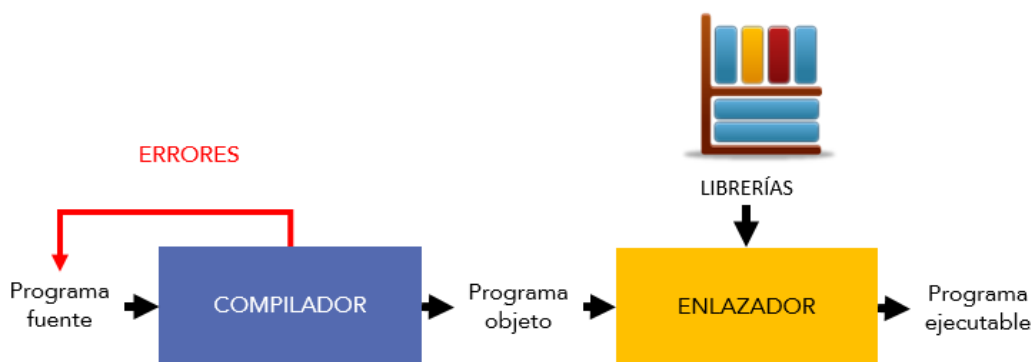
Cuando escribimos un código pasa por distintos estados hasta que se ejecuta:

- **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 los diagramas de flujo o pseudocódigos. No se puede ejecutar directamente en el ordenador.

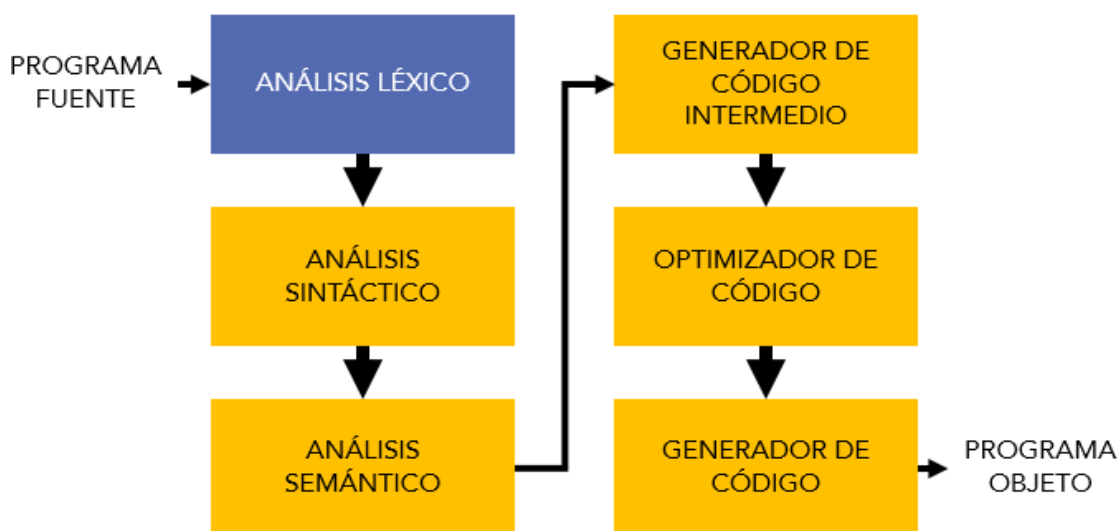
- **Código objeto:** es el código que se crea tras realizar la compilación del código fuente. Este código no es entendido ni por el ordenador ni por nosotros. Es una representación intermedia de bajo nivel.
- **Código ejecutable:** este código se obtiene tras unir el código objeto con varias librerías 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 `resta = 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, etc.
- **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 objetos, las librerías necesarias para que se pueda producir un programa ejecutable. Si se hacen 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:

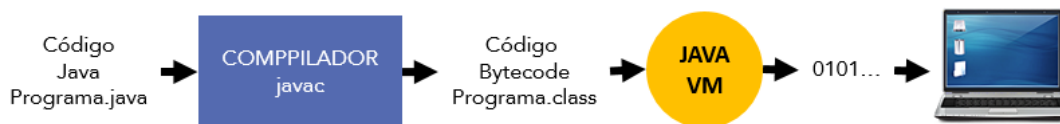
- **Máquinas virtuales de sistema.** Nos permiten virtualizar máquinas con distintos sistemas operativos en cada una. Un ejemplo son los programas *VMware Workstation* o *Virtual Box* que podremos usar para probar nuevos sistemas operativos o ejecutar programas.
- **Máquinas virtuales de proceso.** Se ejecutan como un proceso normal dentro de un SO y solo soporta 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.

Las máquinas virtuales requieren de grandes recursos por lo que hay que tener cuidado y ejecutarlas en ordenadores capaces de soportar los procesos que requieren dichas máquinas para que no nos funcionen lentas o se colapsen.

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:

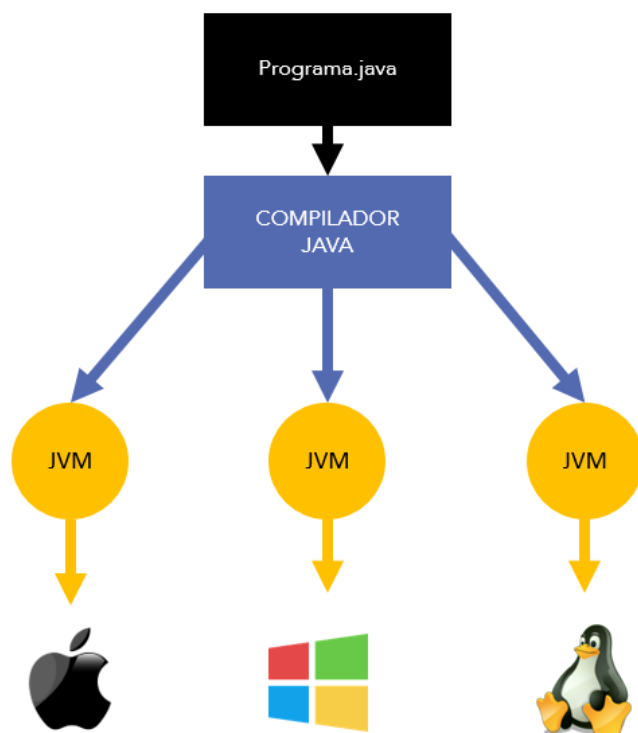
1º) El código fuente estará escrito en archivos de texto planos con la extensión .java.

2º) El compilador **javac** generará uno o varios archivos siempre que no se produzcan errores y tendrán la extensión .class.

3º) Este fichero .class contendrá un lenguaje intermedio entre el ordenador y el SO y se llamará *bytecode*.

4º) La *Java VM* coge y traduce el *bytecode* en código binario para que el procesador de nuestro ordenador sea capaz de reconocerlo.

Los ficheros .class podrán ser ejecutados en múltiples plataformas.



Entre las tareas que puede realizar la máquina virtual Java pueden estar:

- La reserva de espacio para objetos creados y liberar aquella memoria que no se usa.
- Comunicación con el sistema en el que se ejecuta la aplicación para varias funciones.
- Observar que se cumplen las normas de seguridad para las aplicaciones Java.

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. No obstante, cabe destacar que no una desventaja demasiado crítica.

Para poder instalar la Java VM tendremos que acceder a la siguiente url: <http://www.oracle.com/technetwork/java/javase/downloads/index.html>, pulsar en el botón *download* donde aparece JRE (*Java Runtime Environment*) y descargar el fichero que nos interese según el sistema operativo que usemos. Una vez descargado, la instalación es fácil, ya que solo tendremos que ir siguiendo los pasos indicados. Cuando finalice la instalación puede que debamos reiniciar el equipo.

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:

Según su nivel de abstracción:	Lenguajes de bajo nivel
	Lenguajes de nivel medio
	Lenguajes de alto nivel
Según la forma de ejecución:	Lenguajes compilados
	Lenguajes interpretados
Según el paradigma de programación:	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.

```

-u 100 1a
OCFD:0100 BA0B01 MOV DX,010B
OCFD:0103 B409 MOV AH,09
OCFD:0105 CD21 INT 21
OCFD:0107 B400 MOV AH,00
OCFD:0109 CD21 INT 21
-d 10b 13f
OCFD:0100 48 6F 6C 61 2C Hola,
OCFD:0110 20 65 73 74 65 20 65 73-20 75 6E 20 70 72 6F 67 este es un prog
OCFD:0120 72 61 6D 61 20 68 65 63-68 6F 20 65 6E 20 61 73 rama hecho en as
OCFD:0130 73 65 6D 62 6C 65 72 20-70 61 72 61 20 6C 61 20 sembler para la
OCFD:0140 57 69 68 69 70 65 64 69-61 24 Wikipedia$

```

Fuente: https://es.wikipedia.org/wiki/Lenguaje_ensamblador

– Lenguajes de nivel medio:

Posee 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 para usarlo lo hacemos con palabras que solemos utilizar. 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, Pascal, etc.

• 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. Devolverá errores si el lenguaje fuente está mal escrito y lo ejecutará si el lenguaje destino es ejecutable por la máquina. Ejemplo: C, C++, C#, Objective-C.

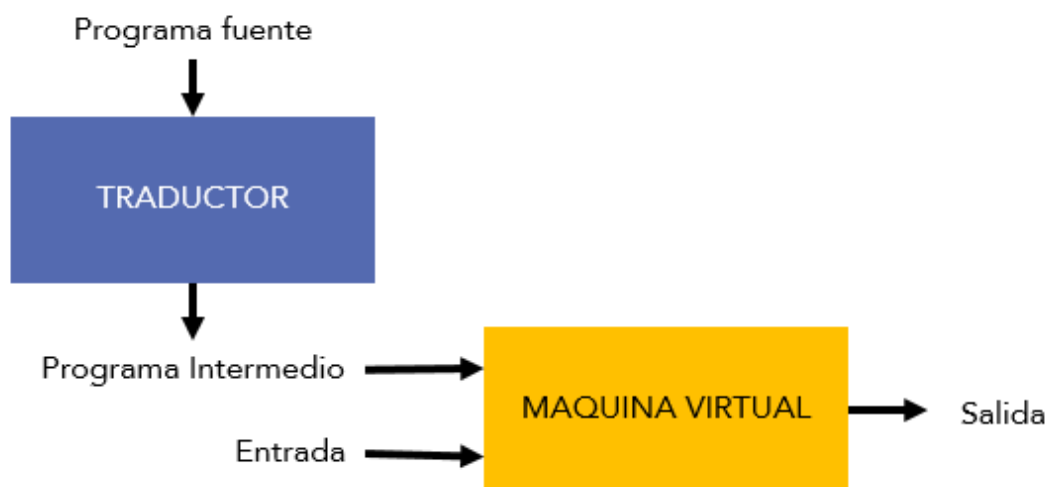


– Lenguajes interpretados:

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, Basic, etc.

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.

La sentencia principal es la **asignación**. 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.

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 argumentos que se aplican.

Entre sus características, destacan que no existe la operación de asignación las variables almacenan definiciones a expresiones, la operación fundamental es la aplicación de una **función a una serie de argumentos** y, además, la computación se realiza evaluando expresiones.

Algunos ejemplos de este tipo de lenguaje son: 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 que podremos consultar. La ejecución será en forma de consultas hacia esa base de datos.

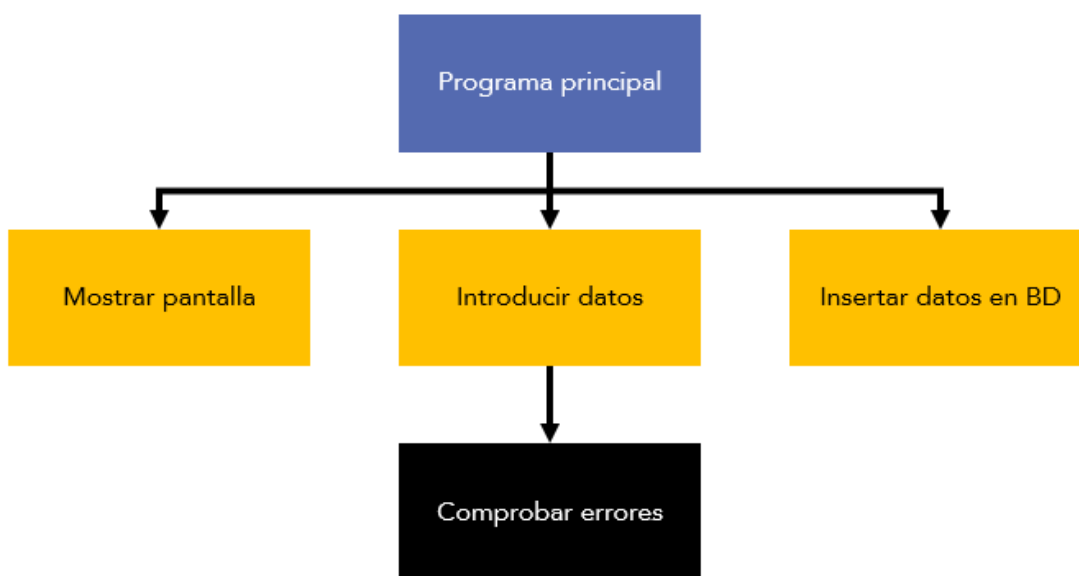
El lenguaje lógico más importante es *Prolog*, especialmente preparado para sistemas expertos, demostración de teoremas, consultas de bases de datos relacionales y procesamiento de lenguaje natural.

– Lenguajes estructurados:

Utilizan las tres construcciones lógicas nombradas anteriormente y resulta fácil 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.

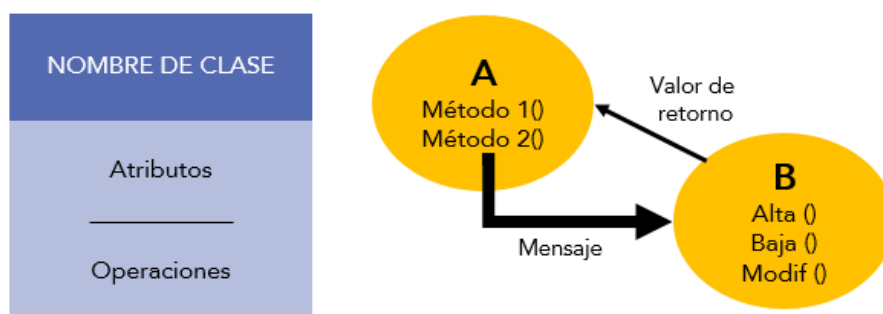
Algunos ejemplos de este lenguaje son: Pascal, C, Fortran, Modula-2, etc.

– **Lenguajes orientados a objetos:**

Este lenguaje estará definido 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. Por el contrario, su principal desventaja es la dificultad para programar en este lenguaje, ya que es muy poco intuitivo.

Algunos ejemplos de lenguajes orientados a objetos son: C++, Java, Ada, Smalltalk, etc.

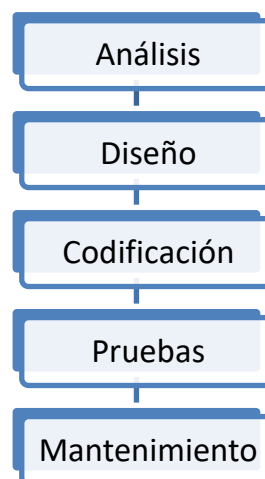
1.5. Fases del desarrollo de una aplicación: análisis, diseño, codificación, pruebas, documentación, mantenimiento y explotación

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

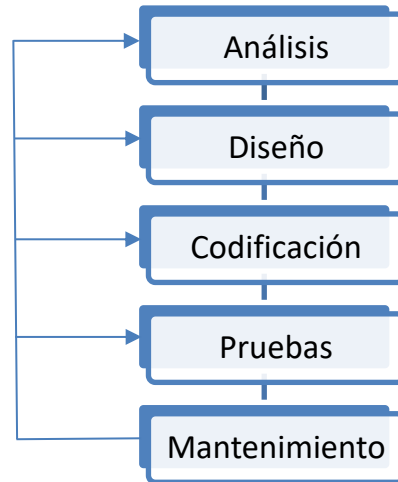
Modelos de desarrollo

Modelo en cascada

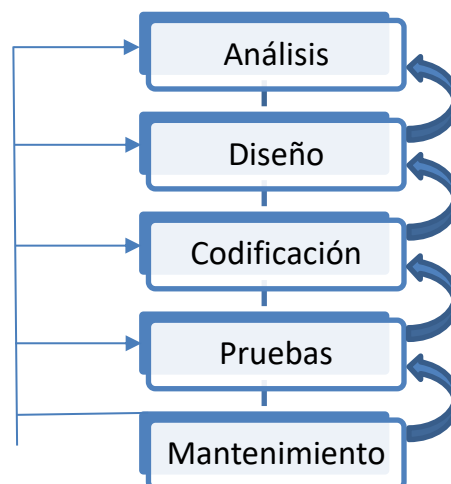
En este modelo las etapas para el desarrollo de software tienen un orden; de tal forma que, para empezar una etapa, es necesario finalizar la etapa anterior. Después de cada etapa se realiza una revisión para comprobar si se puede pasar a la siguiente.



Este modelo permite hacer iteraciones. Por ejemplo, durante la etapa de mantenimiento del producto, el cliente requiere una mejora, 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. Por ejemplo, supongamos que en cualquiera de las etapas se detectan fallos (los requisitos han cambiado, han evolucionado, ambigüedades en la definición de estos, etc.), 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.

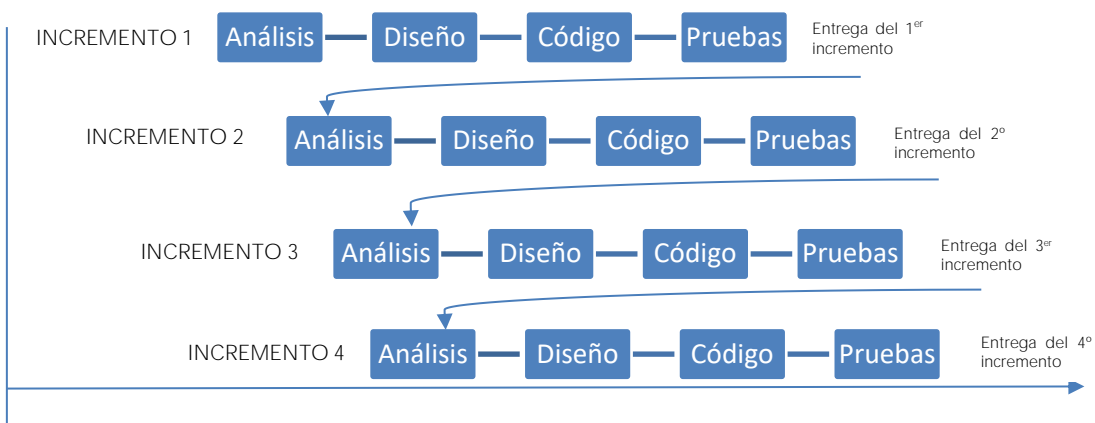


Ventajas	Inconvenientes
Fácil de comprender, planificar y seguir.	La necesidad de tener todos los requisitos definidos desde el principio.
La calidad del producto resultante es alta.	Es difícil volver atrás si se cometen errores en una etapa.
Permite trabajar con personal poco cualificado.	El producto no está disponible para su uso hasta que no está completamente terminado.
Se recomienda cuando	
El proyecto es similar a alguno que ya se haya realizado con éxito anteriormente.	
Los requisitos son estables y están bien comprendidos.	
Los clientes no necesitan versiones intermedias.	

Modelo iterativo incremental

El modelo incremental está basado en varios ciclos cascada realimentados aplicados repetidamente. Este modelo entrega el software en partes pequeñas, pero utilizables, llamadas *incrementos*. 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, se observa de forma iterativa el modelo en cascada para la obtención de un nuevo incremento mientras progresa el tiempo en el calendario.



Un procesador de textos se puede considerar como ejemplo de software desarrollado bajo este modelo. En el primer incremento se desarrollan funciones básicas de gestión de archivos y de producción de documentos; en el segundo incremento se desarrollan funciones avanzadas de paginación, y así sucesivamente.

Ventajas	Inconvenientes
No necesitan conocer todos los requisitos.	Es difícil estimar el esfuerzo y el coste final necesario.
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.
Se recomienda cuando	
Los requisitos o el diseño no están completamente definidos y es posible que haya grandes cambios.	
Se están probando o introduciendo nuevas tecnologías.	

Modelo en espiral

Este modelo combina el modelo cascada con el modelo iterativo de construcción de prototipos. El proceso de desarrollo del software se representa como una espiral, donde cada ciclo se desarrolla una parte de este. Cada ciclo está formado por cuatro fases y, cuando 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 ser 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:

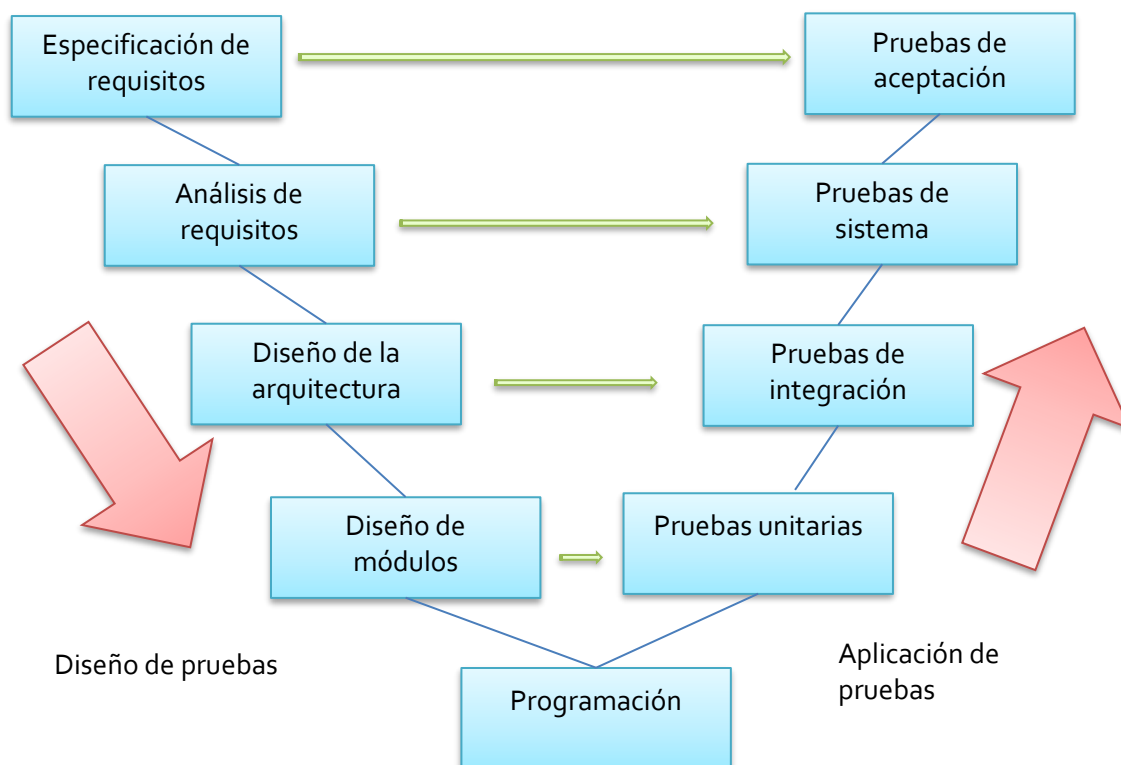
1. **Determinar objetivos:** cada ciclo de la espiral comienza con la identificación de los objetivos, las alternativas para alcanzar los objetivos y las restricciones impuestas a la aplicación de las alternativas.
2. **Análisis del riesgo:** a continuación, hay que evaluar las alternativas en relación 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, etc.).
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.
4. **Planificación:** revisar y evaluar todo lo que se ha hecho y, con ello, decidir si se continúa; 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.
Incorpora objetivos de calidad.	
Se recomienda cuando	
Proyectos de gran tamaño y que necesitan constantes cambios.	
Proyectos donde sea importante el factor riesgo.	

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 de cascada, ya que es muy rígido y contiene una gran cantidad de iteraciones.

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



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.

Aunque pueda parecerlo, no es una tarea fácil porque 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 con la que hablamos con el cliente.
- **Desarrollo conjunto de aplicaciones (JAD).** Entrevista de dinámica de grupo en la que cada uno tiene un rol (usuarios, administradores, desarrolladores, analistas, etc.).
- **Planificación conjunta de requisitos (JRP).** Subconjunto de JAD y se usa para productos de alto nivel.
- **Brainstorming.** Reuniones en las que se intentan crear ideas desde distintos puntos de vista. Idónea para el comienzo del proyecto.

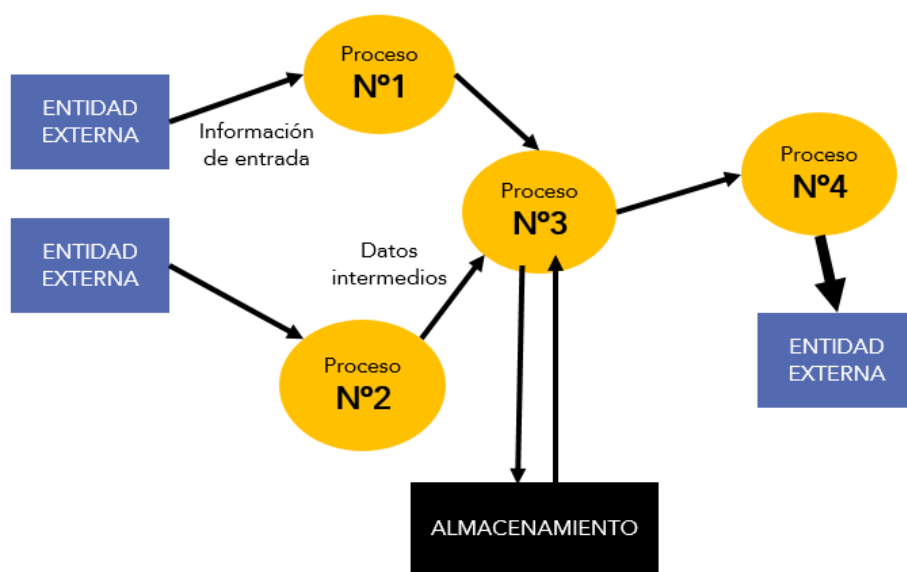
- **Prototipos:** Versión inicial del sistema en el que se puede ver el problema y sus posibles soluciones. Se puede desechar o usar para añadir más cosas.
- **Casos de uso.** Técnica definida por UML (del inglés *Unified Modeling Language*) y se basa en la representación de lo que queremos que haga el sistema. Describe qué hace el sistema, pero no cómo lo hace.

Podemos clasificar los requisitos en:

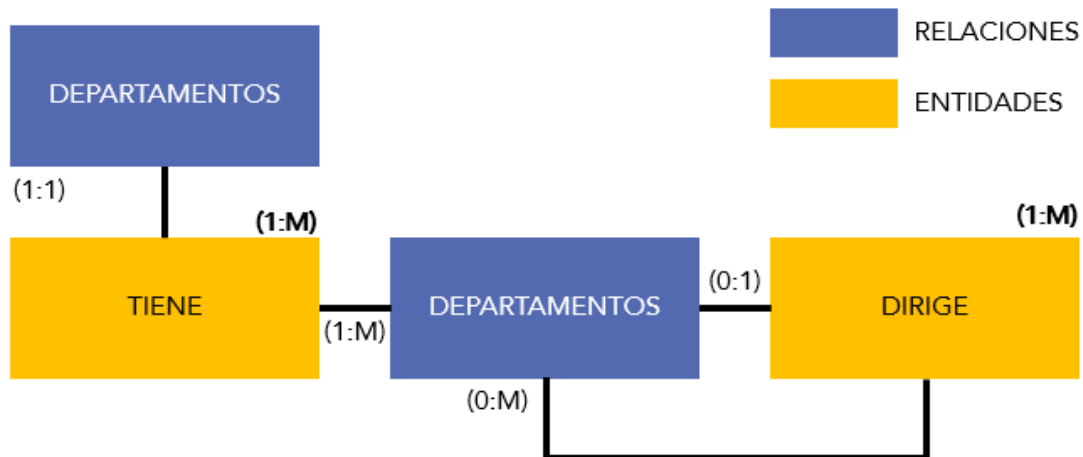
- **Requisitos funcionales.** Nos describen al detalle la función que realiza el sistema, la reacción ante determinadas entradas y cómo se comporta en distintas situaciones.
- **Requisitos no funcionales.** Los requerimientos que no se refieren directamente a las funciones específicas que entrega el sistema, sino a las propiedades emergentes del mismo, como la fiabilidad o la capacidad de almacenamiento.

A la hora de representar estos requisitos podemos hacerlo de distintas formas:

- **Diagramas de flujo de datos (DFD).** Nos va a representar el flujo de datos entre procesos (identifican funciones), 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.
- **Diagramas de transición de estados (DTE).** Representación de cómo se comporta el sistema ante acciones externas.
- **Diagrama Entidad/Relación (DER).** Se usa para representar datos y sus relaciones.



- **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. La estructura del documento ERS es la siguiente:

UC- NUM	Login	
Versión	1.0 (dd/mm/yyyy)	
Autores		
Dependencias	<ul style="list-style-type: none"> • [OBJ-0001] Objetivo Principal • [OBJ-0009] Gestión de usuarios 	
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 <i>pide nickname y password</i>
	2	El actor Invitado (ACT-0001) <i>introduce los datos y selecciona "entrar"</i>
	3	El sistema <i>comprueba en la base de datos que los datos son válidos y el caso de uso finaliza</i>
Postcondición	El usuario está dentro de la aplicación	
Excepciones	Paso	Acción
	3	Si <i>los datos no son válidos</i> , el sistema <i>no permite el acceso</i> , a continuación este caso de uso <i>queda sin efecto</i>

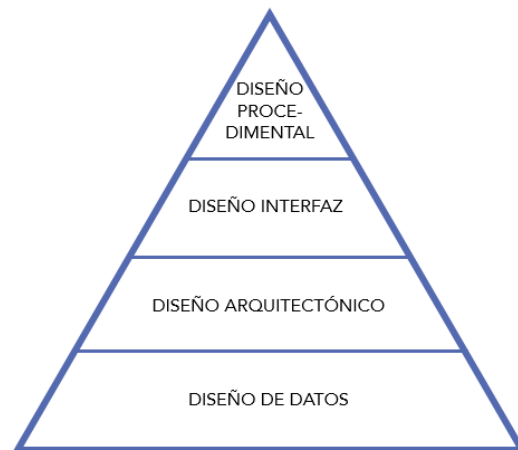
1.5.2. Diseño

Una vez 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.

Hay dos tipos de diseño:

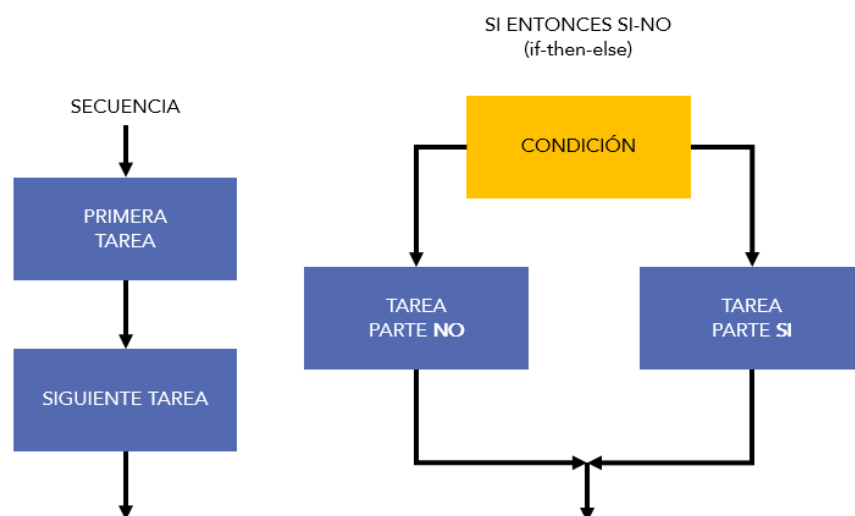
- **Diseño estructurado.** Basado en el flujo de datos a través del sistema. Produce un modelo de diseño con cuatro elementos:
 - **Diseño de datos.** Transforma la información relativa al mundo real en estructuras informáticas para su posterior implementación mediante diferentes lenguajes de programación.
 - **Diseño arquitectónico.** Se centra en la representación de la estructura de los componentes del software, sus propiedades y sus interacciones. Partiendo de los DFD, establecemos la estructura modular del software que se desarrolla.
 - **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.

- **Diseño a nivel de componentes (diseño procedimental):** Convierte elementos estructurales de la arquitectura del software en una descripción procedimental 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, etc.

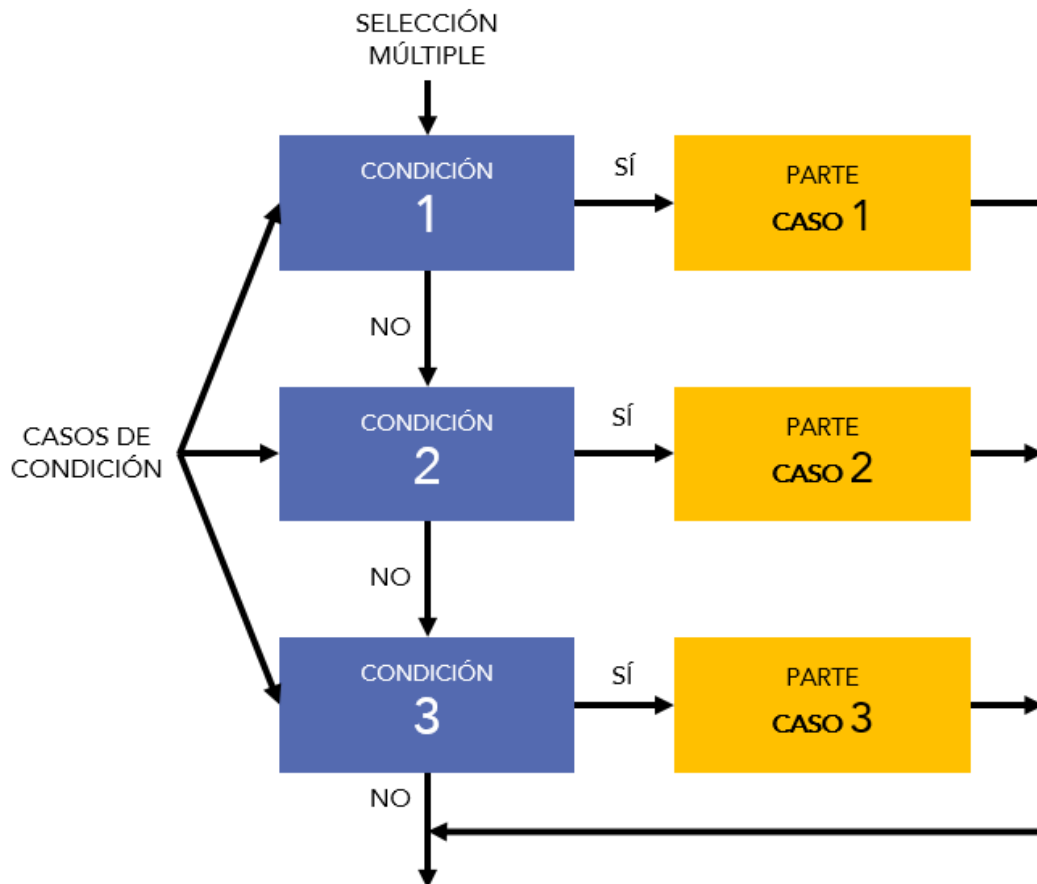


Para la programación estructurada existen una serie de construcciones que fueron fundamentos para el diseño a nivel de componentes. Estas construcciones son: secuencial, condicional y repetitiva.

1. **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.

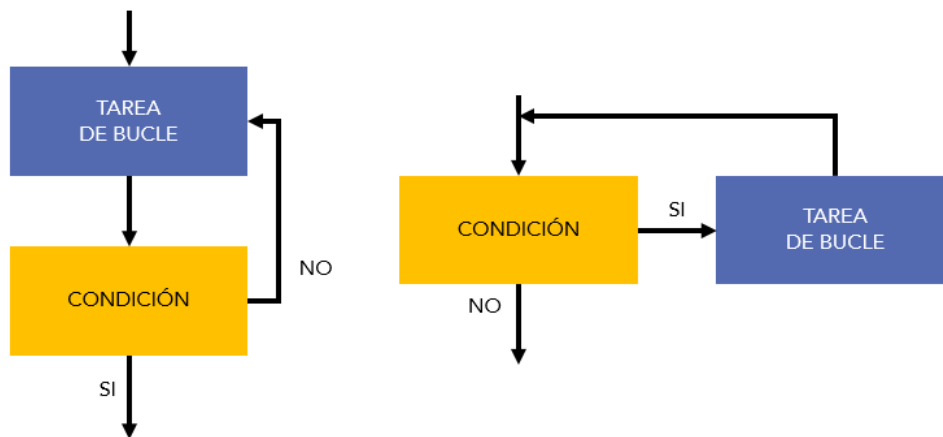


2. **Construcción condicional:** selecciona un proceso u otro según una condición lógica (rombo). Si se cumple se realiza la Parte Si, si no, se realiza la Parte No. La selección múltiple es una extensión de la estructura "Si entonces si-no"; un parámetro se prueba por continuas decisiones hasta que alguna es verdadera y ejecuta según ese camino.



3. **Construcción repetitiva:** es la que proporciona bucles. **Repetir-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.

Hacer-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.



- **Diseño orientado a objetos:** conjunto de objetos con propiedades y comportamientos, además de eventos que activan operaciones que modifican el estado de los objetos.

1.5.2.1. Notaciones gráficas para el diseño

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 procedimental.

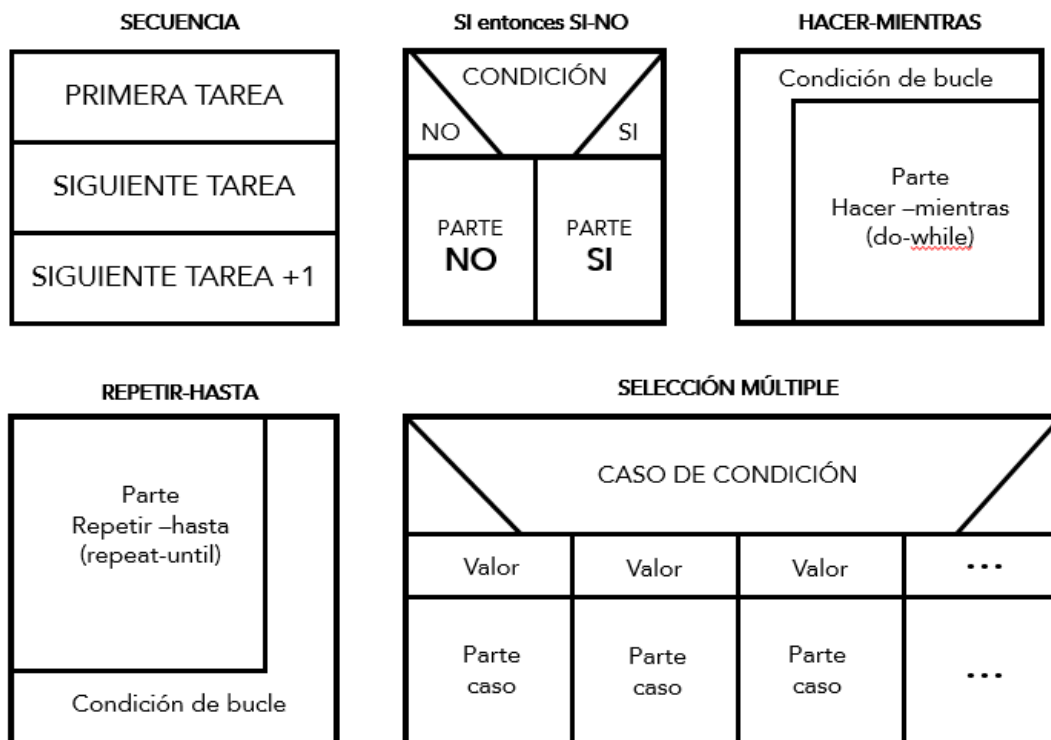
- **Caja:** paso del proceso.
- **Rombo:** condición lógica.
- **Flechas:** flujo de control.



- **Diagrama de cajas**

Surgió para la representación del diseño procedimental sin que violara las construcciones estructuradas.

- **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 (*do-while*) o inferior (*repeat-until*).
- **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 a ejecutar.



• Tablas de decisión

Nos permiten representar en una tabla las condiciones y las acciones que se llevarán a cabo al combinar esas condiciones. Se dividirá en cuatro cuadrantes:

- **Superior izquierdo:** lista de condiciones.
- **Inferior izquierdo:** lista de acciones posibles según la combinación de condiciones.
- **Superior derecho:** entrada de las condiciones.
- **Inferior derecho:** entrada de las acciones.

REGLAS

CONDICIONES	1	2	3	4			n
Condición N°1	<input checked="" type="checkbox"/>			<input checked="" type="checkbox"/>			
Condición N°2		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>				
Condición N°3	<input checked="" type="checkbox"/>			<input checked="" type="checkbox"/>			
ACCIONES							
Acción N°1	<input checked="" type="checkbox"/>			<input checked="" type="checkbox"/>			
Acción N°2	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>					
Acción N°3		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>			
Acción N°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	<p>Si <condición> Entonces</p> <p style="padding-left: 40px;"><Instrucciones></p> <p>Si no</p> <p style="padding-left: 40px;"><Instrucciones></p> <p>Fin si</p>
Condicional múltiple	<p>Según sea <variable> Hacer</p> <p style="padding-left: 40px;">Caso valor 1:</p> <p style="padding-left: 80px;"><instrucciones></p> <p style="padding-left: 40px;">Caso valor 2:</p> <p style="padding-left: 80px;"><instrucciones></p> <p style="padding-left: 40px;">Caso valor 3:</p> <p style="padding-left: 80px;"><instrucciones></p> <p style="padding-left: 40px;">Otro caso:</p> <p style="padding-left: 80px;"><instrucciones></p> <p>Fin según</p>
Repetir-hasta	<p>Repetir</p> <p style="padding-left: 40px;"><instrucciones></p> <p>Hasta que <condición></p>
Hacer-mientras	<p>Mientras <condición> Hacer</p> <p style="padding-left: 40px;"><Instrucciones></p> <p>Fin mientras</p>

A continuación, vemos un ejemplo de pseudocódigo:

Inicio

Abrir Archivo

Leer Datos del Archivo

Mientras no sea Fin de Archivo **Hacer**

Procesar Datos leído

Leer Registro del Archivo

Fin mientras

Cerrar Archivo

Fin

1.5.2.2. Diseño orientado a objetos

El diseño de software orientado a objetos (DOO) es muy complicado. Por ello, antes abordaremos un análisis orientado a objetos (AOO). En este análisis tendremos que definir las clases, las operaciones y los atributos asociados, así como las relaciones, comportamientos y comunicaciones entre clases.

Se definen 4 capas de diseño:

- 1) **Subsistema:** funciones y procedimientos que realizan las acciones del sistema principal.
- 2) **Clases y objetos:** en esta capa se indica la arquitectura de objetos global y la jerarquía de clases que hacen falta para implementar un sistema.
- 3) **Mensajes:** indica cómo se realiza la colaboración entre objetos.
- 4) **Responsabilidades:** operaciones y atributos que identifican a cada clase.

En el DOO y AOO utilizamos un **UML** (*Lenguaje de Modelado Unificado*). Es un lenguaje que está basado en diagramas para expresar modelos (representación donde se ignoran detalles pequeños).

1.5.3. Codificación

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

- **Indentación**

- Se usarán 4 espacios como unidad de indentación.
- Longitud de líneas de código no superior a 80 caracteres.
- Longitud de líneas de comentarios no superior a 70 caracteres.
- Si una expresión no cabe en una sola línea se deberá romper antes de una coma o un operador y se alineará al principio de la anterior.

- **Comentarios**

Contendrá 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 3 tipos:

- De bloque:

```
/*
    * Esto es un comentario de bloque
    */
```
- De línea:

```
/* Comentario de línea */
```
- 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 "(".
 - 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 de 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 está vacío no irá entre llaves.
- Sentencias *return* no irán entre paréntesis.

- **Separaciones**

Hacen más legible el código. Se utilizarán:

- **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, los operadores binarios menos el punto, expresiones del *for*, y entre un *cast* y la variable.

- **Nombres**

Los nombres de las variables, métodos, clases, etc. 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. Ej.: java.io
- **Clases e interfaces**: deben ser sustantivos o descriptivos, según lo que estemos creando. Si está compuesta por varias palabras, la primera letra de cada palabra irá en mayúscula.
- **Métodos**: se usarán verbos en infinitivo. Si está formado por varias palabras, el verbo estará en minúscula y la siguiente palabra empezará con mayúscula.
- **Variables**: deben ser cortas (incluso una letra) y significativas. Si está formada por varias palabras, la primera debe ir en minúscula.
- **Constantes**: el nombre debe ser descriptivo. Será totalmente escrita en mayúscula y si son varias palabras, separadas por un carácter de subrayado.

Una vez 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 enlacemos 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.

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:

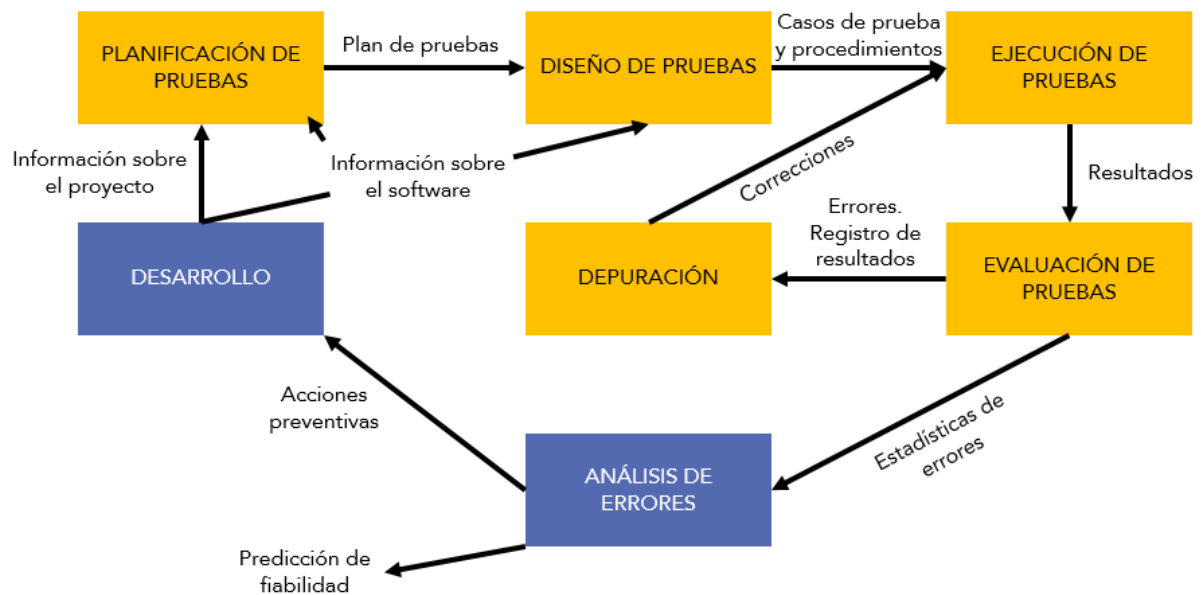
- **Verificación:** conjunto de actividades que permiten comprobar si el producto se está construyendo correctamente.
- **Validación:** acciones para comprobar si el producto es correcto y si se ajusta a los requisitos del cliente.

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

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

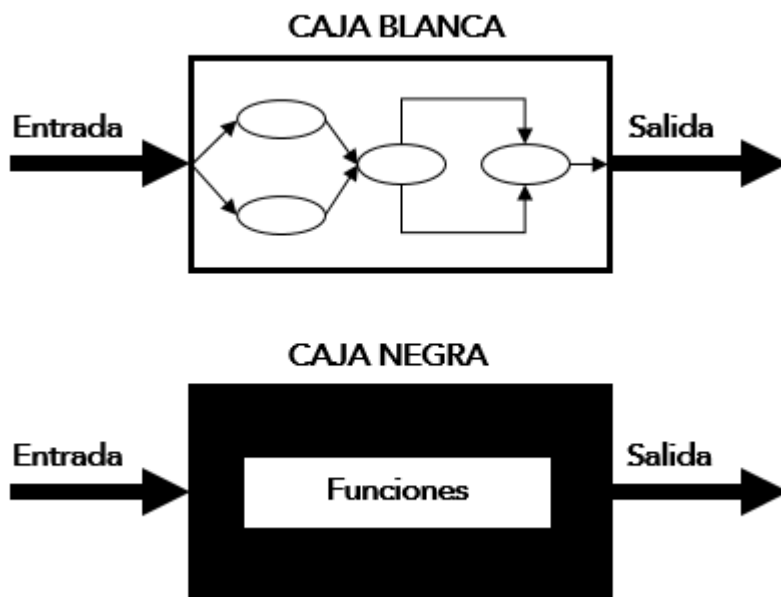
- Cada prueba definirá los resultados de la salida esperados.
- Evitar que el programador pruebe sus propios programas.
- Comprobación de cada resultado en profundidad.
- Incluir todo tipos de datos, tanto válidos y esperados como inválidos e inesperados.
- Comprobar que el software hace lo que debe 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.
- A más pruebas que se realicen, mayor es la probabilidad de encontrar errores y, una vez solucionados, tendremos una probabilidad 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 a probar.
- 2) Después se **diseñan las pruebas**. Técnicas que utilizar.
- 3) **Generación de los casos de prueba.**
- 4) **Definición de los procedimientos de prueba.**
- 5) **Ejecución de las pruebas.**
- 6) **Evaluación.** Identificación de posibles errores.
- 7) **Depuración.** Localizar y corregir errores. Testar de nuevo tras encontrarse un error para verificar que se ha corregido.
- 8) **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.



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 del 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, aunque también puede ser útil durante el desarrollo.

- **DOCUMENTACIÓN DEL USUARIO**

Distinguiremos entre usuarios finales y usuarios administradores del sistema:

Los **usuarios finales** solo se centran en cómo el programa realiza las tareas, no les interesa los detalles informáticos.

Los **administradores del sistema** gestionarán los programas que usan los usuarios finales. Puede funcionar como gestor de red o como técnico resolviendo los problemas de 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:



Documento de instalación del sistema	Destinado a administradores	Describe cómo instalar el sistema. Contiene información de fichero del sistema, su configuración, hardware mínimo o cómo iniciar el sistema
Manual introductorio	Destinado a usuarios noveles	Explicaciones sencillas de cómo empezar a usar el sistema. Solución de errores
Manual de referencia del sistema	Destinado a usuarios experimentados	Descripción detallada del sistema y lista completa de errores
Guía del administrador del sistema	Destinado a administradores	Para sistemas que hay comandos, describir las tareas, los mensajes producidos y las respuestas del operador
Tarjeta de referencia rápida	-	Sirve para realizar búsquedas rápidas

• 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:

Fundamentos del sistema	Se describen los objetivos del sistema
El análisis y especificación de requisitos	Información exacta de los requisitos
Diseño	Se describe la arquitectura del sistema
Implementación	Descripción de la forma que se expresa el sistema y acciones del programa en forma de comentarios

Plan de pruebas del sistema	Evaluación individual de las unidades del sistema y las pruebas que se realizan
Plan de pruebas de aceptación	Descripción de pruebas que el sistema debe pasar antes de que los usuarios las acepten
Los diccionarios de datos	Descripciones de los términos que se relacionan con el software en cuestión

Cuando los sistemas son más pequeños, la documentación también suele ser menos amplia. Aun así, deberá incluir, como mínimo, 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 consultarlo, 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 de este.
- Debe poseer un índice con capítulos, secciones y subsecciones.
- Incluir al final un glosario de términos.

El estándar IEEE (Institute of Electrical and Electronics Engineers) propone la siguiente estructura:

Componente	Descripción
Datos de identificación	Título e identificación
Tabla de contenidos	Capítulo, nombre de sección y número de página
Lista de ilustraciones	Números de figura y títulos
Introducción	Propósito del documento y breve resumen
Información para el uso de la documentación	Sugerencia sobre cómo usar la documentación de forma eficaz

Conceptos de las operaciones	Explicación del uso del software
Procedimientos	Instrucciones sobre cómo utilizar el software
Información sobre los comandos software	Descripción de cada uno de los comandos del software
Mensajes de error y resolución de problemas	Descripción de los mensajes de error y los tipos de resolución
Glosario	Definiciones de términos especiales
Fuentes de información relacionadas	Enlaces a otros documentos para proporcionar información adicional
Características de navegación	Permite encontrar su ubicación actual y moverse por el documento
Índice	Lista de términos clave y páginas que estos referencian
Capacidad de búsqueda	Forma de buscar términos específicos

1.5.6. Explotación

En esta etapa se lleva a cabo la instalación y puesta en marcha de producto y se realizarán las siguientes tareas:

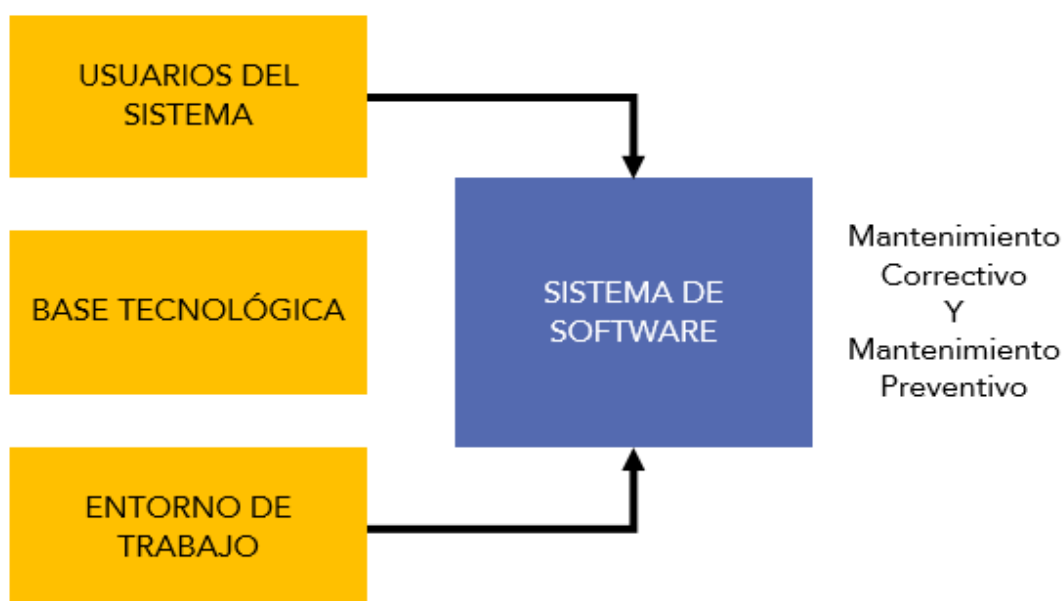
- **Estrategia para la implementación del proceso.** Se definen procedimientos para recibir, registrar, solucionar, hacer un seguimiento de los problemas y probar el producto.
- **Pruebas de operación.** Para cada lanzamiento del producto se llevará a cabo una prueba y, si los criterios establecidos son satisfactorios, se procederá a liberar el software.
- **Uso operacional del sistema.** El sistema entrará en acción en el entorno previsto.
- **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 rendimiento o adaptar el producto a un entorno modificado.

Existen cuatro tipos de mantenimiento que dependen de las demandas del usuario:

- **Mantenimiento adaptativo.** Estará centrado en la modificación del producto según los cambios que se puedan producir tanto a nivel de software como de hardware. Es el mantenimiento más común debido a los rápidos cambios que se producen en la informática.
- **Mantenimiento correctivo.** Se centrará en corregir los errores que puedan producirse una vez entregado el producto.
- **Mantenimiento perfectivo.** Perfeccionamiento de la aplicación tras el uso y descubrimiento de nuevas mejoras que pueden ser incluidas.
- **Mantenimiento preventivo.** Modificación del producto sin alterar las especificaciones de este. Sobre todo, está relacionado con la mejora de la compresión y la usabilidad del programa.

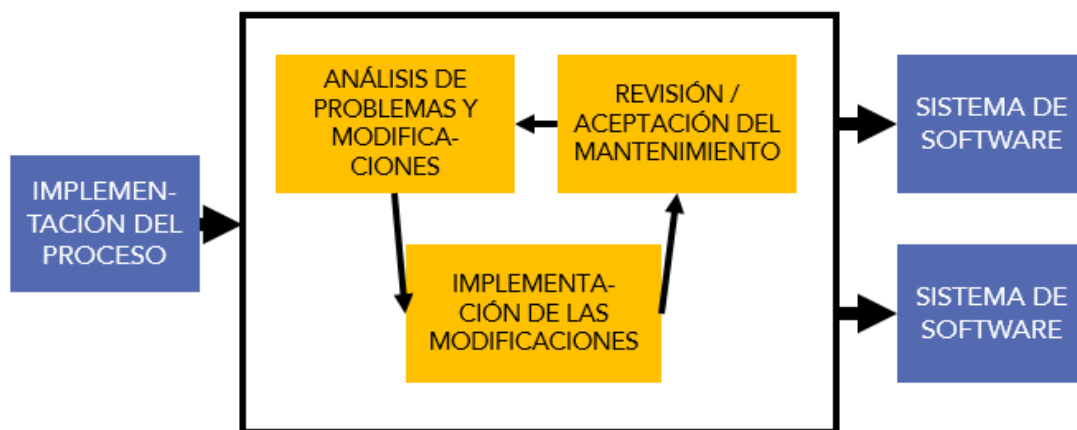


Las tareas que se realizarán en esta etapa son:

- **Implementación del proceso.** La persona encargada de la preparación, documentación y ejecución de los diferentes procedimientos va a ser el responsable del mantenimiento para que, de esta forma, pueda realizar las

diferentes actividades que estén relacionadas con dicho proceso de mantenimiento.

- **Análisis de problema y modificaciones.** Deberá analizar el informe de del problema, verificarlo, desarrollar opciones para la implementación de la modificación y documentar la solicitud de modificación, así como los resultados y las opciones de implementación.
- **Implementación de las modificaciones.** Tendrá que analizar y determinar qué documentación, unidades de software y versiones se deben modificar. También es preciso documentarlo y se deberá ejecutar el proceso de desarrollo para que se implementen las modificaciones.
- **Revisión/aceptación del mantenimiento.** Deberá de llevar a cabo revisiones para determinar la integridad del sistema modificado.
- **Migración.** Preparar, documentar y ejecutar un plan de migración o transferencia.
- **Retirada del software.** El producto se retirará por petición del propietario. Se deberá preparar y documentar un plan de retirada del soporte activo, así como notificar los planes y actividades de la retirada. También se deberá facilitar la transición al nuevo sistema llevando a cabo paralelamente la retirada y la implementación del nuevo software. En este período, se deberá formar a los usuarios.



1.6. Metodologías ágiles

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).
- Mejora la satisfacción del cliente.
- Mejora la motivación e implicación del equipo de desarrollo.
- Mejora la calidad del producto.
- Eliminar aquellas características innecesarias del producto.
- Alerta rápidamente tanto de errores como de problemas.

En este punto vamos a destacar tres metodologías:

- **Scrum:**



Aportar 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 auto organizados, antes que en la calidad de los procesos empleados.

Solapamiento de las diferentes fases de desarrollo.

Seguir los pasos del desarrollo ágil: desde el concepto o cisión general de la necesidad del cliente, construcción del producto de forma incremental a través de iteraciones. Estas iteraciones (en Scrum se llaman Sprint) se repiten de forma continua hasta que el cliente da por cerrada la evolución del producto.

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 que se va

liberando el producto por pares, periódicamente, iterativamente, poco a poco y, además, cada entrega es el incremento de funcionalidad respecto a la anterior. Cada periodo de entrega → Sprint.

- Reunión diaria. Máximo 15 minutos. Se trata de 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 qué se ha completado y de qué no.
- Retrospectiva del Sprint. También al final de Sprint, y sirve para que los implicados den sus impresiones sobre Sprint; se utiliza para la mejora del proceso.

- **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.

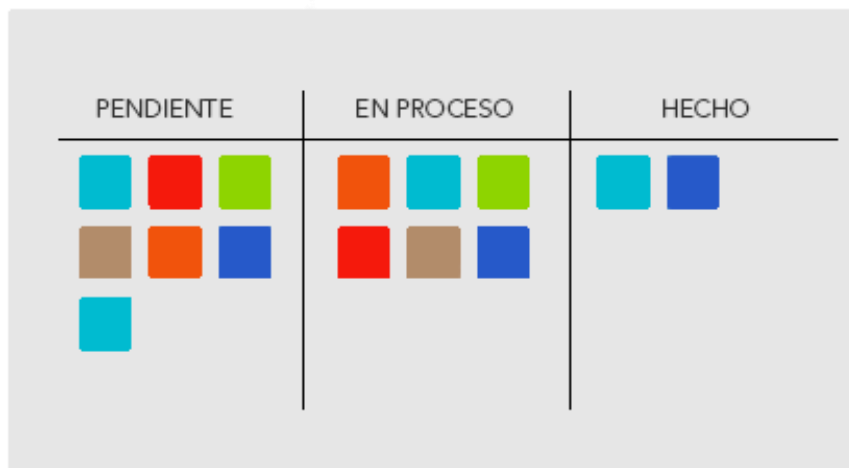
XP se basa en retroalimentación continua entre cliente y el equipo de desarrollo. XP es especialmente adecuada para proyectos con requisitos imprecisos y muy cambiantes.

Características específicas de XP:

- 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 software.

- Desarrollar un software que funcione, más que conseguir una buena documentación.
- La colaboración con el cliente. Se propone que exista una interacción constante entre el cliente y el equipo de desarrollo.
- Responder a los cambios. 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:**



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:

- 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 cada uno puede tardar a realizarlas.

1.7. Proceso de obtención de código a partir del código fuente; herramientas implicadas

Durante la etapa de diseño, se construyen los componentes software con el suficiente nivel de detalle, de tal forma que sirvan como guía en la generación de código fuente en un lenguaje de programación; algunas herramientas utilizadas para ello son los diagramas de flujo o el pseudocódigo.

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

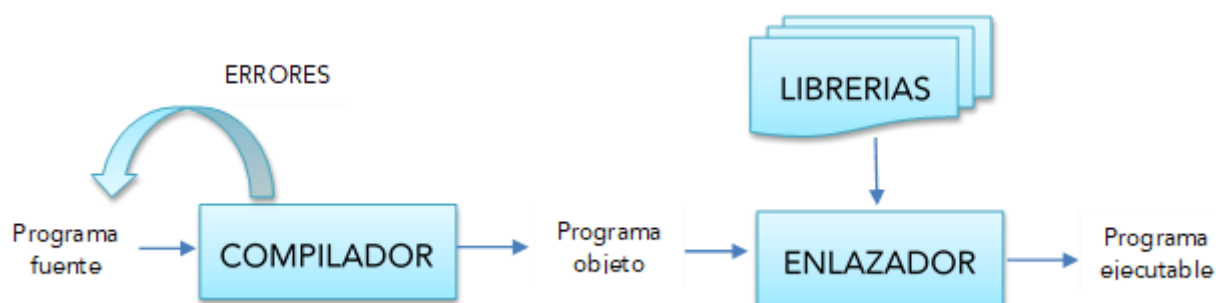
1.7.1. Tipos de código:

- **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.

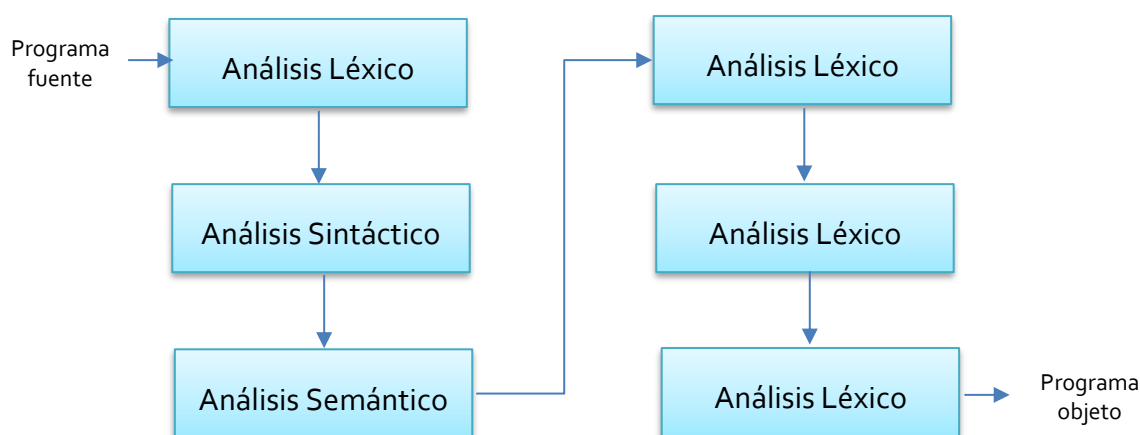
1.7.2. Compilación:

La compilación de un programa se lleva a cabo mediante dos programas: el compilador y el enlazador. Si el compilador en el proceso de traducción devuelve algún error, no se generará el programa objeto; será necesario modificar el programa fuente y pasarlo de nuevo por el compilador.

El compilador se compone internamente de varias etapas o fases que realizan distintas operaciones:



- **Análisis léxico:** se lee secuencialmente todo el código fuente, que se obtiene de unidades significativas de caracteres denominadas *tokens*. Por ejemplo, la instrucción: suma = x+ 2, generan 5 tokens: suma, =, x, +, 2.
- **Análisis sintáctico:** recibe el código fuente en forma de *tokens* y realiza el análisis sintáctico que determina la estructura del programa; es decir, se comprueba si las construcciones de *tokens* cumplen las reglas de sintaxis definidas en el lenguaje de programación correspondiente. El proceso es semejante al análisis gramatical sobre alguna frase en lenguaje natural.
- **Análisis semántico:** se comprueba que las declaraciones son correctas. Se verifican los tipos de todas las expresiones y si las operaciones se pueden realizar sobre esos tipos, si los arrays tienen el tamaño y tipo adecuados, etc.
- **Generador de código intermedio:** después del análisis se genera una representación intermedia similar al código máquina con el fin de facilitar la tarea de traducir al código objeto.
- **Optimización de código:** trata de mejorar el código intermedio generado en la fase anterior, de tal forma que el código resultante sea más fácil y rápido de interpretar por la máquina.
- **Generación de código:** genera el código objeto de nuestro programa.



El programa enlazador inserta en el código objeto las funciones de librería necesarias para producir el programa 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 en los que conoceremos sus características, evaluaremos los entornos integrados de desarrollo, instalación y configuración de estos entornos y abordaremos la creación de modelos de datos y desarrollo de programas.

2.1. Funciones de un entorno de desarrollo.

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.

En cada fase del desarrollo intervienen varias herramientas hasta llegar al resultado final. Podremos tener varias IDEs para todas las etapas.

• COMPONENTES DE UN ENTORNO DE DESARROLLO

- **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 en el que la máquina sea capaz de interpretarlo y ejecutarlo.
- **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 (del acrónimo en inglés *What You See Is What You Get*) de arrastrar y soltar.
- **Control de versiones.** Controla los cambios realizados sobre las aplicaciones, obteniendo así revisiones y versiones de las aplicaciones en un momento dado.

2.2. Instalación de un entorno de desarrollo.

Existen muchos tipos de entornos de desarrollo, pero en esta unidad hemos optado por la instalación de 2 tipos: uno orientado al trabajo con bases de datos y crear modelos de datos (SQLDEVELOPER con Data Modeler, herramienta de Oracle), y otro orientado al desarrollo de programas (Eclipse).

Ambas aplicaciones se pueden descargar gratis:

Oracle: <http://www.oracle.com/us/downloads/index.html>

Eclipse: <http://www.eclipse.org>

Será necesario la instalación de Java, recomendado la instalación de JDK 7 o superior.

Son herramientas con bastante éxito en el mercado y que seguirán siendo usadas una vez terminados los estudios.

2.2.1. Instalación de Eclipse

Tenemos que descargarnos el archivo comprimido desde la dirección apuntada anteriormente y descomprimirlo en nuestro disco duro. Para descargarlo, pulsaremos sobre la versión **Standard**. Iremos añadiendo plugin para aumentar funcionalidades.

Así, la versión a descargar será *Eclipse Kepler Standard*, disponible para 32 y 64 bits. Antes de comenzar la instalación comprobaremos qué versión tiene nuestro sistema operativo.

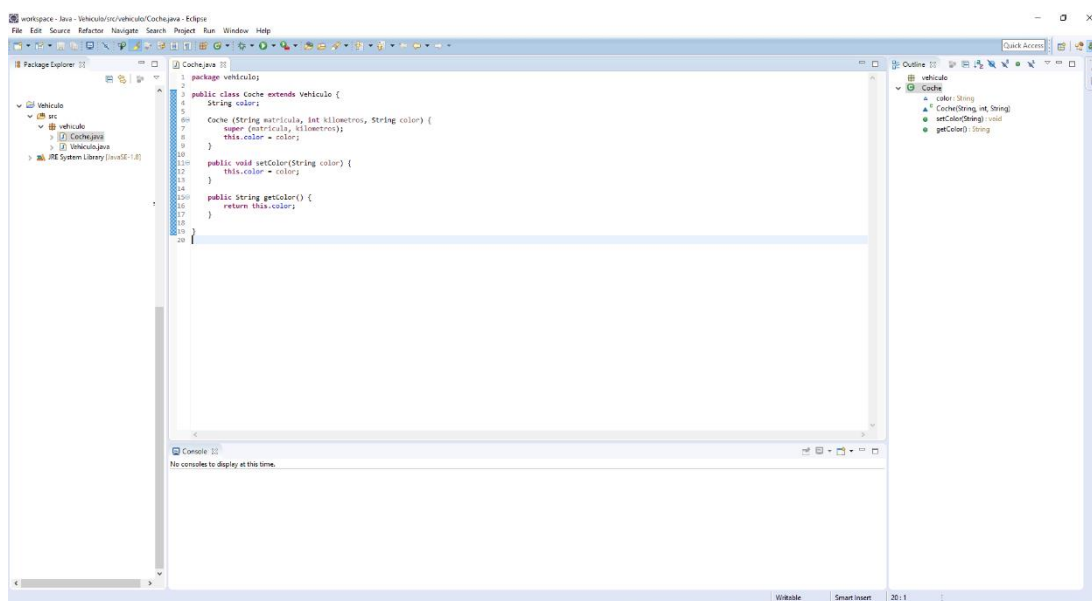
Una vez descomprimido, ejecutaremos el archivo eclipse.exe y nos aparecerá una ventana, la cual nos preguntará la ruta de nuestro espacio de trabajo (**workspace**). En Eclipse todos los proyectos se crean en espacios de trabajo, por lo que podremos crear una carpeta nueva en este momento o seleccionar una ya creada.

Antes de entrar aparecerán unas ventanas informativas para ver tutoriales, novedades o ejemplos. Seleccionamos **Workbench** para ir a la ventana principal.

Si queremos copiar un proyecto ya creado para usar las distintas herramientas podemos hacerlo abriendo el menú y accediendo a **File/Import**. Desplegamos la carpeta **General** y seleccionamos **Existing Projects into Workspace**. En la siguiente ventana seleccionamos nuestro proyecto y pulsamos en **Finish**.

Podemos distinguir los siguientes bloques en la ventana principal de Eclipse:

- **Package Explorer** o zona de proyectos. Desde esta área podremos navegar por todos los elementos del proyecto.
- **Zona de edición**. Espacio donde escribiremos el código del programa. El texto se resaltará para identificar la sintaxis y, si hay algún error, marcará la línea y ofrecerá soluciones. También mostrará plantillas para simplificar la escritura.
- **Outline**. Esquema de la clase que se edita para acceder a sus métodos.
- **Consola Java**. Muestra resultado de la ejecución, mensajes de salida por consola y errores de ejecución.



Podremos visualizar distintas perspectivas dependiendo del tipo de desarrollo que queramos realizar y de los plugin que tengamos instalados. También tenemos la opción de mostrar varias vistas. Para ello, iremos a menú, Windows/Show View. Si queremos ejecutar el proyecto pulsaremos en el botón *Run* de la barra de botones.

2.2.2. Instalación de Plugin

Un plugin es un complemento que se añade a un programa principal para agrega características o funcionalidades. Su presencia es habitual en navegadores web, reproductores de música, sistemas de gestión de contenidos o herramientas de desarrollo como Eclipse.

No son parches ni actualizaciones, sino que son propiedades que se añaden al programa principal.

2.2.2.1. Plugin WindowsBuilder

La forma más básica de Eclipse no posee una interfaz gráfica de usuario para crear aplicaciones, sino que tendremos que instalar un plugin que nos permita realizar esa función. Uno de los más populares es **WindowsBuilder**, desarrollado por Google, que nos permite realizar de forma rápida y sencilla las GUI de las aplicaciones Java.

Para realizar la instalación de este plugin debemos saber la versión de Eclipse que tenemos instalada. irlo Podemos consultarla en el menú **Help/About Eclipse**. En esta dirección podemos descargarnos el plugin:

<http://www.eclipse.org/windowsbuilder/download.php>.

Accederemos al link de descarga y se nos bajará un archivo comprimido. Para instalarlo iremos al menú **Help/Install New Software**. Nos aparece una ventana y pulsamos en **Add**, aparecerán los paquetes y pulsamos en **Next**.

Visualizaremos una ventana con todos los elementos y componentes que se nos instalarán y nos pedirá que aceptemos los términos. Aceptamos y comenzará la descarga e instalación. Después habrá que reiniciar el equipo.

- **CREACIÓN DE UN PROYECTO CON UNA INTERFAZ GRÁFICA**

También se podrá instalar escribiendo la siguiente URL:

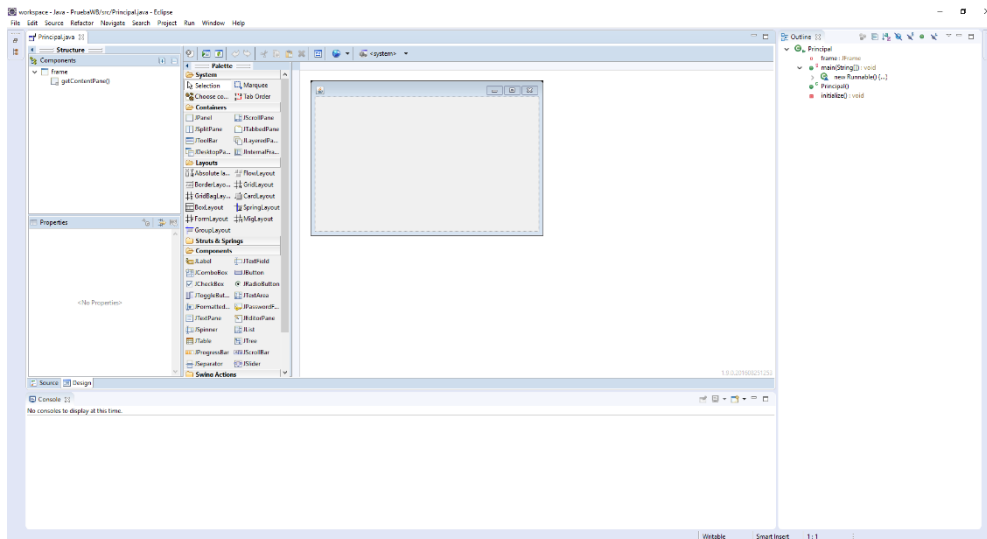
<http://download.eclipse.org/windowsbuilder/WB/release/R201406251200/4.3/>

En el campo *Work With* en el menú **Help/Install New Software**.

Si queremos crear un proyecto abrimos el menú **File/New/Java Project** y, en la ventana que nos aparece, dejamos las opciones que están por defecto y pulsamos **Finish**. En la siguiente ventana pondremos el nombre del proyecto y pulsando el botón derecho del ratón, en el menú contextual elegimos **New/Other**. Abrimos la carpeta **WindowsBuilder/Swing Designer** y escogemos **Application Window**. Seguidamente, escribimos el nombre de la ventana y pulsamos **Finish**. Así crearemos una ventana de aplicación que inicialmente estará vacía.

Se habrá creado una clase con el nombre, no se podrá editar en modo *Source* (fuente) o en modo *Desing* (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 arrástralo arrastramos al marco interno de la ventana, lo que nos va a permitir colocar los componentes en cualquier parte de la misma.

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

2.2.3. Instalación de SQL Developer

SQL Developer es una herramienta gráfica de Oracle que simplifica el desarrollo de las bases de datos, especialmente en ORACLE. Con esta herramienta podremos:

- Navegar, editar y crear objetos de base de datos Oracle.
- Ejecutar sentencias SQL.
- Editar y depurar PL SQL.
- Ejecutar informes.
- Colocar archivos bajo control de versiones.
- Crear modelos de datos y generar DLL correspondientes.
- Realizar modelos relacionales e ingeniería entre modelo lógico y relacional.

Instalaremos la versión **sqldeveloper3.2.10.09.57** y el proceso a seguir es similar a Eclipse, explicado anteriormente. Descomprimos el fichero descargado e iniciamos el ejecutable `sqldeveloper.exe`. Indicamos la ruta donde está instalado el JDK. Tendremos en cuenta la versión del sistema operativo (32 o 64 bits).

Esta herramienta está pensada para trabajar con Oracle, aunque puede usarse con otras bases de datos como MySQL. Para ello, solo tendríamos que instalar el conector que corresponda.

Para el uso de la herramienta Data Modeler o el control de versiones que incorpora SQL Developer no es necesario tener instalada la BD Oracle. Sin embargo, es recomendable tenerlo instalado, al igual que MySQL, para los modelos de datos que veremos posteriormente.


Al arrancar SQL Developer veremos en la parte izquierda un bloque donde configuraremos las conexiones a la BD y un bloque central donde tendremos ayudas y tutoriales.

• CREACIÓN DE CONEXIONES A BD DESDE LA HERRAMIENTA

Para crear una conexión pulsaremos en el botón *más (+)* en el panel de conexiones. Se visualizará un cuadro de diálogo para poner los datos de la conexión y tendremos que rellenar:

- **Nombre de la conexión**, es decir, el nombre que se verá en el panel. Después se creará un usuario y una contraseña.
- **Tipo de conexión**, dejaremos las opciones por defecto. También será importante saber el puerto y el SID (nombre de la instancia).

Una vez creada una conexión, si hacemos clic sobre ella veremos la ventana de trabajo SQL, donde podremos escribir y ejecutar sentencias SQL. También podremos ver los objetos asociados a una conexión, las tablas y, si hacemos clic sobre ella, el detalle de la misma.

Si nos aparece una ventana alertando de que la contraseña va a caducar al acceder con el usuario, deberemos escribir en la hoja de trabajo *alter user USUARIO identified by hr2*, donde *USUARIO* será el nombre del usuario de la conexión y *hr2* la nueva clave. Ejecutamos la sentencia con el icono  (Sentencia de ejecución). También debemos poner la contraseña en propiedades de la conexión.

• CREACIÓN DE UNA CONEXIÓN A UNA BD MySQL

Si queremos crear una BD distinta de Oracle tendremos que instalar el driver JDBC que corresponda. Para MySQL instalaremos **mysql-connector-java-5.1.18-bin.jar** situado en la carpeta recursos. Lo copiamos en otra carpeta y vamos a *Herramientas/Preferencias*. En la siguiente ventana accedemos a la sección *Bases de datos* y seleccionamos *Controladores JDBC de Terceros*. Ahora seleccionamos la ruta

del fichero .jar. En este punto, ya nos aparecerá una pestaña nueva en la ventana de creación de conexiones. Habrá que reiniciar el SQL Developer.

Para conectarnos a la base de datos pulsaremos en *nueva conexión*, escribimos el nombre de usuario, el servidor y el puerto de entrada. En la selección de la base de datos elegimos la que queramos usar y guardamos la conexión.

Una vez conectados se podrán realizar consultas y otro tipo de sentencias SQL.

2.3. Herramientas para el modelado de datos

En este apartado vamos a aprender cómo se crea un modelado de datos a través de las dos herramientas que hemos instalado anteriormente. Será un complemento a los módulos anteriores de Base de Datos en los que hemos visto el modelo E/R y el relacional.

2.3.1. Data Modeler de Oracle

Una herramienta de modelado de datos que podemos descargar de forma gratuita es Oracle Data Modeler desarrollada por Oracle, la cual podrá ser descargada por separado o incorporada al *SQL Developer*. Sus características más importantes son:

- Los modelos se almacenan bajo una estructura de directorios.
- Puede trabajar con cualquier tipo de BD, no será necesario que sea Oracle.
- Permite ingeniería inversa.
- Dispone de los siguientes niveles de diseño: lógico, relacional y físico.
- Define dominios de tipos de datos.
- Es una herramienta de diseño visual y con muchas funcionalidades.
- Múltiples opciones de generación del DDL.

En este apartado vamos a estudiar la versión *Data Modeler* que incorpora SQL Developer **versión *sqldeveloper-3.2.10.09.57***. Para abrir el explorador de Data Modeler pulsamos en el menú *Ver/Data Modeler/Explorador* y se mostrará el módulo del *Data Modeler*. Si queremos abrir una ventana del modelo lógico, desde el menú contextual de *Modelo Lógico* elegimos *Mostrar*. En cambio, si queremos crear un modelo relacional elegimos *Modelo Relacional* del menú contextual de *Modelos Relacionales*.

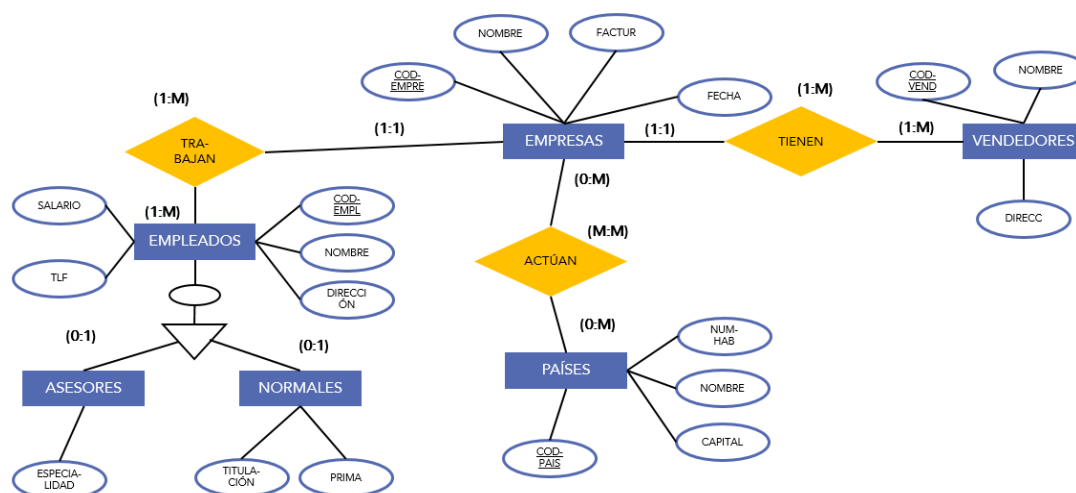
Antes de empezar a crear ningún modelo insertaremos las tablas de nuestro usuario. Para ello vamos a elegir desde el menú contextual de Modelos Relacionales la opción *Nuevo Modelo Relacional*. En la hoja de trabajo que se nos muestra arrastraremos todas las tablas.

En la barra de botones vamos a poder incluir, entre otras funciones, tablas, vistas y claves ajenas y podremos dividir también tablas. En la opción de *Realizar ingeniería al modelo lógico* se van a crear las diferentes entidades y relaciones de un determinado modelo relacional.

Cuando queramos guardar este diseño de datos pulsamos en el menú *Archivo/Data Modeler/Guardar* o *Guardar como*, seleccionamos la carpeta de almacenamiento y guardamos. Si lo que queremos es abrir un modelo, lo haremos desde el menú *Archivo/Data Modeler/Abrir*. Cuando se guarda el proyecto se crea un fichero con la extensión *.dmd* y una carpeta con el mismo nombre.

2.3.1.1. Creación de modelos de datos con Data Modeler

El próximo paso por seguir es la creación de un modelo lógico de datos con la herramienta instalada. Partiremos de diseños ya realizados. Suponemos que hemos realizado un modelo de datos según un enunciado y tenemos el siguiente esquema:



Para crear un nuevo diseño, dentro del explorador de *Data Modeler* abrimos el menú contextual del nodo *Diseños* y seleccionamos *Nuevo Diseño*. Usando la barra de botones añadimos las entidades y relaciones que tengan entre ellas y cuando lo tengamos creado se realiza la ingeniería inversa para crear el modelo relacional.

Para crear una nueva entidad pulsamos el botón *Nueva Entidad*. Desde esta ventana añadimos las propiedades que tendrá la entidad, pero prestaremos especial atención en las propiedades de los apartados general y atributos.

- **Propiedades de General.** Datos que tienen que ver con la entidad. Lo más importantes es poner el nombre y, si trabaja con jerarquías indicar el

supertipo y cómo se desea la ingeniería. El resto de opciones las dejaremos por defecto. Las más importantes son:

- **Nombre:** nombre de la entidad.
 - **Abreviatura:** nombre corto para la entidad.
 - **Sinónimos:** sinónimos de la entidad.
 - **Sinónimo para mostrar:** sinónimo a mostrar para la entidad.
 - **Abreviatura preferida:** nombre abreviado que utilizaremos para la correspondencia de las tablas.
 - **Nombre completo:** nombre que utilizaremos para visualizar el nombre de las entidades y atributos.
 - **Estrategia Ingeniería directa:** Es importante al trabajar con entidades supertipo y subtipo. Se define la estrategia de las entidades para hacer ingeniería a las tablas del modelo relacional:
 - **Tabla única.** Datos de entidad subtipo incluidas en tabla supertipo. No hay tabla subtipo.
 - **Tabla por secundario.** Datos de entidad supertipo incluidas en tablas de subtipo. No hay tabla supertipo.
 - **Tabla para cada entidad.** Tabla separada para cada entidad supertipo y subtipo.
 - **Basado en tipo estructurado:** Si se basa en tipo estructurado se selecciona de la lista. Contendrá los atributos definidos en el tipo estructurado.
 - **Supertipo:** Si la entidad es un subtipo, se selecciona el supertipo de la lista. Esta entidad hereda todos los atributos del supertipo especificado.
- **Propiedades de Atributos.** Se añadirán los atributos a la entidad. Las propiedades de cada atributo incluirán su nombre y tipo de datos. Si es el identificador principal se marca *UID*, mientras que si es un identificador de relación se marca *UID de Relación* y si es un campo obligatorio se marca la casilla.

En tipo de datos podremos seleccionar dominios y datos de tipo lógico o de tipo estructurado. Seleccionaremos inicialmente datos de tipo lógico en el que tendremos los tipos: **NUMERIC** para datos numéricos, **VARCHAR** para cadenas y **DATE** para fechas.

Si seleccionamos **NUMERIC** escribimos una *precisión*, que nos indicará el número total de dígitos para la cantidad, y una *escala* para indicar las posiciones decimales. Si seleccionamos **VARCHAR** escribimos el tamaño de la cadena dejando la opción *unidades* por defecto.

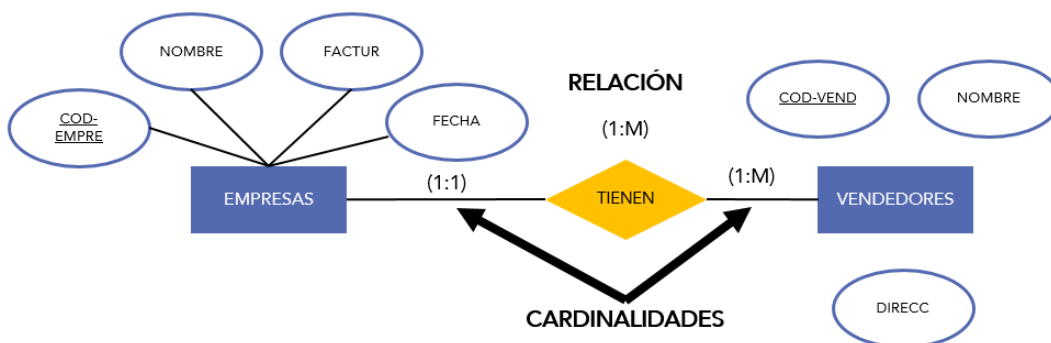
Para agregar atributos pulsamos en el botón Agregar (+) y para eliminarlos hacemos clic en Eliminar (X). Las flechas moverán de posición los atributos. Si queremos ver las propiedades de cada atributo haremos doble clic sobre ellos.

- **Resto de propiedades.** Dejaremos las opciones por defecto para el resto de propiedades, excepto si queremos añadir alguna restricción. Las más importantes son:
 - **Identificadores únicos.** Visualizan identificadores únicos UID y las claves primarias de la entidad PUID (restricción PRIMARY KEY). Se podrán añadir y eliminar identificadores.
 - **Relaciones.** Son las relaciones que se asocian con la entidad.
 - **Propiedades de volumen.** Se refiere a la cantidad de datos y tasa de crecimiento.
 - **Realizar Ingeniería.** Indicación del modelo relacional con el que la entidad tendrá que ser propagada en las siguientes operaciones de ingeniería.
 - **Los comentarios.** Utilizado para añadir comentarios al modelo de datos.

• CREAR RELACIONES

Para crear relaciones utilizamos los botones de la barra de botones del modelo lógico. Creamos primero dos entidades y luego las relacionamos. Las relaciones pueden ser del tipo 1:N, M:N y 1:1. Al crear una relación 1:M, se hace clic en el botón 1:N y primero seleccionamos la entidad de 1 y arrastramos a la entidad del N.

Cuando creamos la relación se abre una ventana de propiedades en la que aparecerá los siguientes elementos:



- **Nombre:** nombre de la relación.
- **Nombre completo:** muestra cómo queda la relación.
- **Cardinalidad del Origen:** *Origen*, *Nombre en origen* y *Sinónimo* indican la entidad de origen de la relación. En *Cardinalidad de Origen a Destino* aparece la cardinalidad máxima. En **Origen Opcional** indicamos la cardinalidad mínima. Hemos de marcarla cuando la cardinalidad sea mínima a 0.
- **Cardinalidad del Destino:** será igual que la anterior, pero con la entidad destino. Cardinalidad de *Destino a Origen* es 1.
- **Suprimir regla:** indica cómo se borrarán los registros relacionados al eliminar un registro de la tabla principal. Con *RESTRICT* no podremos eliminar la tabla si posee registros relacionados. Con *NO ACTION*, bloquea las operaciones y con *SET NULL* permanecen los registros relacionados pero se asigna el valor *NULL* a las columnas relacionadas.
- Pulsaremos en *Aplicar* para ir viendo los cambios y en *Aceptar* para salir del modelo lógico.

• CREAR JERARQUÍAS

Antes de crear jerarquías deberemos establecer primero todas las entidades en las propiedades subtipo:

- En **Estrategia Ingeniería directa**, seleccionamos el tipo de transformación que realizaremos:
 - **Tabla única:** engloba todos los atributos de los subtipos en el supertipo. Se suele elegir esta opción cuando los subtipos se diferencian muy poco entre ellos.
 - **Tabla por secundario:** suprimir el supertipo y se quedan los subtipos con atributos comunes y los suyos propios. Elegiremos esta opción en el caso de que tengamos una gran cantidad de atributos y los accesos sobre los mismos.
 - **Tabla para cada entidad:** genera una tabla separada para cada entidad supertipo y subtipo. Se elegirá esta opción cuando los subtipos tengan muchos atributos distintos.
- En **Supertipo**, seleccionamos la entidad supertipo de los subtipos.

• INGENIERÍA AL MODELO RELACIONAL

Una vez se ha creado el modelo lógico, crearemos el modelo relacional a través de una herramienta automática. En *Data Modeler* pulsamos el botón *Realizar ingeniería al Modelo Relacional* ">>" de la barra de botones. Se iniciará un asistente que indicará los elementos a realizar la ingeniería y podremos cambiar algunas opciones. Cuando terminemos pulsamos en *Realizar ingeniería* para que se genere automáticamente el modelo relacional.

Observamos que la columna *NOT NULL* aparece marcada con punto rojo, las *PRIMARYS KEYS* con una P junto a la columna y las *FOREIGN KEYS* con una F. Las columnas con restricción *UNIQUE* aparecerán marcadas con una U.

En el modelo relacional también podremos cambiar las propiedades de las tablas añadiendo más atributos o restricciones, más tablas y más relaciones.

El paso siguiente será crear las DDL pulsando en el botón *Generar DDL*. Elegimos la base de datos y el modelo relacional y pulsamos en el botón *Generar*. Seguidamente nos aparecerá una ventana para añadir opciones y pulsamos en *Aceptar*. Ahora se nos visualizará una ventana con el script y pulsamos en *Guardar*.

Para crear tablas en el esquema de un usuario, abrimos el archivo creado con *SQLDEVELOPER* desde el menú *Archivo/Abrir*. Pulsamos en el botón *Ejecutar Script*, nos pedirá una conexión y elegimos la conexión de nuestro usuario.

• CREAR DOMINIOS

Es un conjunto de valores que puede tomar un atributo e indica qué valores pueden ser asumidos por los atributos de las tablas. Se define a través de la declaración de un tipo de dato para el atributo, aunque también es posible definir dominios más complejos y precisos.

Data Modeler ya dispone de un administrador de dominios, el cual abriremos pulsando en el menú *Herramientas/Data Modeler/Administración de Dominios*. En la ventana que aparecerá se mostrarán opciones similares a la creación de atributos en los modelos lógicos y relacional.

Esta ventana estará dividida por dos secciones:

- **El cuadro Seleccionar Dominio:** se podrá seleccionar un archivo de dominio en el que se describen los dominios creados. Por defecto se cargará *defaultdomains.xml*, situado dentro de la carpeta del *sqldeveloper*: (*sqldeveloper\sqldeveloper\extensions\oracle.datamodeler\types*). Se mostrarán también los dominios creados.

- En **Propiedades de Dominio** indicaremos:
 - El nombre de dominio.
 - El Tipo Lógico asociado al dominio.
 - Tamaño.
 - La Precisión.
 - La Escala.
 - Restricción de Control nos permite ver y editar restricciones.
 - Rangos, donde añadiremos restricciones para que el dominio se encuentre en un rango de valores.
 - Lista de Valores, donde añadiremos excepciones para que un dominio tome solo valores de una lista.

Al añadir restricciones serán de tipo check cuando vayamos a crear la DLL de una tabla con dominios asociados.

2.3.2. Plugin ER Master de Eclipse

Dispondremos de otra herramienta para crear modelo de datos. Se trata del plugin E/R ERMaster de Eclipse. Algunas de sus características son:

- Soporta los principales Sistema Gestores de Bases de Datos como Oracle, PostgreSQL, Mysql o SQLite.
- Permite importar/exportar los datos existentes.
- Permite generar el Lenguaje de Definición de Datos.
- Permite los modelos físicos y lógicos.

Para más información sobre este plugin podemos visitar la siguiente página:

<http://ermaster.sourceforge.net/>

Para proceder a la instalación abrimos el menú *Help/Install New Software*, en el cuadro *Work With* escribimos:

<http://ermaster.sourceforge.net/update-site/>

Y pulsamos en *Add* y en el siguiente cuadro OK. Una vez hemos localizado el plugin y seleccionado todos los elementos pulsamos en *Next*. Aceptamos los términos y pulsamos en *Finish* para que comience la instalación. Pedirá reiniciar el equipo una vez terminada.

2.3.2.1. Creación de modelos de datos con ER Master

Si queremos crear un modelo de datos con ER Master habrá que hacerlo sobre un proyecto creado. Si no lo tenemos, debemos ir al menú *File/New/Java Project*. Abrimos el menú *File/New/Other* y en la ventana que se muestra elegimos ER Master.

Pulsamos en *Next*, se escribe el nombre para el modelo y elegimos la BD donde se creará el modelo. Al pulsar en *Finish* se abrirá la ventana de diseño. En esta podemos observar tres partes: la barra de botones, la paleta de diseño en la parte izquierda y la vista de edición, en la cual se añadirán entidades y relaciones.

- **IMPORTAR ESQUEMAS**

Antes de crear nuestro propio esquema vamos a ver cómo se importa un esquema de otro usuario. Para ello iremos al menú ***Importar/Database*** y en el cuadro mostrado rellenaremos la información en todos los campos.

Una vez rellenados, pulsamos en *Next* y nos pedirá el *path*, ruta donde se encontrará el ***driver de la BD*** para realizar la conexión. Será necesario tener los drivers de la base de datos –en Oracle se usa *classes12.jar* y en MySQL *mysql-connector-java-5.1.18-bin.jar*–.

En la próxima ventana seleccionamos el esquema a importar. Pulsamos en *Next*, seleccionamos todos los objetos y pulsamos en *OK*.

Para finalizar se muestra toda la información de las tablas, vistas y objetos importados.

- **EXPORTAR ESQUEMA A OTRA BD**

Para exportar un archivo DDL lo haremos desde el menú contextual de la vista de edición. Será necesario indicar una carpeta dónde almacenar el fichero, que normalmente será la carpeta del proyecto. Si la exportación se hace a Java lo haremos en la carpeta *src*. Puede ser que nos aparezca algún error, ya que se creará el nombre del fichero con el nombre de la carpeta incluido, por lo que tendremos que renombrarlo. Lo haremos desde el menú *Refactor/Rename* y quitaremos el nombre de la carpeta.

Es importante saber que si exportamos a la base de datos, esta BD debe estar creada en MySQL y el usuario debe existir en ORACLE, ya que se borrarán todos los datos contenidos en la BD de MySQL. Las BD deben estar iniciadas.

- **CREAR MODELOS**

Para crear un modelo iremos al menú *File/New/Other*, seleccionamos *ER Master* y elegimos la carpeta del proyecto de trabajo, le damos un nombre y elegimos la BD.

En ER Master no podremos crear jerarquías, por lo que deberemos saber el tipo de relación que tendrán con anterioridad y se representarán con una tabla con todos los atributos.

Para crear tablas usaremos los botones de la *Paleta de Diseño*:

- *Create Table*, para crear tablas.
- *Create Relation 1:N*, marcaremos primero la tabla desde donde sale el 1 al crear estas relaciones.
- *Create relation by existing columns*, crea relaciones que tienen asociada una columna.
- *Create N:N Relation*, para relaciones M:M.
- *Self Relation*, para relaciones reflexivas.

Al crear una tabla nos mostrará la ventana *Table Information*, la cual nos detallará los campos para añadir sus propiedades.

A la hora de crear las relaciones, seleccionaremos la tabla origen y la de destino. Una vez relacionadas las dos tablas haciendo doble clic, tendremos toda la información.

Al crear las relaciones, las claves se traspasan también a la tabla de destino. El siguiente paso será crear este modelo de datos dentro de una base de datos MySQL, por lo que pulsaremos en el botón *Exportar a BD* y procedemos como en pasos anteriores.

UF2. Optimización de Software

1. Diseño y realización de pruebas

En esta segunda unidad formativa vamos a aprender cómo se usan las distintas técnicas para realizar casos de prueba. Utilizaremos una herramienta de depuración en la que se definen puntos de ruptura y se examinan variables durante la ejecución del programa. Además, aprenderemos a usar la herramienta JUNIT para poder elaborar pruebas unitarias para clases Java.

1.1. 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 para conseguir un objetivo particular de prueba. Será necesario que se definan unas pre-condiciones y post-condiciones, 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 o no.

Para el diseño de los casos de prueba tendremos dos tipos de técnicas, como ya hemos visto en la anterior unidad: la **prueba de caja blanca**, centrada en validar la estructura interna del programa, y la **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.

1.1.1. Pruebas de caja blanca

También se les llama pruebas estructurales o de caja de cristal. Su funcionamiento se basa en un exhaustivo examen de los detalles **procedimentales del código**. 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 en este tipo será la del camino básico que abordaremos más adelante.

1.1.2. Pruebas de caja negra

También se les conoce como prueba de comportamiento. Se realiza sobre la interfaz sin necesidad de conocer la estructura del programa ni cómo funciona. Lo que se busca es que demuestren 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:

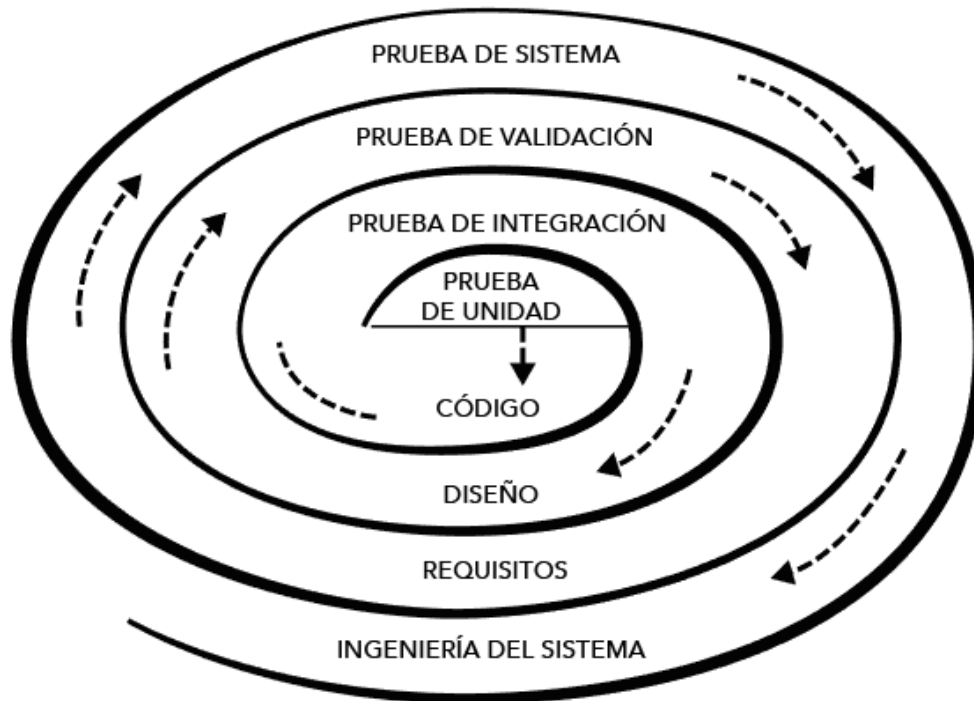
- Errores de interfaz, en el acceso a los datos o en las bases de datos externas.
- Funcionalidades erróneas en el inicio o la finalización del programa.

Algunas técnicas para estos casos son: clases de equivalencia, análisis de valores límite, métodos basados en grafos, pruebas de comparación, etc. Más adelante veremos algunos de ellos.

1.2. Estrategias de pruebas de software

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.



1.2.1. 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 asegurar 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, etc.

1.2.2. 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 donde realmente se comprueba que funcionan correctamente al tener que hacerlo de forma conjunta.

Podemos enfocarlo de dos formas distintas:

- 1) **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.
- 2) **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:
 - a. **Ascendente.** Se comienza con los módulos más bajos del programa.
 - b. **Descendente.** Se empieza por el módulo principal descendiendo por la jerarquía de control.

1.2.3. Prueba de validación

Conseguiremos la prueba de validación cuando el programa funcione de acuerdo con las expectativas expuestas por el cliente y cuando, además, cumpla con lo indicado en el documento de especificación de requisitos del software o ERS. 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.

1.2.4. Prueba del sistema

Esta prueba está formada por varias pruebas que tendrán como misión ejercitar en profundidad el software. Serán las siguientes:

- **Prueba de recuperación:** se fuerza el fallo del software y que la recuperación se realice correctamente.
- **Prueba de seguridad:** se comprueba que el sistema esté protegido frente a acciones ilegales.
- **Prueba de resistencia (Stress).** se realizan acciones que requieran una gran cantidad de recursos.

1.3. Pruebas de código: cubrición, valores límite y clases de equivalencia.

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, observamos y registramos los resultados para compararlos con los resultados que esperamos. Comprobaremos si el resultado es el esperado o no y por qué.

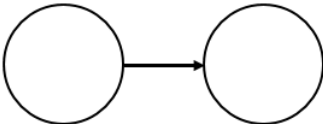
Para este tipo de pruebas se mostrarán diferentes técnicas que van a depender del tipo de enfoque que queramos utilizar: de caja blanca o negra.

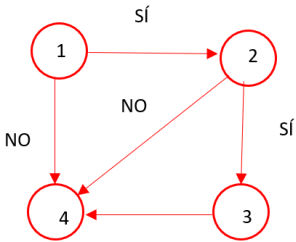
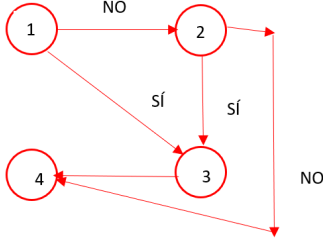
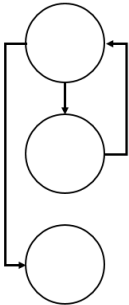
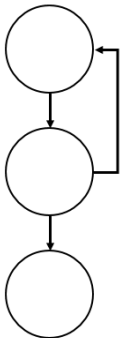
1.3.1. Prueba del camino básico

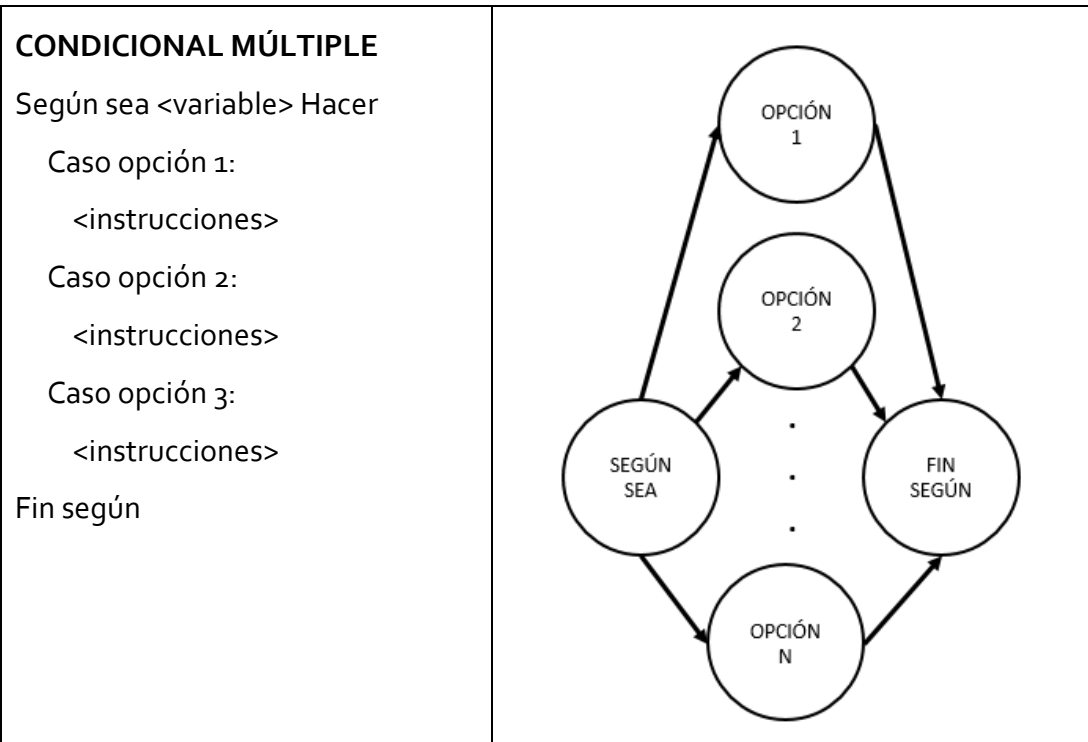
Es una técnica que, mediante prueba, permite al desarrollador obtener la medida de la complejidad de nuestro sistema. Puede usar esta medida para la definición de un conjunto de caminos de ejecución. Para obtener esta medida de complejidad utilizaremos la técnica de representación de grafo de flujo.

- **NOTACIÓN DE GRAFO DE FLUJO**

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

ESTRUCTURA	GRAFO DE FLUJO
SECUENCIAL Instrucción 1 Instrucción 2 ... Instrucción n	 <pre> graph LR A(()) --> B(()) </pre>

<p>CONDICIONAL</p> <p>Si <condición> Entonces <instrucciones></p> <p>Si no <instrucciones></p> <p>Fin si</p> <p>IF condición AND</p>  <pre> 1 2 IF (A && B) { 3 // código 4 } </pre>	 <pre> 1 2 IF (A B) { 3 // código 4 } </pre>
<p>HACER MIENTRAS</p> <p>Mientras <condición Hacer> <instrucciones></p> <p>Fin mientras</p>	
<p>REPETIR HASTA</p> <p>Repetir <instrucciones></p> <p>Hasta que <condición></p>	



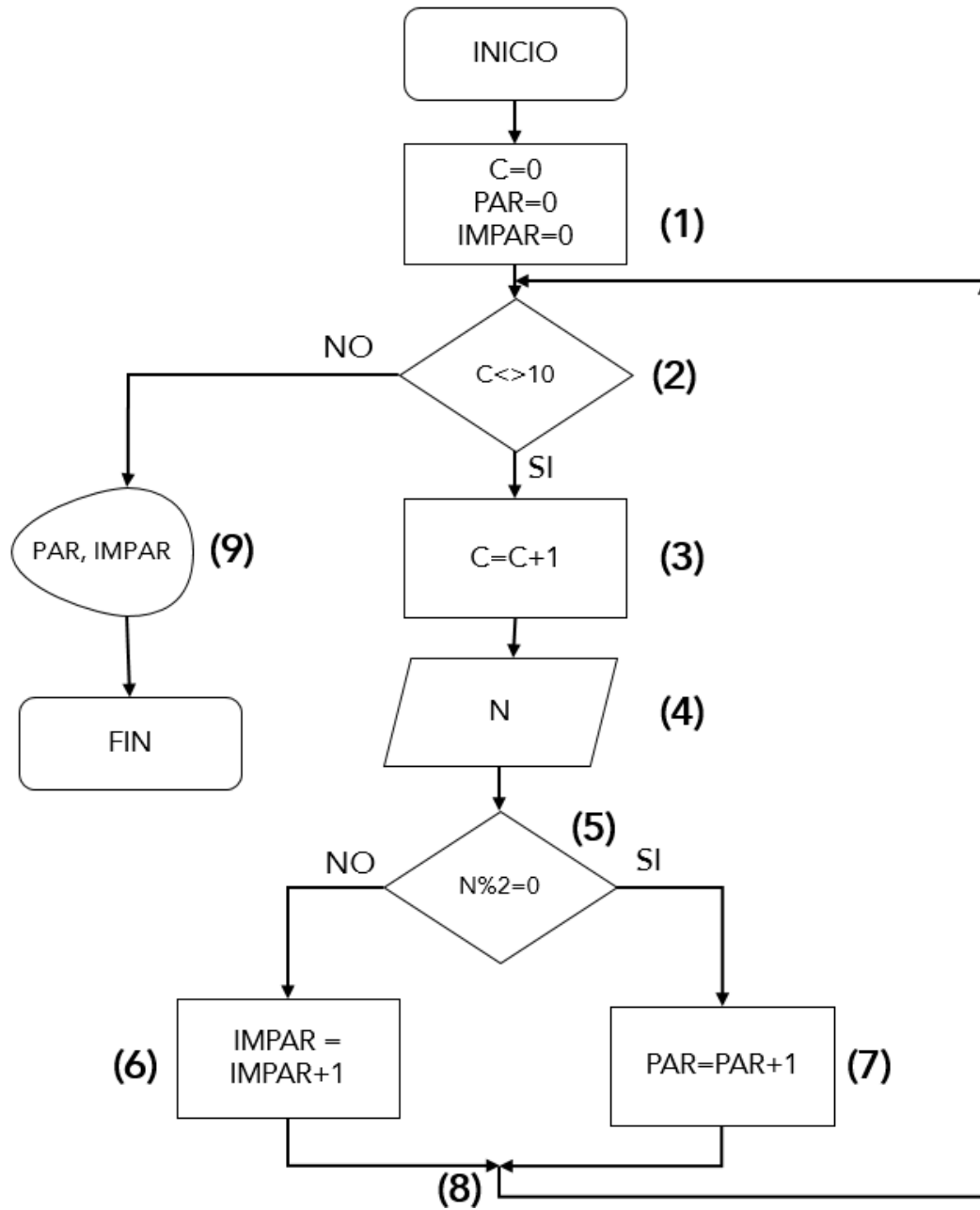
Cada círculo va a representar una o más sentencias, sin bifurcaciones, en pseudocódigo o código fuente. Se numerarán en el diagrama de flujo cada símbolo y los finales de las estructuras, aunque no tenga ningún símbolo.

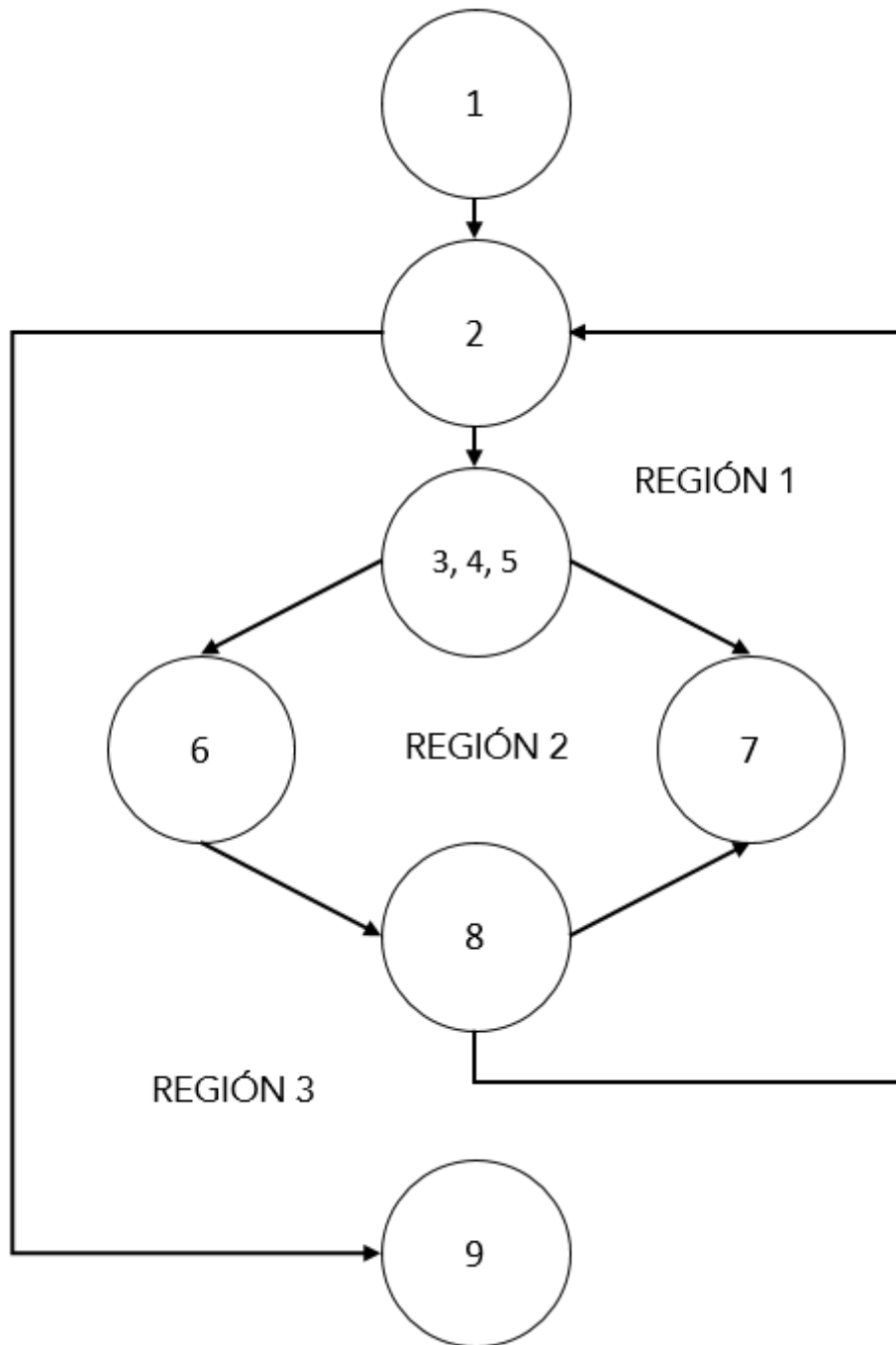
En el grafo de flujo, cada círculo se llamará *nodo*. Va a representar a una o más sentencias procedimentales. Uno solo podrá corresponder con una secuencia de símbolos y un rombo una decisión.

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 procedimental.

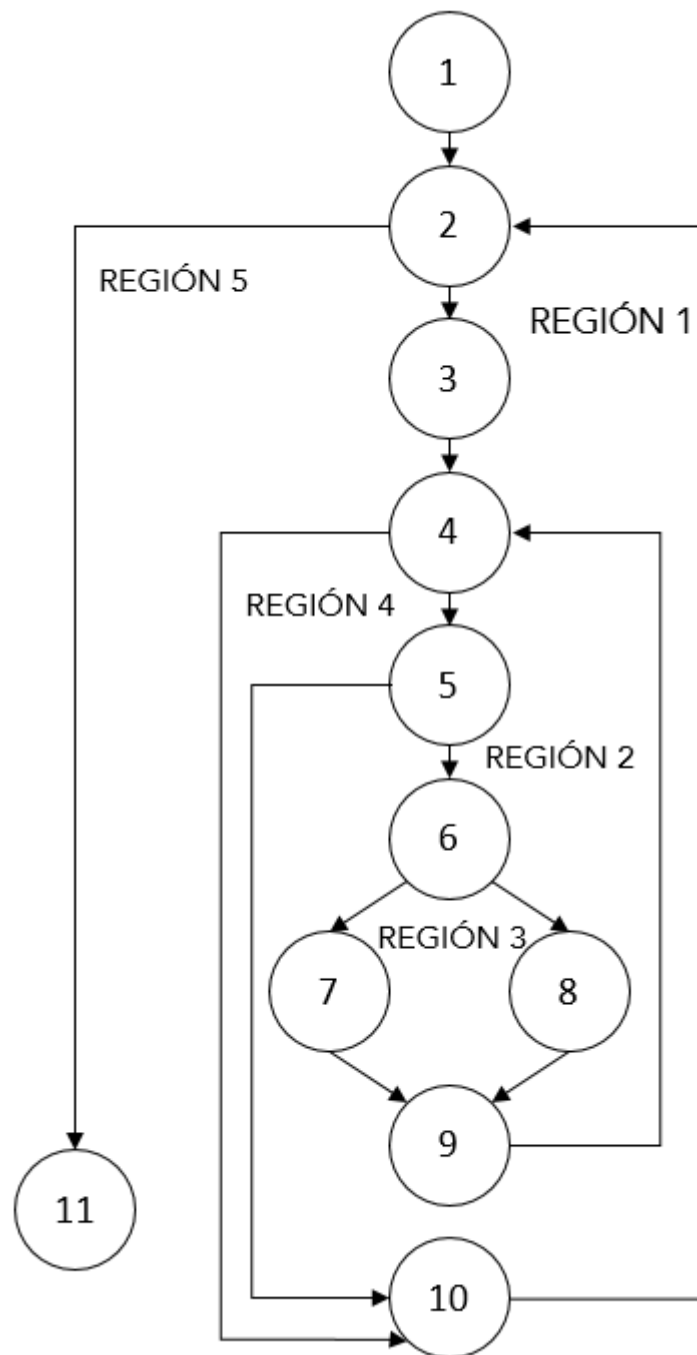
Las *regiones* son áreas que estarán delimitadas por aristas y nodos. Cabe destacar que el área exterior del nodo es otra región más.

El *nodo predicado* contendrá una condición y su principal característica es que salen dos o más nodos de él. Solo de estos nodos podrán salir dos aristas.





Cuando nos encontramos con condiciones compuestas (aparecen uno o más operadores) se nos complicará la generación del grafo de flujo. Tendremos que crear un nodo aparte para cada una de las condiciones.



Inicio

Abrir fichero alumnos
Leer Alumnos (Curso, Nombre, Sexo, Nota)

Mientras Haya registros **Hacer**

NH = 0, NM = 0
Visualizar Curso

Mientras Haya registros **y** Mismo curso **Hacer**

Si Sexo = "H" **Entonces**

NH = NH+1

Si no

NM = NM+1

Fin si

Leer Alumnos (Curso, Nombre, Sexo, Nota)

Fin Mientras

Visualizar NH, NM

Fin Mientras

Abrir Fichero Alumnos

Fin

- **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 3 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

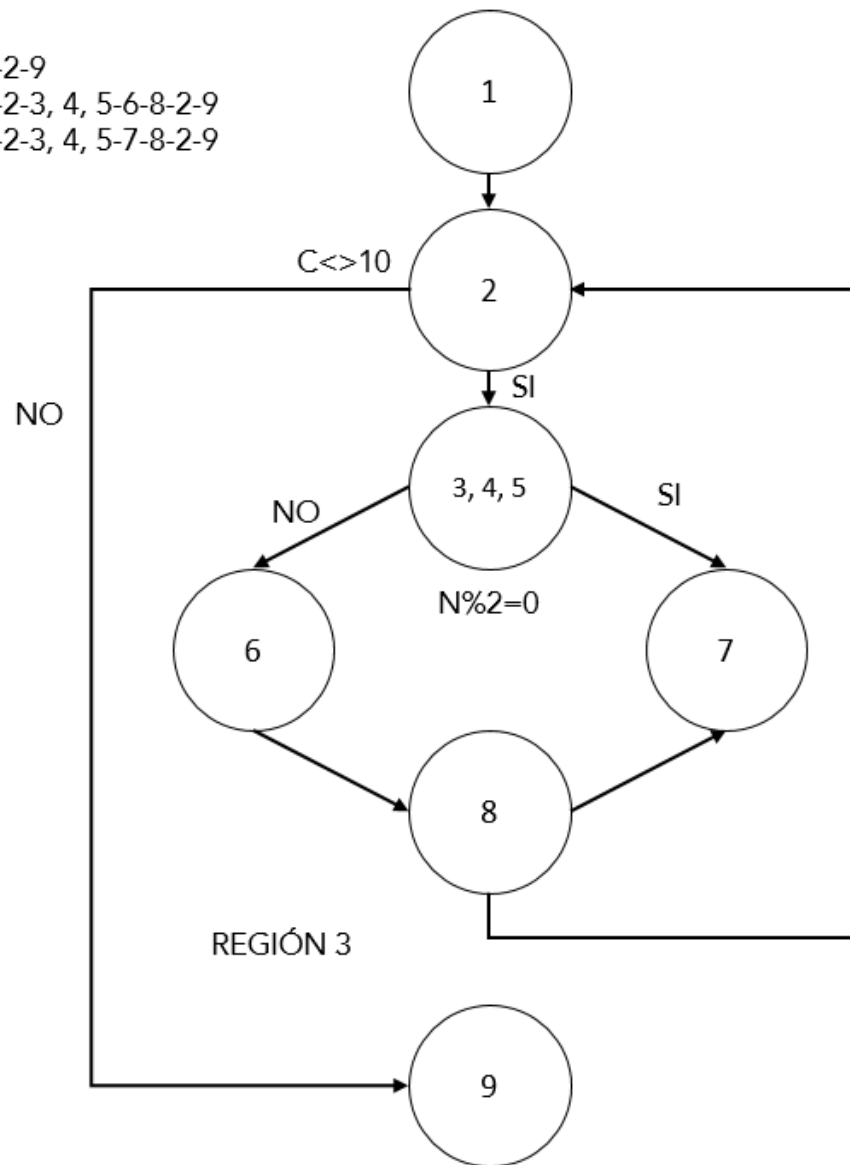
El valor $V(G)$ nos va a dar el número de caminos independientes del conjunto básico de un programa. Un camino independiente será un camino en el cual se introducirán nuevas sentencias de proceso o una condición. Referente a los diagramas de flujo, estarán constituidos por al menos una arista que no ha sido recorrida anteriormente a la definición del camino.

- **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 predicado estén establecidas adecuadamente. Podemos representar 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 $C \leq 10$ $C = 10$	Visualizar el número de pares y de impares
2	Escoger algún valor de C tal que SI se cumpla la condición $C \leq 10$ 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 SI se cumpla la condición $C \leq 10$ Escoger algún valor de N tal que SI se cumpla la condición $N \% 2 = 0$ $C = 2, N = 4$	Contar números pares

Camino 1: 1-2-9
 Camino 2: 1-2-3, 4, 5-6-8-2-9
 Camino 3: 1-2-3, 4, 5-7-8-2-9



El camino 1 no podrá ser probado por sí solo, deberá formar parte también de las pruebas 2 y 3.

Los caminos independientes y los casos de prueba para cada camino se muestran en la siguiente tabla:

Camino	Caso de prueba	Resultado esperado
Camino 1: 1-2-3-5-6	Escoger algún X e Y que NO se cumpla la condición $X < 0 \parallel Y < 0$ $X=4, Y=5$ <code>visualizarMedia(4,5)</code>	Visualiza: La media es 4,5
Camino 2: 1-2-4-6	Escoger algún X tal que SI se cumpla la condición $X < 0$ (Y puede ser cualquier valor) $X=4, Y=5$ <code>visualizarMedia(-4,5)</code>	Visualiza: X e Y deben ser positivos
Camino 3: 1-2-3-4-6	Escoger algún X tal que NO cumpla la condición $X < 0$ y escoger algún Y que SI cumpla la condición $Y < 0$ $X=4, Y=5$ <code>visualizarMedia(4,-5)</code>	Visualiza: X e Y deben ser positivos

1.3.2. Partición o clases de equivalencia

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 podrá definir dos tipos de clases de equivalencia:

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

Vamos a seguir una serie de directrices para definir las equivalencias:

- 1) Si una condición de entrada requiere un **rango**, se define **una** clase de equivalencia válida y **dos** no válidas.
- 2) Si requiere un **valor específico**, define **una** clase válida y **dos** no válidas.
- 3) Si especifica un **miembro de un conjunto**, define **una** válida y **una** no válida.
- 4) Si es **lógica**, define **una** clase válida y **una** no válida.

1.3.3. 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. Complementa a la técnica anterior. Además, no solo estará centrado en las condiciones de entrada, sino que 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 ejerciten la estructura de datos en sus límites, primer y último elemento.

1.4. Pruebas unitarias con JUNIT

Hasta ahora, hemos realizado pruebas de forma manual. En este apartado veremos cómo funciona un programa que realiza pruebas para verificar nuestro programa.

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.

1.4.1. 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
assertTrue(boolean expresión) assertTrue(String mensaje, boolean expresión)	Comprueba que la expresión se evalúe <i>true</i> . Si no es <i>true</i> y se incluye el String, al producirse error se lanzará el <i>mensaje</i> .
assertFalse(Boolean expresión) assertFalse(String mensaje, Boolean expresión)	Comprueba que la expresión se evalúe <i>false</i> . Si no es <i>false</i> y se incluye el String, al producirse error se lanzará el <i>mensaje</i> .
assertEquals(valorEsperado, valorReal), assertEquals(String mensaje, valorEsperado, valorReal)	Comprueba que el <i>valorEsperado</i> sea igual al <i>valorReal</i> . Si no son iguales y se incluye el String, entonces se lanzará el <i>mensaje</i> . <i>ValorEsperado</i> y <i>ValorReal</i> pueden ser de diferentes tipos.
assertNull(Object objeto), assertNull(String mensaje, Object objeto)	Comprueba que el <i>objeto</i> sea <i>null</i> . Si no es <i>null</i> y se incluye el String, al producirse error se lanzará el <i>mensaje</i> .
assertNotNull(Object objeto), assertNotNull(String mensaje, Object objeto)	Comprueba que el <i>objeto</i> no sea <i>null</i> . Si no es <i>null</i> y se incluye el String, al producirse error se lanzará el <i>mensaje</i> .
assertSame(Object objetoEsperado, Object objetoReal) assertSame(String mensaje, Object objetoEsperado, Object objetoReal)	Comprueba que <i>objetoEsperado</i> y <i>objetoReal</i> sean el mismo objeto. Si no son el mismo y se incluye el String, al producirse el error se lanzará el <i>mensaje</i> .
assertNotSame(Object objetoEsperado, Object objetoReal) assertNotSame(String mensaje, Object objetoEsperado, Object objetoReal)	Comprueba que <i>objetoEsperado</i> y <i>objetoReal</i> no sean el mismo objeto. Si no son el mismo y se incluye el String, al producirse el error se lanzará el <i>mensaje</i> .
fail() fail(String mensaje):	Hace que la prueba falle. Si se incluye un String la prueba falla lanzando el <i>mensaje</i> .

1.4.2. Tipos de anotaciones

JUnit tiene disponible unas anotaciones que permite 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 se hayan finalizado todas las pruebas.

1.4.3. Pruebas parametrizadas

Para ejecutar una prueba varias veces, pero con distintos valores, JUnit nos lo permite siguiendo los siguientes 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 a testar incluiremos en el método la etiqueta **@Parameters**.

1.4.4. Suite de pruebas

Para poder ejecutar una serie de pruebas, unas tras otras, necesitamos un mecanismo de JUnit llamado Test Suites. Para crearla vamos al menú: *File -> New -> Other -> Java -> JUnit -> JUnit Test Suite* y pulsamos en *Next*. Elegimos las clases que formarán parte de la suite y marcamos la opción *New JUnit 4 suite* y la nombramos. A continuación, pulsamos en *Finish*.

Podemos destacar dos anotaciones: **@RunWith(Suite.class)**, que le indica a JUnit que es una suite de pruebas, y **@SuiteClasses()**, que indica las clases que forman parte del conjunto de pruebas y las que se ejecutarán. No se generará ninguna línea de código dentro de la clase.

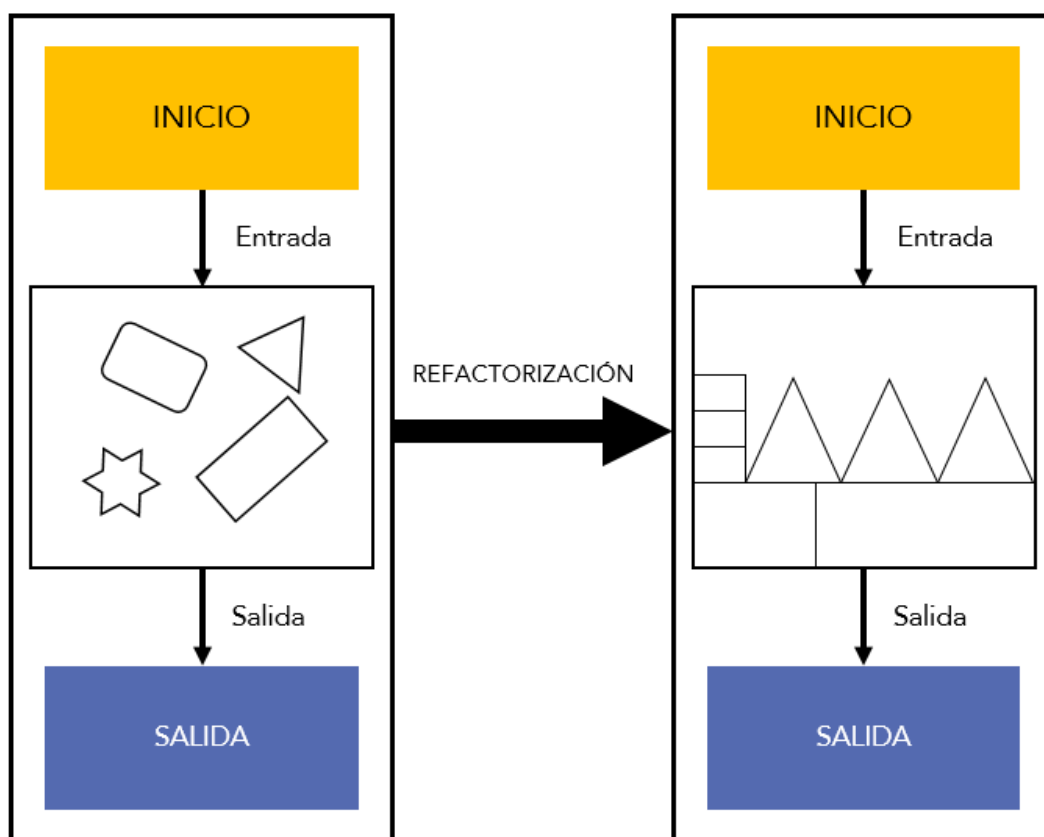
2. Documentación y optimización.

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

2.1. Refactorización. Concepto. Limitaciones. Patrones de refacción más usuales.

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.



2.1.1. Cuándo refactorizar. Malos olores (*bad smells*)

La refactorización la deberemos ir haciendo a medida que desarrollamos el software. En 2003, Mario G. Piattini 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:

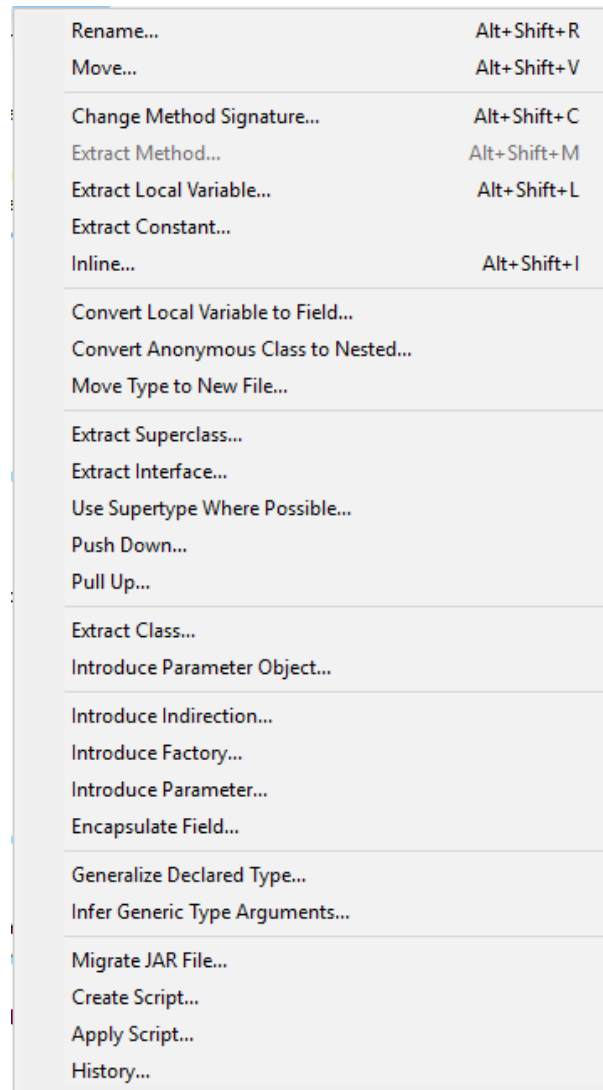
- **Código duplicado** (*Duplicated code*). Esta será la principal razón para realizar la refactorización. Si encontramos algún código repetido deberemos unificarlos.
- **Métodos muy largos** (*Long method*). Los métodos largos normalmente pueden estar compuestos de métodos más pequeños por lo que deberemos 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. Éstos 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.
- **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 facilidad 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.

2.1.2. Refactorización en Eclipse

En Eclipse disponemos de distintas formas de refactorizar. En función de donde lo hagamos tendremos un menú contextual u otro. Para ello, deberemos ir a la opción **Refactor** del menú contextual.



- **MÉTODOS DE REFACTORIZACIÓN**

Son prácticas para refactorizar el código. A través de distintas herramientas plantearemos elementos para refactorizar y 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 realizada, 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 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 y 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. Si es estática se realizará inmediatamente y, si no lo es, nos pedirá el nombre de la clase.
- **Extract Interface.** Nos va a permitir escoger los métodos de una clase para crear una *Interface*. Una *Interface* es una plantilla que define los métodos pero no los desarrolla. Serán las clases de la *interface* 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.
- **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 interface (sin *implements*) y la definición de la clase anónima entre llaves.

2.2. Control de versiones. Estructura de las herramientas de control de versiones.

El control de versiones es 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.

- **TERMINOLOGÍA**

Veamos algunos términos útiles en 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 el 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.

2.2.1. Subversión. Ciclo de vida de subversión

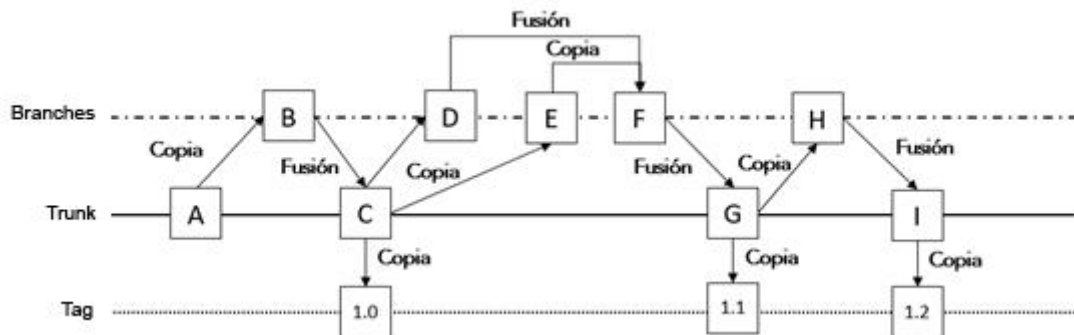
Durante el desarrollo de un proyecto, es fundamental el uso de una herramienta multiplataforma de código abierto que garantice el control de versiones proceso conocido como subversión. Esta herramienta usa una base de datos central llamada repositorio que contendrá archivos cuyas versiones e historias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

La subversión 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 es útil 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.



2.2.2. Cliente TortoiseSVN

TortoiseSVN es un cliente gratuito de código abierto para el sistema de control de versiones *Subversion*. Cuando se instala aparece integrado en la *shell* de *Windows*.

Podemos descargarlo en la siguiente URL:

<http://tortoisesvn.net/downloads.html>

Estará disponible para 32 y 64 bits, y también existirá el instalador del idioma. En esta unidad trabajaremos con la versión 1.8.4. Primero instalaremos la herramienta y después el idioma.

El proceso de instalación es muy sencillo. Solo debemos aceptar los términos de la licencia, indicamos la carpeta donde se instalará y pulsamos en *Instalar*. Seguidamente instalamos el idioma. Para configurarlo, pulsamos el botón derecho del ratón en el escritorio, seleccionamos *TortoiseSVN/Settings* y el cuadro que nos aparece en General seleccionamos el idioma en *Language*. Seleccionamos español y comprobamos que las opciones que nos aparecen están en español.

Ahora vamos a ver cómo se crea un repositorio en el PC para controlar los archivos y documentos que se almacenan en él.

Para crearlo debemos crear una carpeta en nuestro disco duro, la seleccionamos y haciendo clic derecho elegimos la opción *TortoiseSVN* y accedemos a la opción *Crear repositorio aquí*. A continuación, hacemos clic en el botón *Crear estructura de carpetas* (se creará la estructura *branches*, *tags* y *trunks*) y pulsaremos *Navegador de repositorios* para que nos muestre el navegador. La revisión creada será la 1 e irá aumentando conforme hagamos cambios en el repositorio. Para acceder a las demás revisiones pulsamos en el botón *HEAD*.

Podremos crear todos los repositorios que queramos y administrarlos desde el navegador. Se creará la estructura de carpetas ***trunk-tags-branches***. Esa estructura variará dependiendo de la organización que queramos, ya que podremos organizarlo por ramas o por proyectos.





2.2.2.1. Operaciones con Tortoise




Para probar esta herramienta podremos usar archivos y carpetas de nuestro disco duro y tanto desde dentro del repositorio como desde fuera. Dentro del navegador y desde el menú contextual de *TortoiseSVN* se podrán seleccionar las operaciones a realizar.

• OBTENIENDO INFORMACIÓN DEL ESTADO DEL REPOSITORIO

Cuando estamos trabajando con nuestra copia de trabajo es importante saber en qué estado se encuentran los archivos con los que trabajamos, si han sido cambiados, añadidos, borrados, etc. En este sentido, los archivos tendrán unos iconos superpuestos. Podemos cambiar su configuración desde *TortoiseSVN/Configuración/Iconos Sobrepuestos*.

Los iconos más comunes son:

-  Una copia de trabajo recién obtenida y sincronizada indica estado de Subversión normal.
-  Al editar un archivo, el estado cambia a modificado y el icono cambia entonces a una marca de exclamación roja.
-  Este icono muestra que algunos archivos o carpetas dentro de la carpeta actual se han marcado para ser eliminados del control de versiones, o bien que falta un archivo que está bajo el control de versiones dentro de una carpeta.
-  El signo "+" indica que el archivo o carpeta está programado para ser añadido al control de versiones, se le ha indicado que se añadirá desde el menú *TortoiseSVN/Añadir*.

-  Si durante una actualización ocurre un conflicto, el icono cambia a un signo de exclamación amarillo.
-  Si se ha bloqueado un archivo y el estado de Subversión es normal, este icono recordará que se debe liberar el bloqueo si no se está utilizando. Ello permitirá a los demás que puedan confirmar sus cambios en el archivo.
-  Este icono muestra los archivos y carpetas que no están bajo el control de versiones, pero tampoco han sido ignorados. Es un icono opcional y suele aparecer cuando hay conflictos, marca con el icono a las versiones del archivo.

• RESOLVER CONFLICTOS

Una vez ya hemos realizado los cambios, subimos los archivos al repositorio con la opción *SVN Confirmar*. Si alguien ha realizado cambios en el fichero antes que nosotros se producirá un **conflicto**, *TortoiseSVN* nos avisará del fallo y no subirá el cambio. Podremos seleccionar que se actualice o se cancele la copia. Si actualizamos se creará una copia de cada archivo y entrará en estado "En conflicto". Para resolverlo pulsaremos en *Resolver* dentro del menú.

2.2.2.2. Creación de ramas y etiquetas con Tortoise

Al principio vimos que se podrían crear distintas líneas para corregir errores, realizar cambios o añadir funcionalidades. A estas líneas se les llama ramas. Cuando es estable, la nueva línea se fusiona con la rama principal (tronco).

También es posible realizar revisiones particulares para recrear ciertos entornos, acción conocida como etiquetar.

No se crearán copias del repositorio, sino que se crearán vínculos internos, apuntando a una revisión y a un árbol específicos. De esta forma, las ramas y las etiquetas se crearán de forma más rápida sin ocupar espacio en el repositorio.

• CREAR ETIQUETAS

Para crear carpetas, nos colocamos en la carpeta o archivo dentro del repositorio local y hacemos clic con el botón derecho. Seleccionamos *Rama/etiqueta* del menú. En la ventana se indicará el origen y el destino de la etiqueta y sobre qué elementos se crea.

Antes de crear las ramas deberemos crear una carpeta en el repositorio dentro de *branches* o *tags*, para que no cuelguen directamente desde ellas.

Al crear la etiqueta nos aparecerá:

- En **CT/URL**: indicar la carpeta que se desea etiquetar del repositorio local.
- En **Ruta Destino**: indicar la carpeta destino de la etiqueta. Seleccionamos la carpeta que hemos creado dentro de *tags* y añadimos un nombre de versión. Se creará una carpeta nueva donde guardarán los archivos a etiquetar.

Para elegir el origen de la copia dispondremos de 3 opciones:

- **Revisión HEAD en el repositorio**: la nueva etiqueta/rama se copia directamente en el repositorio desde la revisión HEAD del mismo.
- **Revisión específica en el repositorio**: la nueva etiqueta/rama se copia directamente en el repositorio, eligiendo la versión anterior. No se transfieren datos.
- **Copia del trabajo**: la nueva etiqueta/rama es una copia idéntica de la copia local de trabajo. Si se ha realizado algún cambio, irá directamente a la copia.

• CREAR RAMAS

El proceso es similar a las etiquetas. Tendremos que crear una carpeta dentro de *branches* poniendo el mismo nombre a la carpeta a enramar. Trabajar desde un *tag* no es buena idea. Pero si se necesitan realizar cambios se deberá crear una rama desde la etiqueta, realizar los cambios y después crear una nueva etiqueta con esta rama. Si se realiza algún cambio desde una rama y se confirman, los cambios irán a la rama y no al tronco.

• FUSIONAR RAMAS

Los cambios se realizarán siempre en las ramas y, una vez se han fusionado, las modificaciones al tronco se confirmarán desde las copias de trabajo asociadas. **La fusión siempre se realizará sobre copias de trabajo.**

Para fusionar los cambios realizados en la rama, se abre la carpeta sobre la que se creó la rama y, dentro de esa carpeta modificada, se abre el menú *TortoiseSVN* y se elige *Fusionar*. Aparecen dos opciones de fusión:

- **Fusionar un rango de revisiones**: Este método cubre el caso en el que ha hecho una o más revisiones a una rama (o al tronco) y desea portar estos cambios a una rama diferente. La dirección de la carpeta que deseamos cargar en la copia de trabajo la escribiremos en el campo *URL desde*.
- **Fusionar dos árboles diferentes**: Este método se usará cuando queramos fusionar las diferencias de dos ramas distintas en una copia de trabajo. Se le dirá que haga los cambios necesarios para ir desde la *revisión inicial del rango a fusionar*, hasta la *revisión del fin del rango a fusionar*, y que se apliquen esos

cambios a la copia de trabajo. El objetivo de esta fusión es que la copia del tronco sea igual a la de la rama.

2.2.3. Servidor subversión, VisualSVN Server

VisualSVN Server es un servidor de subversión que va a funcionar como servidor web. Se instala y administra fácilmente e incluye un servidor *Apache* (servidor web HTTP de código abierto).

Podemos descargarlo desde:

<http://www.visualsvn.com/server/>

El servidor podremos administrarlo desde *Management Console*. Los repositorios de *VisualSVN Server* están almacenados en formatos estándar y se podrá acceder con clientes estándar como *TortoiseSVN*. Es muy completo y se puede integrar en una red corporativa basada en Windows.

La instalación es muy sencilla, después de iniciar el ejecutable tendremos que aceptar los términos de la licencia, seleccionamos la opción *VisualSVN Server and Management Console* y la **Standard Edition**.

Para configurar el Server indicamos:

- En **Location**: carpeta de instalación de la aplicación.
- En **Repositories**: dónde se almacenarán los repositorios.
- En **Server Port**: puerto que se usará para conexiones seguras (si tenemos una instalación de Apache anterior conviene cambiar el puerto que viene por defecto).

Si todo está correcto seleccionamos la opción *Start VisualSVN Server* y pulsamos en *Finish*. Se abrirá la ventana de administración desde la que podremos crear los usuarios y repositorios.

Para crear un repositorio pulsamos sobre *Repositories* con el botón derecho del ratón y seleccionamos *Create new Repository*. Lo nombramos y pulsamos en siguiente, y en la próxima ventana indicamos si el repositorio se creará vacío o con la estructura **trunks-branches-tags**. En nuestro caso vamos a elegir la opción **trunks-branches-tags**.

Ahora podremos elegir el tipo de acceso de los usuarios, pero dejaremos las opciones seleccionadas por defecto. Para finalizar pulsamos en *Create* y en la ventana se nos mostrará la URL que tendremos que usar para conectarnos desde los clientes como *TortoiseSVN* o vía web.

Tendrá este formato:

https://NOMBRE_EQUIPO:PUERTO/svn/NOMBRE_REPOSITORIO

Al salir, nos mostrará en la consola el repositorio creado. El siguiente paso será la creación del usuario para poder conectarnos al repositorio. Nos colocamos en la carpeta **Users** y pulsamos con el clic derecho del ratón para elegir **Create User**. Escribiremos el nombre de usuario y la contraseña.

Ahora solo nos quedará probar si funciona todo correctamente. Para ello, copiaremos la URL anterior en el navegador y, cuando nos pida el usuario y la contraseña, escribiremos los creados anteriormente. Si vemos nuestro repositorio significa que lo hemos creado bien.

2.2.4. Subversión en SQL Developer

En este apartado vamos a usar la versión *sqldeveloper-4.0.2.15.21*, ya que añade más funcionalidades.

Desde SQL Developer podemos crear repositorios o conexiones con otros servidores. Para ello, accedemos al menú *Equipo/Subversión/* y elegimos *Crear Conexión* para conectarnos con un servidor de repositorios como es *VisualSVN Server* o crear un repositorio local.

Para crear una conexión con el repositorio creado recientemente elegimos *Equipo/Subversión/* y elegimos *Crear Conexión*, escogiendo la conexión manual y, en la ventana que nos muestra, completamos los datos del servidor (el nombre, la URL y el usuario con la clave).

- **AÑADIR PROYECTO AL REPOSITORIO**

Si queremos añadir un proyecto a un repositorio lo abriremos con Data Modeler desde el menú *Archivo/Data Modeler/Abrir*. Una vez abierto, con el explorador de Data Modeler lo importamos al repositorio. Para ello, iremos al menú *Equipo/Subversión/Importar Archivos*. Ejecutamos el *Asistente de importación a Subversión*, seleccionamos el repositorio y la carpeta que está dentro del repositorio donde importaremos el proyecto. Pulsamos siguiente y elegimos la carpeta a importar (se podrá añadir un comentario para describir el momento de la subida). Pulsamos en siguiente, se dejan las opciones por defecto y le damos a finalizar.

Podremos visualizar los mensajes de importación y el número de versión en la *Consola de SVN*. En la nueva carpeta que hemos importado se crearán dos nuevas carpetas, una con el sufijo *.svn-import-backup* y otra con *.svn-import-workarea*.

Si queremos hacer *checkout* desde el repositorio a una carpeta local seleccionamos la carpeta del repositorio y elegimos *Desproteger* en el menú contextual.

Podemos acceder desde el menú *Equipo/Data Modeler* a todas las opciones de control de versiones sobre los diseños de Data Modeler.

2.2.5. 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 SVN y, entre los resultados mostrados, seleccionamos *SVN Team Provider*. Pulsamos en instalar, seleccionamos y confirmamos los elementos a instalar y aceptamos los términos de la licencia para comenzar la instalación.

Si queremos conectar un proyecto a un repositorio será necesario instalar los conectores de *Subversive*. La ventana que nos permitirá seleccionar los conectores nos aparecerá cuando creemos el proyecto o lo carguemos. Si no nos aparece, tendremos que comprobar que el conector no esté instalado desde el menú *Window/Preferences/Team/SVN/SVN Connector*, pestaña *SVN Connector*. Si es así será necesario instalarlos.

Al instalar el conector habrá que saber qué versión de *Subversive* hemos instalado. Podremos comprobarlo desde la administración de consola del servidor *VisualSVN*, haciendo clic en el enlace de la versión de *VisualSVN*. En el cuadro de diálogo visualizaremos la versión de subversión y apache instalada.

- **AÑADIR UN PROYECTO AL REPOSITORIO**

Una vez tenemos todo instalado, tanto el plugin como el conector, procedemos a incluir un proyecto al repositorio. Sin embargo, antes deberemos crear un repositorio nuevo con la misma estructura que el anterior. Para ello, seleccionamos un proyecto ya creado o creamos uno nuevo y pulsamos sobre él con el botón derecho del ratón, eligiendo la opción ***Team/Share Project***.

En el primer cuadro seleccionamos el tipo de repositorio al que nos conectaremos, elegimos *SVN* y pulsamos en *Next*. Ahora nos pedirá la información del repositorio y en la pestaña ***Advanced*** desmarcamos la casilla para que se cargue dentro de *trunk* nuestro proyecto.

En la pestaña ***General*** indicamos la URL del proyecto. Pulsamos en *Browse* para elegir la carpeta *Trunk* de nuestro proyecto y nos pedirá la conexión con el usuario. En *Authentication* añadiremos los datos del usuario. Pulsamos en *Next*. Y seguidamente en *Finish*.

Al observar el proyecto vemos que cada nodo tendrá un icono indicando que está en un repositorio, y también aparecerá el número de revisión al lado.

- **OPERACIONES CON SVN ECLIPSE**

A la hora de realizar las operaciones con Eclipse veremos que podremos realizar las mismas operaciones que 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.

Cuando hay **cambios entrantes** nos aparece una flecha hacia dentro en azul, mientras que si aparece el signo "+" es que el archivo es nuevo. Tendríamos que actualizar los cambios entrantes.

- **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.

- **CREAR RAMAS**

Se hará desde la perspectiva SVN Repository Exploring y nos situaremos en la raíz del repositorio.

Para subir una rama hay que situarse en el elemento *Branches* de la raíz, y creamos una carpeta dentro de la rama. Ahora nos posicionamos en **ROOT** (seguimos en la raíz), desplegamos y buscamos el proyecto a subir a la rama. Sobre de él, pulsamos el botón derecho y elegimos **New/Branch**. En la ventana que nos saldrá elegimos la

carpeta que creamos en la rama. Toda la operación se hace desde la vista **SVN Repository Exploring** y desde la conexión a la raíz del repositorio.

Ahora los *CheckOut* que se realicen se harán desde las ramas. Para añadir los cambios de las ramas a la versión *trunk* se hará un **Merge/Reintegrate** desde la copia de trabajo del trunk con la copia del repositorio de la rama. Colocados encima de la copia de trabajo, pulsamos el botón derecho del ratón y elegimos **Team/Merge** pestaña **Reintegrate**. Mostramos la vista **Synchronize** con dichos cambios, aceptamos los cambios y se hace *commit* de la copia local del proyecto de trunk con la del repositorio. Para finalizar, haremos update para realizar la sincronización de ambos sitios.

2.3. 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 documente 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 sino 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 sucesos externos e internos.
 - **Criterios de validación:** documento sobre el límite de rendimientos, 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, etc.

- **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 2 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, el javadoc para JavaScript. Nosotros usaremos Javadoc.

2.3.1. 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 comentario para que genere la documentación son:

- **Comentarios en línea:** comienzan con los caracteres `"/"` y terminan en la misma línea.
- **Comentarios tipo C:** 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 dos partes: 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 <i>void</i> .
@throws	Descripción de la excepción que puede propagar. Habrá una etiqueta <i>throws</i> por cada tipo de excepción
@since	Número de versión de la que existe el método

- **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 del JDK y elegimos la carpeta bin.
- En los cuadros inferiores elegiremos el proyecto y las clases a 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 seleccionará la barra de navegación y el índice.

UF3. Introducción al diseño orientado a objetos

1. Introducción al UML

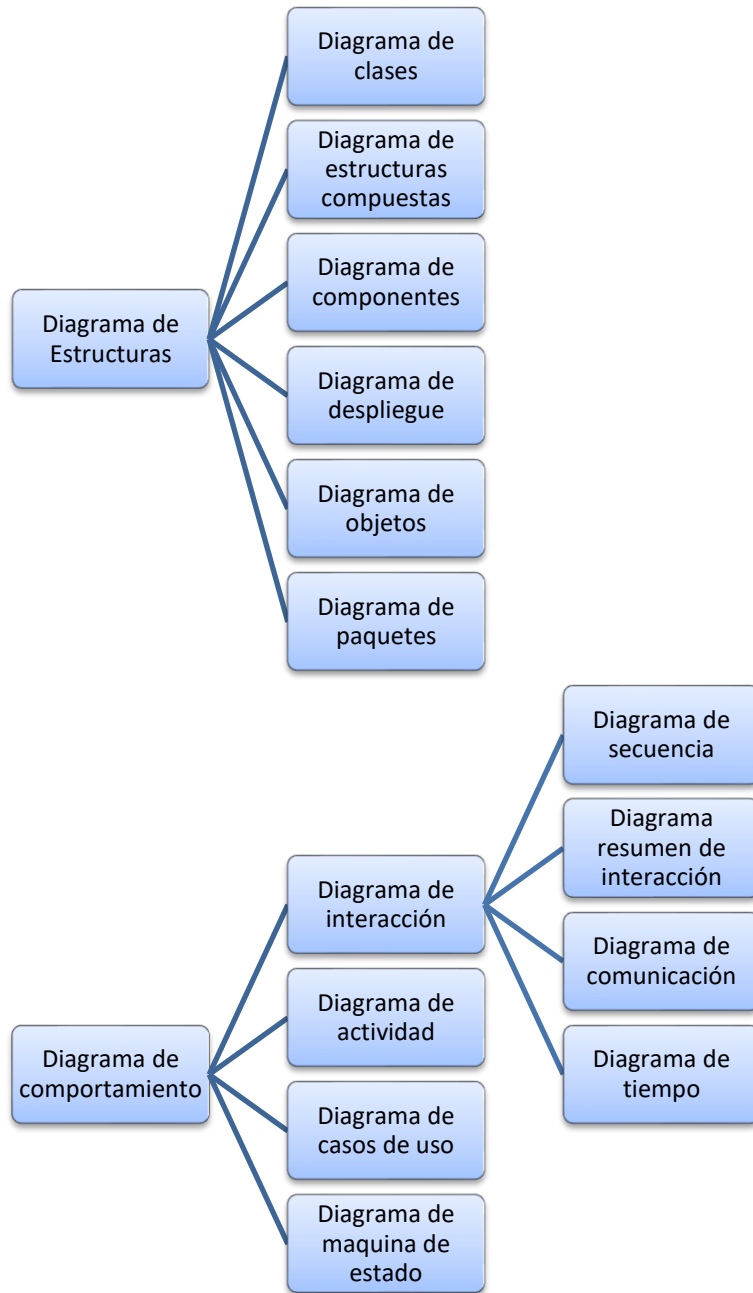
El lenguaje de modelado unificado (UML) es un lenguaje gráfico para visualizar, especificar y documentar cada una de las partes que comprende el desarrollo de software. Este lenguaje se puede utilizar para modelar sistemas de software, de hardware u organizaciones del mundo real. Para ello utiliza una 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 90 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 tres categorías: 6 tipos de diagramas representan la estructura estática de la aplicación o del sistema, 3 representan tipos generales de comportamiento y 4 representan diferentes aspectos de las iteraciones:

- **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 y diagrama de estado.
- **Diagramas de interacción:** todos los derivados del diagrama de comportamiento en general. Incluyen 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.



2. Elaboración de diagramas de clases

En esta primera parte de la tercera unidad formativa estudiaremos los conceptos básicos de la tecnología 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 interacciones 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.

2.1. Objetos

Es aquello que tiene atributos (propiedades más valores), comportamiento (acciones y reacciones a mensajes) e identidad (propiedad 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, donde se capturan sus movimientos. Se pretende conseguir una descripción formal. La abstracción es clave en el proceso y análisis del diseño OO, ya que podremos crear un conjunto de clases.
- **Encapsulación:** oculta detalles que no aporta a sus características esenciales, es decir, 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 no pueda ser intercambiada.
- **Concurrencia:** propiedad de un objeto que está activo de otro que no lo está.
- **Persistencia:** propiedad de un objeto a través de la cual su existencia trasciende en el tiempo y/o el espacio.

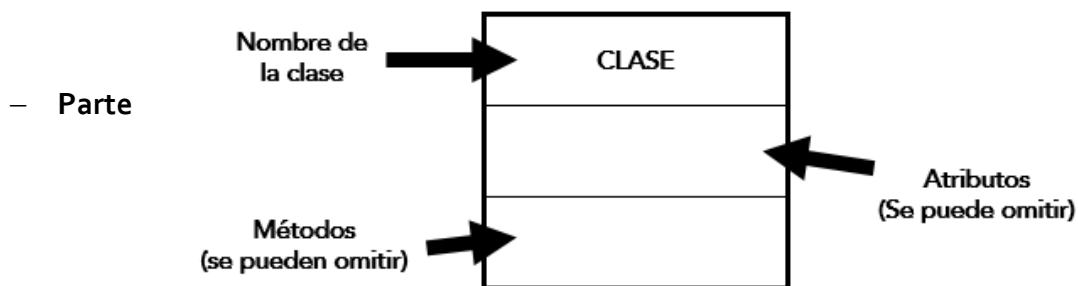
2.2. Diagramas de clases.

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, realización y dependencia.

2.2.1. 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 3 partes:



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 eliminar los métodos y atributos. La visibilidad de los métodos debe ser por defecto *public* y la de los atributos, *private*.

- **ATRIBUTOS**

Representan las propiedades de la clase. Se pueden representar solo con el nombre o también incluyendo su nombre, tipo y valor por defecto. Al crear los atributos indicaremos el tipo de dato. En UML los más básicos son: *Integer*, *String* y *Boolean*.

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

Se pueden distinguir los siguientes tipos:

- **public**: el atributo será público, visible tanto fuera como dentro de la clase. Se representa con el signo "+".
- **private**: el atributo solo será accesible dentro de la clase (solo sus métodos). Se representa con signo "-".
- **protected**: el atributo no será accesible desde fuera de la clase, pero sí por métodos de la clase y de subclases. Se representa con "#".
- **package**: el atributo será visible en las clases del mismo paquete. Se representa con la tilde "~".

- **MÉTODOS**

Es la implementación de un servicio de la clase que muestra un comportamiento común a todos los objetos. Los métodos nos definirán cómo interactuará la clase con su entorno. Los métodos, al igual que los atributos, pueden ser:

- **public**: el atributo será público, visible tanto fuera como dentro de la clase. Se representa con el signo "+".
- **private**: el atributo solo será accesible dentro de la clase (solo sus métodos). Se representa con el signo "-".
- **protected**: el atributo no será accesible desde fuera de la clase, pero sí por métodos de la clase y de subclases. Se representa con "#".
- **package**: el atributo será visible en las clases del mismo paquete. Se representa con la tilde "~".

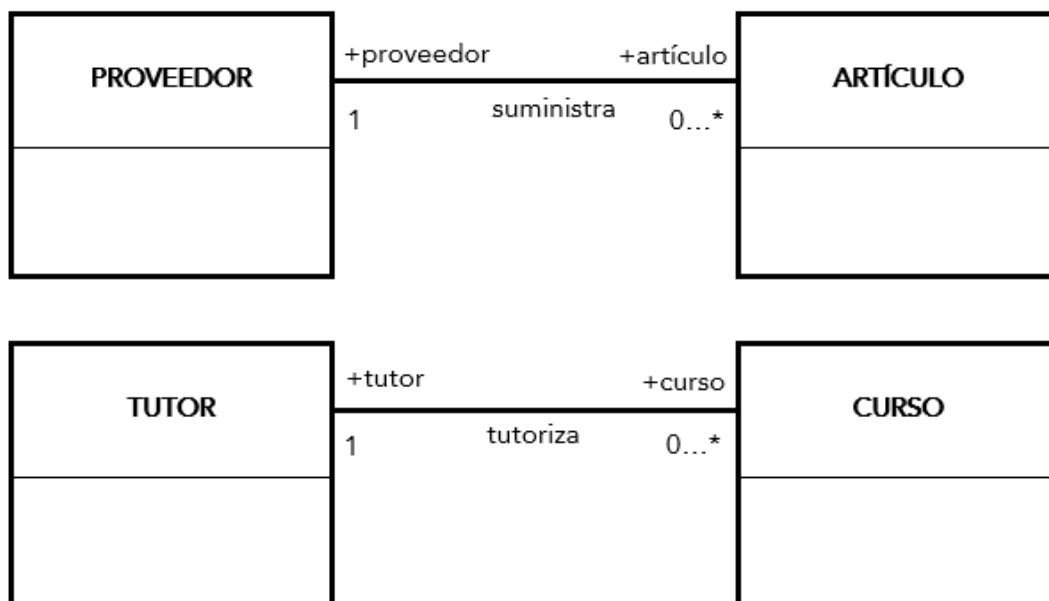
2.2.2. Relaciones

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



En una asociación **bidireccional**, cada una de las clases **Sí** se convierte en dos clases unidas por una asociación bidireccional, cada una de las clases tendrá un objeto que lo relacione. 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 *set* de objetos 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.

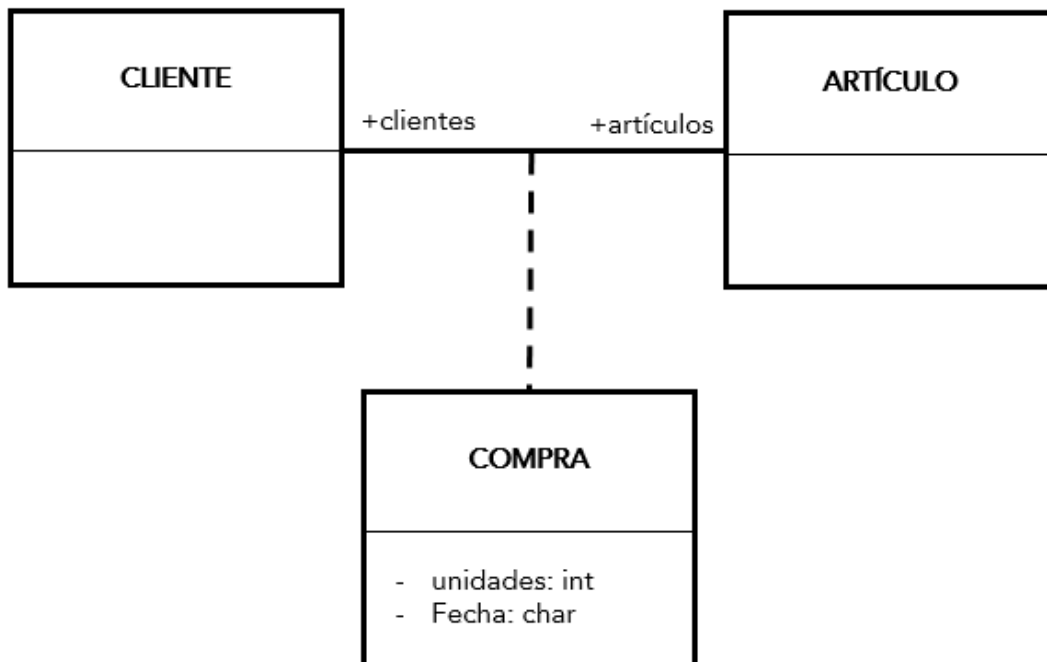


Algunas clases pueden asociarse consigo mismas creando así una asociación reflexiva. Estas asociaciones unen entre si instancias de una misma clase.




• CLASE ASOCIACIÓN

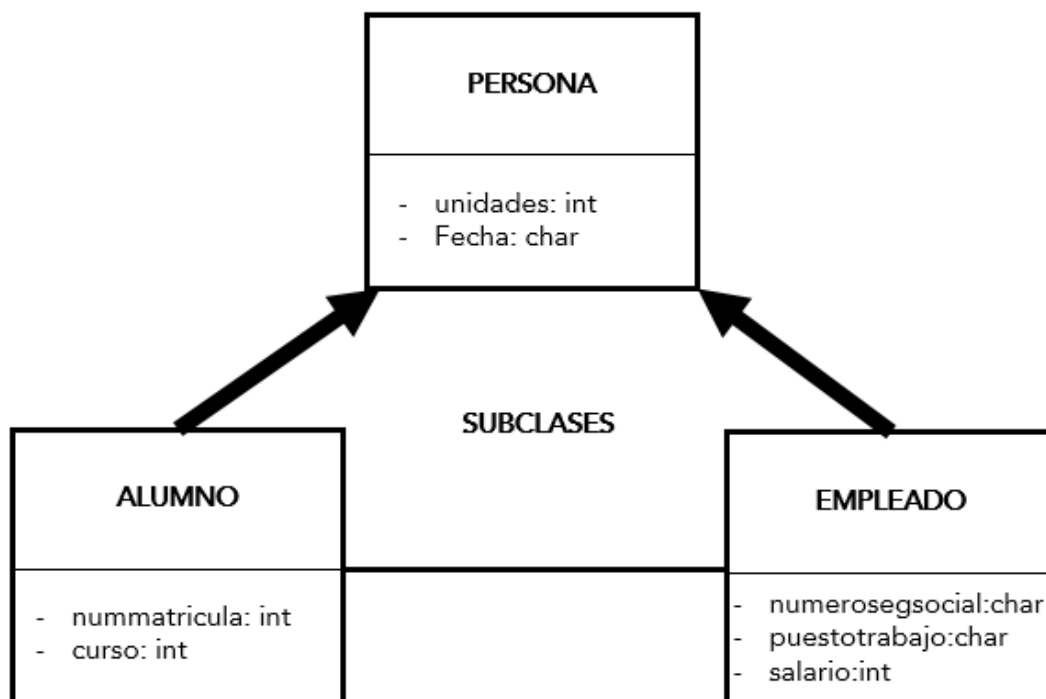
Hay asociaciones entre clases que podrán tener información necesaria para dicha relación, por lo que se creará una clase llamada clase asociación. Recibirá el estatus de clase y las instancias serán elementos de la asociación, al igual que podrán vincularse con otras clases.



- **HERENCIA (GENERALIZACIÓN y ESPECIALIZACIÓN)**

Podremos organizar las clases de forma jerárquica y, a través de 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.

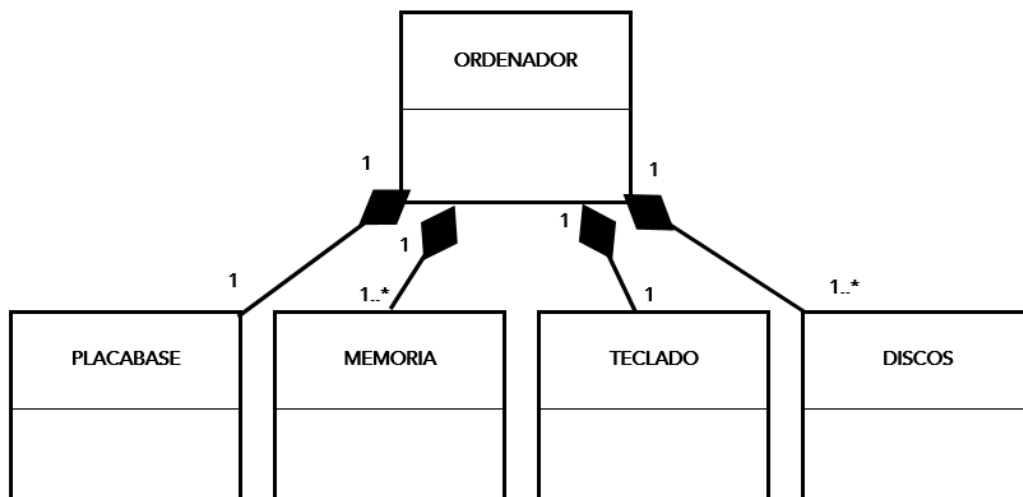
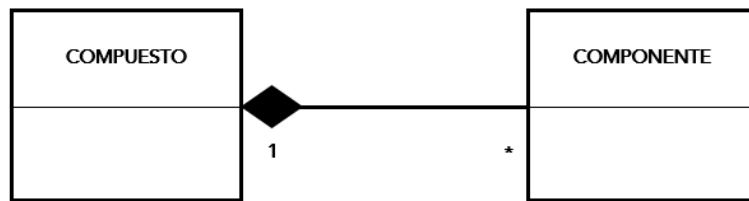


• COMPOSICIÓN

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

En la asociación fuerte, los objetos están constituidos por componentes y no pueden ser compartidos entre varios objetos compuestos. La cardinalidad será uno y la supresión del objeto comporta la supresión de los componentes.

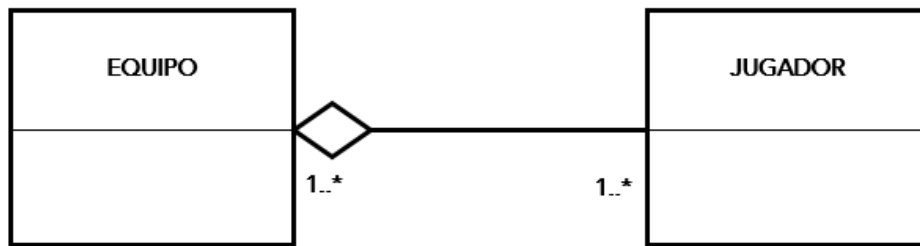
Se representa con una línea con un rombo lleno



• 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. Al comienzo del proyecto podremos usar solamente agregación y, después, determinar cuál de ellas son composición.

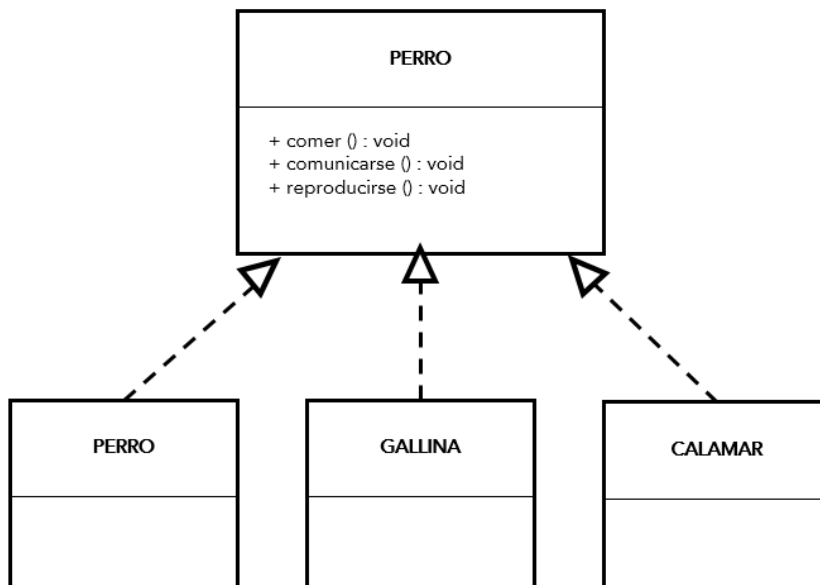
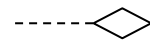
Su representación es una línea con un rombo vacío



• REALIZACIÓN

Será la relación de herencia que existe entre la clase interfaz y una subclase que implementa esa interfaz.

Se representa gráficamente con una flecha con línea discontinua



- **DEPENDENCIA**

Relación que se establece cuando una clase utiliza el contenido de otra clase.

Se representa con una flecha sin relleno discontinua que -----> irá desde la clase principal a la utilizada. Un cambio en la clase utilizada puede afectar a la principal, pero no al contrario.

2.3. Herramientas para el diseño de diagramas

2.3.1. ArgoUML

Es una herramienta líder de modelado UML de código abierto e incluye soporte para todos los diagramas UML. Podrá ejecutarse en cualquier plataforma Java y está disponible en 10 idiomas.

Podemos descargarla desde:

<http://argouml.tigris.org/>

Algunas características son las siguientes:

- Escrita en Java y disponible en cualquier plataforma Java.
- Capaz de crear los siguientes diagramas: casos de uso, clases, secuencia, colaboración, estado, actividades y despliegue.
- Compatible con UML 1.4.
- Soporte para creación de perfiles y distribución de modelos que hacen referencia a perfiles. Se entrega con referencia a perfiles: Java, C++ y UML 1.4.
- Proporciona generación de código para Java, C++, C#, SQL, PHP4 y PHP5.
- Proporciona generación de ficheros PNG, GIF, JPG, SVG, EPS desde los diagramas.
- Dispone de **críticas de diseño** que analizan el diseño y sugieren mejoras.

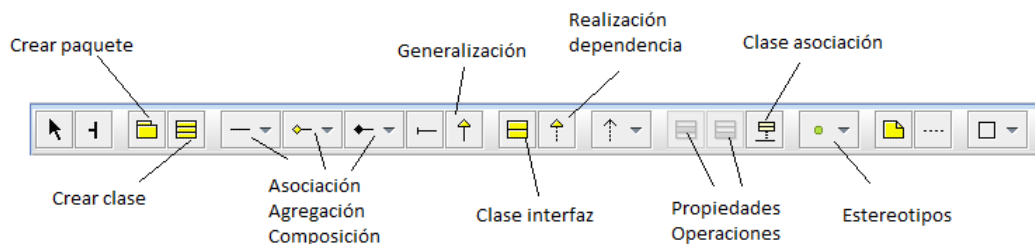
Una vez instalada, al abrirla nos aparecerá la ventana inicial con la barra de menú y la barra de herramientas.

Se distinguirán 4 paneles:

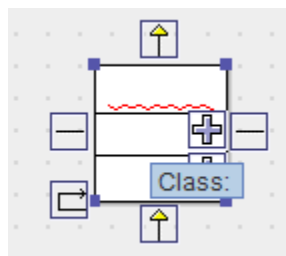
- **Panel de explorador:** situado a la izquierda, se verá en forma de árbol los diagramas del modelo y los elementos que lo componen.
- **Panel de diseño:** situado a la derecha, se colocarán los elementos que lo forman.
- **Panel de críticas:** situado a la parte inferior izquierda. Es de gran utilidad para los desarrolladores durante el diseño.
- **Panel de detalles y propiedades:** situado en la parte inferior derecha. Se configurarán las propiedades de los elementos y podremos visualizar el código que se va generando.

• CREACIÓN DE DIAGRAMAS DE CLASE

Cuando abrimos el programa ArgoUML por defecto nos aparecerá la ventana para crear el diseño de diagramas de clases. En la barra de botones tendremos las herramientas para poder insertar los elementos en el diagrama, solo tendremos que hacer clic en el elemento escogido. Si queremos crear una asociación, seleccionamos el tipo de asociación y hacemos clic desde la clase origen a la clase destino.



Podemos elegir el tipo de asociación de forma más rápida si pasamos el cursor por encima de la clase.



Al crear la clase, pondremos el nombre y añadiremos los atributos y operaciones, por lo que pulsaremos sobre el icono de añadir de la clase o pulsaremos los botones que correspondan en la barra.

Cuando hemos creado la clase, en propiedades podemos indicar la visibilidad (*pública, paquete, protegida o privada*) y el tipo de clase o modificadores. Los tipos de modificadores son:

- **Is Root:** para indicar que es una clase raíz, sin antecesores.
- **Is Leaf:** indica que la clase no puede ser en un futuro una especialización.
- **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.

Si lo que estamos creando no cumple con las reglas establecidas, aparecerán sugerencias en el apartado de críticas. Estas nos ayudarán a corregir fallos en el modelo y hacer que cumplamos las reglas. Podemos seleccionar individualmente cada crítica y en el panel de detalles y propiedades veremos desglosado el problema y la sugerencia para resolverla.

Si la clase posee una crítica aparecerá con el icono correspondiente en la parte superior de la clase.

El panel de críticas, por defecto, aparecerá *Por prioridad*, aunque también podremos elegir que se visualice *Por decisión*, *Por objetivo*, *Por causante*, *Por emisor* o *Por tipo de conocimiento*.

Cabe destacar que, si queremos eliminar un elemento del diseño, debemos suprimirlo desde la barra de botones pulsando *Borrar del modelo*. Si solo lo eliminamos con la tecla suprimir, el elemento seguiría permaneciendo en el modelo.

2.3.2. UML con Eclipse

Para poder hacer diagramas desde Eclipse tendremos que instalar el plugin UML2. Para ello, iremos al menú *Help* y seleccionamos *Eclipse Market Place*. En *Find* escribimos *UML2* y pulsamos en *Go*. Elegiremos el que trabaje con nuestra versión de Eclipse, en este caso *UML Designer (Eclipse Kepler Version)*. Pulsamos en *Install*, aceptamos las condiciones y comenzamos la instalación.

Después de realizar la instalación y de reiniciar el equipo, Eclipse observa que hay nuevas vistas. Nos interesa la vista *Modeling*.

Para crear un nuevo proyecto iremos al menú *File/New/UML Project*, escribimos el nombre y aceptamos las opciones asignadas por defecto para crear el proyecto.

Para crear un nuevo diagrama nos colocaremos sobre el proyecto, pulsamos el botón derecho del ratón y elegimos *Create Representation* y elegimos el diagrama a crear.

Nos fijaremos en los siguientes tipos:

- **Diagramas de comportamiento (UML Behavioral Modeling):** para diagramas de actividad, de estado, de casos de uso y de secuencia.
- **Diagramas de estructuras o estructurales (UML Structural Modeling):** para diagrama de clases, de componentes, de estructura compuesta, de implementación o despliegue, de objetos o de paquetes.

• CREACIÓN DE DIAGRAMAS DE CLASE

De la misma manera, para crear el diagrama de clases le damos en el botón derecho del ratón encima del proyecto, elegimos *Create Representation* y dentro de *UML Structural Modeling* seleccionamos *Class Diagram*. Pulsamos en siguiente, seleccionamos el modelo, escribimos el nombre y abrimos la vista de diseño. En este punto podremos observar cuatro zonas en el programa: vista del proyecto, vista de diseño, paleta de elementos y pestaña propiedades.

En la paleta seleccionamos el elemento que queremos insertar en el diagrama, clase, paquete, atributo, operación o tipo de relación. Solo tendremos que marcar y arrastrar. Esta acción también puede hacerse desde la barra flotante de botones.

Cuando creamos una clase, al hacer doble clic sobre ella nos muestra una ventana de creación de la clase, desde donde podremos teclear el nombre, elegir la visibilidad (**public, private, protected y package**) y marcar el tipo de clase (**Is Abstract, Is Leaf, Is Active**).

Si queremos añadir atributos y operaciones a la clase, lo haremos desde las pestañas **Attributes** u **Operations** y usaremos los botones para añadirlos, eliminarlos o cambiarlos de posición.

Al añadir los atributos indicamos el nombre, la visibilidad, las propiedades y el tipo.

Propiedad	Valor por defecto	Descripción
Is Read Only	False	Si es true, será solo lectura y no podrá cambiarse el valor
Is Static	False	Si es true, las instancias comparten valor para el atributo
Is Leaf	False	Si es true, no está definido para que se redefina en los tipos derivados

Is Derived	False	Si es true, se calcula a partir de otros atributos, es un atributo calculado
Is Ordered	False	Si es true, la colección forma una lista secuencial y ordenada
Es Unique	False	Si es true, no hay valores duplicados

Cuando asignamos un tipo de dato a los atributos, nos aparece una ventana para elegir el tipo de dato. Para elegir los tipos de datos *int*, *float*, *String* y *Date* lo haremos desde la lista ***PrimitiveType***<***Primitive Type***>.

Para crear una operación escribimos el nombre, la visibilidad, marcamos el tipo y añadimos los parámetros si los tuviera. Para ello pulsamos en *Parameters* y el botón añadir (+). En la nueva ventana, escribimos el nombre del parámetro, elegimos el tipo de dato y se marca el tipo de parámetro (**in** para indicar que es de entrada, **inout** de entrada-salida, **out** de salida y **return** si devuelve un valor de retorno).

Podemos cambiar las propiedades de la clase. Si no se muestra el cuadro abrimos el menú ***Windows/Show view/Properties***. Podremos ver, añadir y cambiar los atributos, las operaciones, las relaciones de la clase, la apariencia y la semántica.

3. 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 comportamientos.

3.1. 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 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 e interacción	Cooperación entre otros diagramas de interacción

3.2. 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.

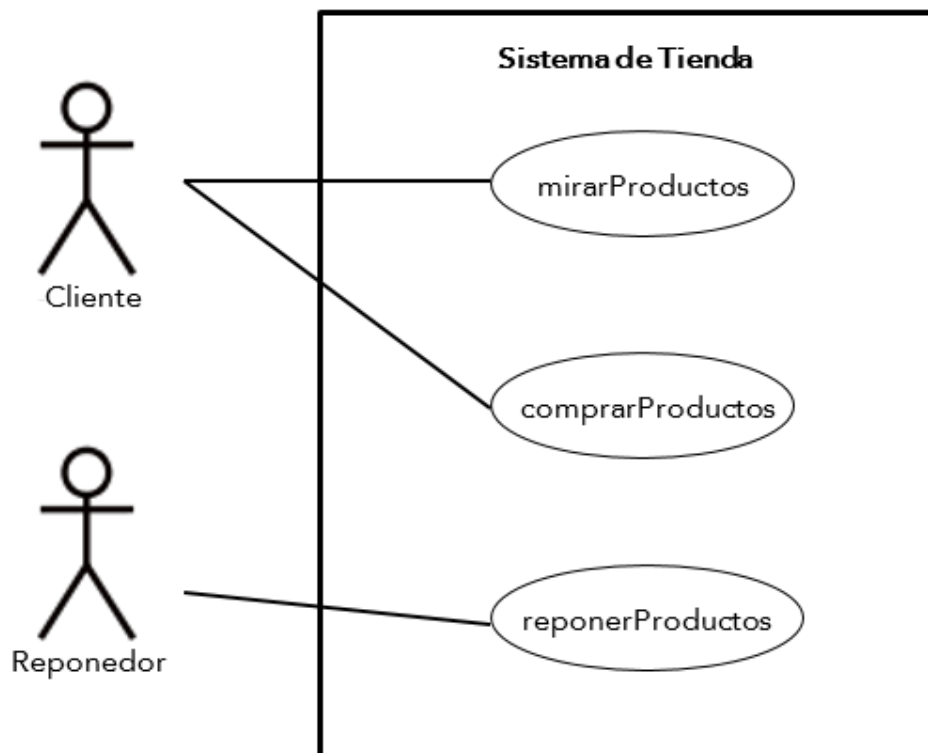
Se usará un lenguaje sencillo y deberá describir todas las formas de utilizar el sistema, por lo que define todo el comportamiento de este.

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

3.2.1. 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. Aunque se representa con un monigote y un nombre debajo, no tiene por qué ser una persona.
- **Casos de uso:** representa una unidad funcional del sistema que realizará una orden de algún agente externo, tanto de un agente como de otro caso de uso. 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 entre actores y casos de uso representada con una línea continua.
- También podemos tener un rectángulo que delimite el sistema.



Los casos de uso siempre 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.

3.2.2. 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.

3.2.3. Identificar casos de uso


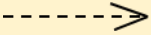

Para identificarlos será necesario entender el funcionamiento del sistema y lo que quiere hacer. Para ello tendremos que buscar los actores que participan y saber cómo lo usarán.

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.
- **Postcondiciones.** Las condiciones que se deberán cumplir cuando finalice el caso de uso.
- **Alternativas.** Errores o excepciones.

3.2.4. 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>>	Especifica que el comportamiento de un caso de uso es diferente según las circunstancias. Apuntará a la relación de caso de uso que extenderá	
Generalización de casos de uso	Generalización entre clases. El caso de uso hijo hereda el comportamiento y significado del padre.	

Inclusión <<include>> o <<uses>>

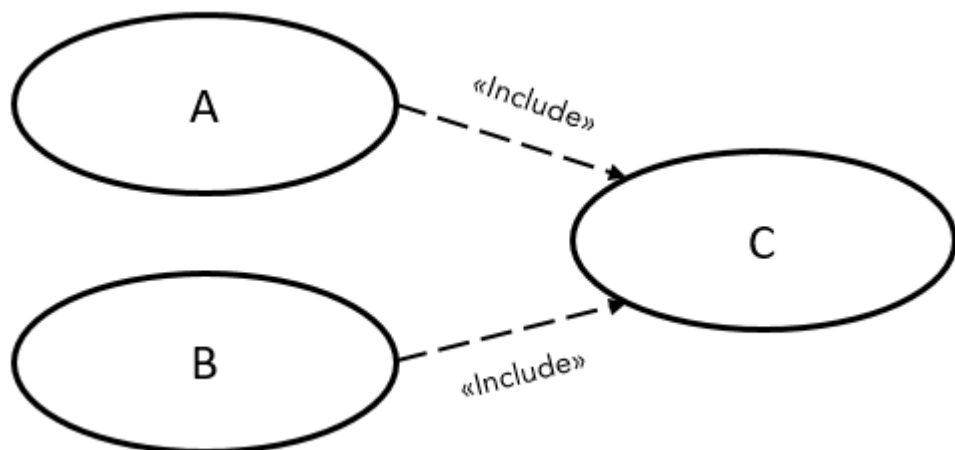
Permite que un caso de uso base incluya el comportamiento de otro caso de uso

<<include>>
----->

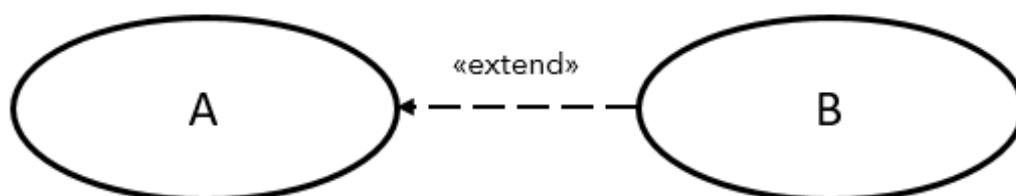
Suele ocurrir que las clases <<extend>> e <<include>> se confunden.

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 X e Y que poseen una serie de pasos en común, podrán ponerse esos pasos en un mismo caso de uso Z, y X e Y lo incluyen para usarlo.

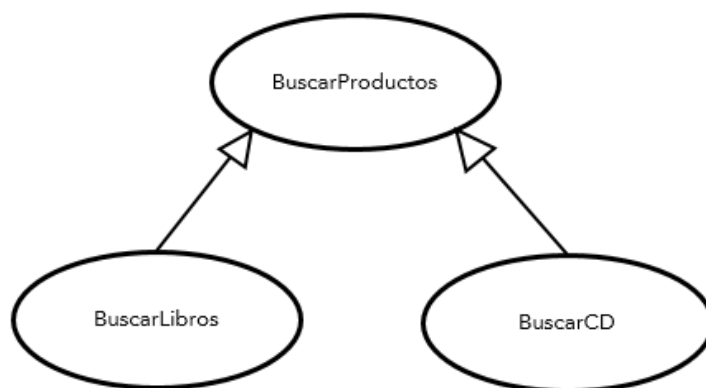


- **Extend:** se usará esta relación cuando un caso de uso extiende la funcionalidad de otro caso de uso. Si tenemos un caso de uso Y extenderá la funcionalidad del caso de uso X añadiendo algunos pasos.



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

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.



3.2.5. Herramientas para la creación de diagramas

- **ArgoUML**

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

Cuando abrimos ArgoUML, nos aparece un diagrama de clase y un diagrama de casos de uso. Si hacemos clic sobre ellos, aparecerá el panel de edición con iconos para elaborarlos.

Para crear el límite del sistema usamos la herramienta límites que se muestra en la barra y se arrastra al diagrama.

Para crear la **relación de asociación** pulsamos en el botón *Asociación nueva* en la barra de botones y solo tendremos que arrastrar del actor al caso de uso o viceversa. También se puede realizar pulsando en el actor y arrastrando desde el icono que se muestra a derecha e izquierda hasta el caso de uso.

Para **borrar** algún elemento tendremos que seleccionarlo, hacer clic derecho con el ratón y pulsar la opción *Borrar del modelo*. Se podrá crear varios diagramas en un mismo modelo.

Si queremos crear un **punto de extensión**, crearemos primero la relación extiende sobre el caso de uso a ser extendido.

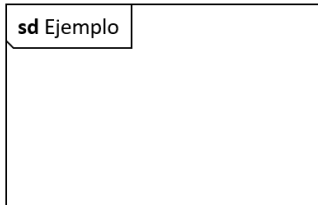

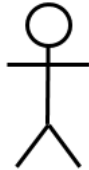
Para obtener una imagen del diagrama, pulsamos sobre él con el botón derecho del ratón y seleccionamos la opción *Copiar diagrama al portapapeles como imagen*.





3.3. 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. Partimos de los casos de uso elaborados en la etapa de análisis. El diagrama tendrá dos dimensiones: la dimensión vertical, que representa el tiempo; y la dimensión horizontal, que representa los roles de la interacción.

Cada rol 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 horizontales 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:

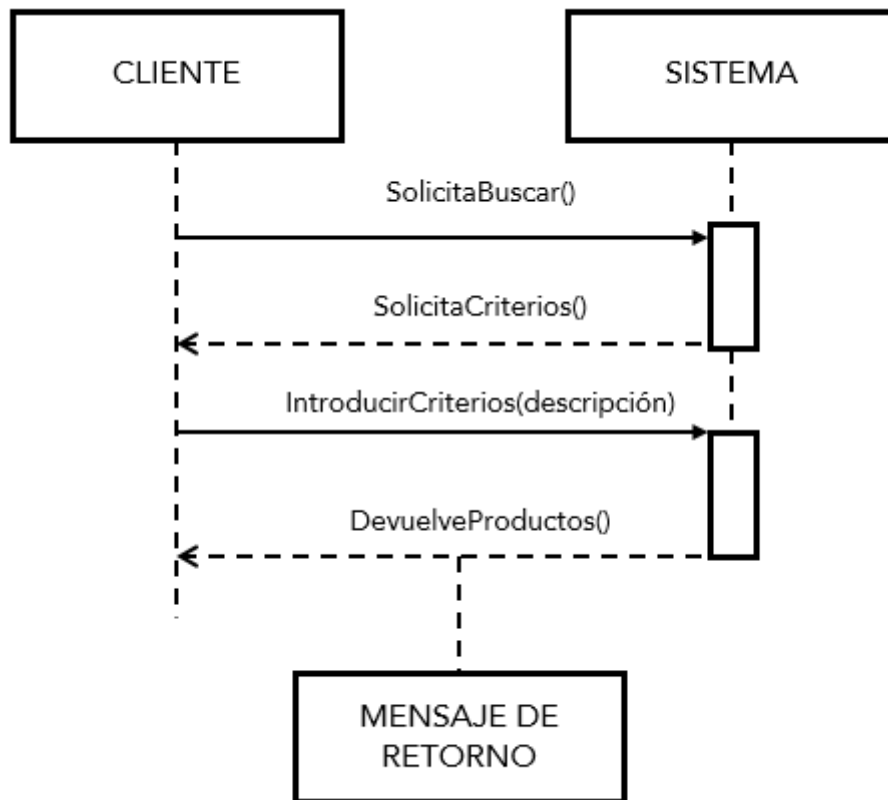
SÍMBOLO	FUNCIÓN	NOTACIÓN
Marco	Da borde visual al diagrama de secuencia. A la izquierda del marco se escribe la etiqueta sd seguida de un nombre	
Línea de vida	Representa un participante durante la interacción	
Actor	Representa el papel desempeñado por un usuario	

Mensaje	Mensaje síncrono	Mensaje() 
	Mensaje asíncrono	Mensaje() 
	Mensaje de retorno	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 dibujará con una flecha que irá dirigida desde el participante que lo envía hasta el que lo ejecuta. Tendrá 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 que el receptor finalice la ejecución del mensaje. Su utilización la podemos ver en sistemas multi-hilos donde se producen procesos concurrentes.
- **Mensaje de retorno:** representa mensaje de confirmación. Su uso es opcional.

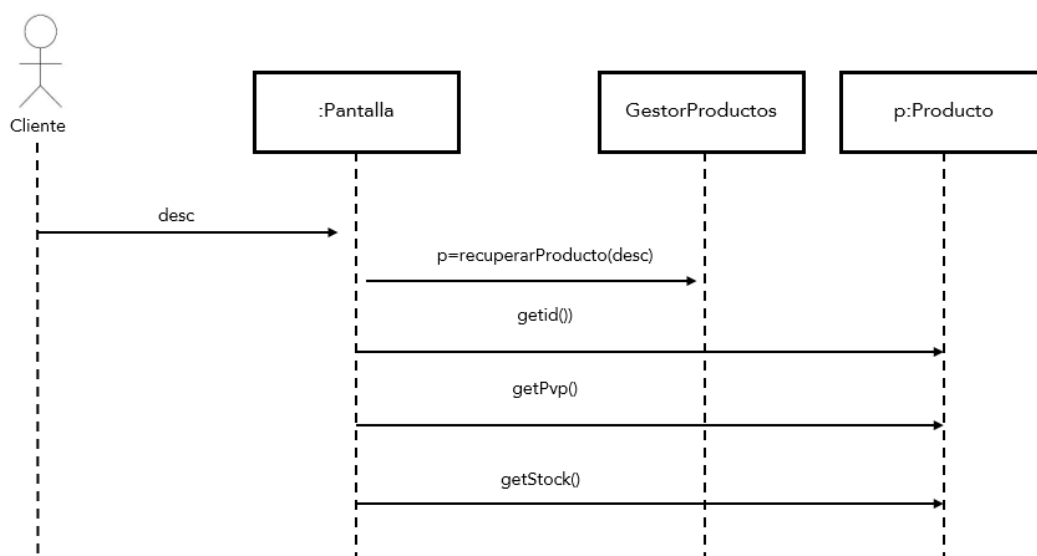
En estos diagramas hemos visto solamente eventos que fluyen desde el cliente hacia el sistema. En la dimensión horizontal solo se incluían esos dos roles. Dado que trabajamos en un sistema orientado a objetos en el que tenemos varias clases y varios objetos que se comunican unos con otros, las clases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



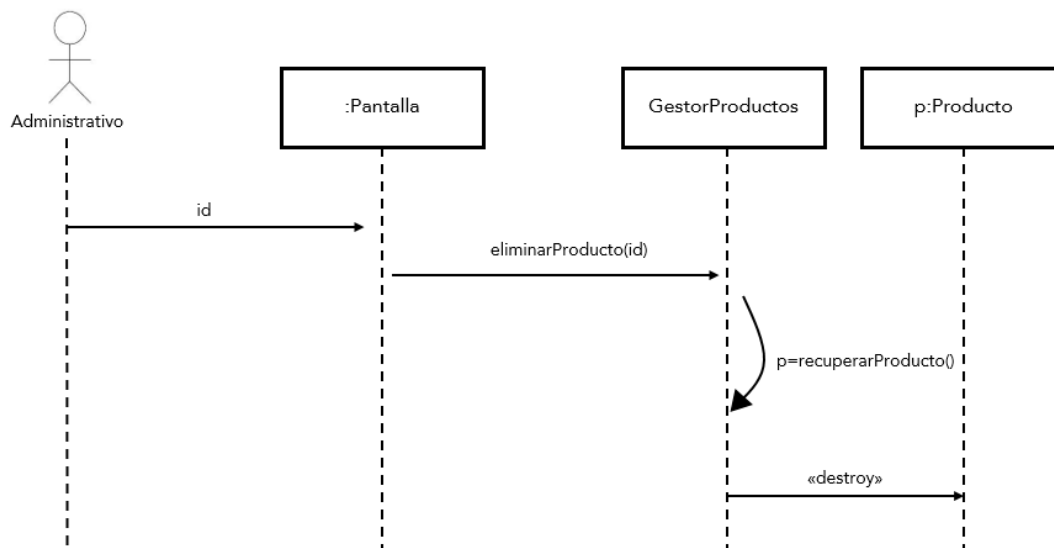
• CREACIÓN Y DESTRUCCIÓN DE OBJETOS

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.

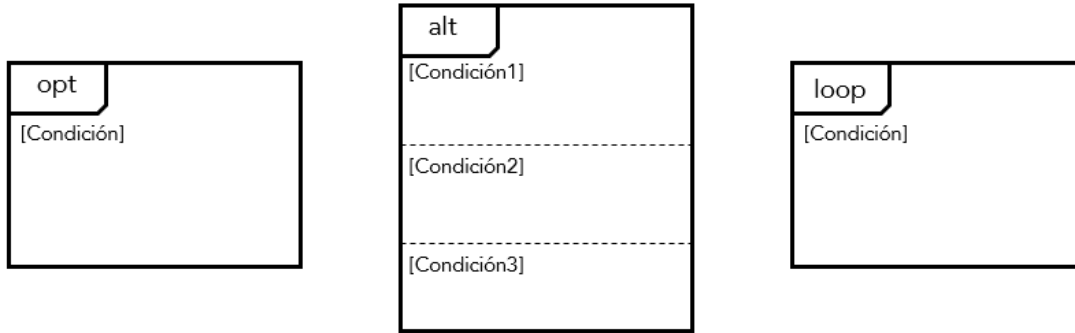
• ALTERNATIVAS Y BUCLES

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 varios tipos de 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 a 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.



3.4. Diagramas de colaboración. Objetos, mensajes.

En el diagrama de colaboración, 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. De hecho, hay herramientas que permiten convertir un diagrama de secuencia en un diagrama de colaboración y viceversa.

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.

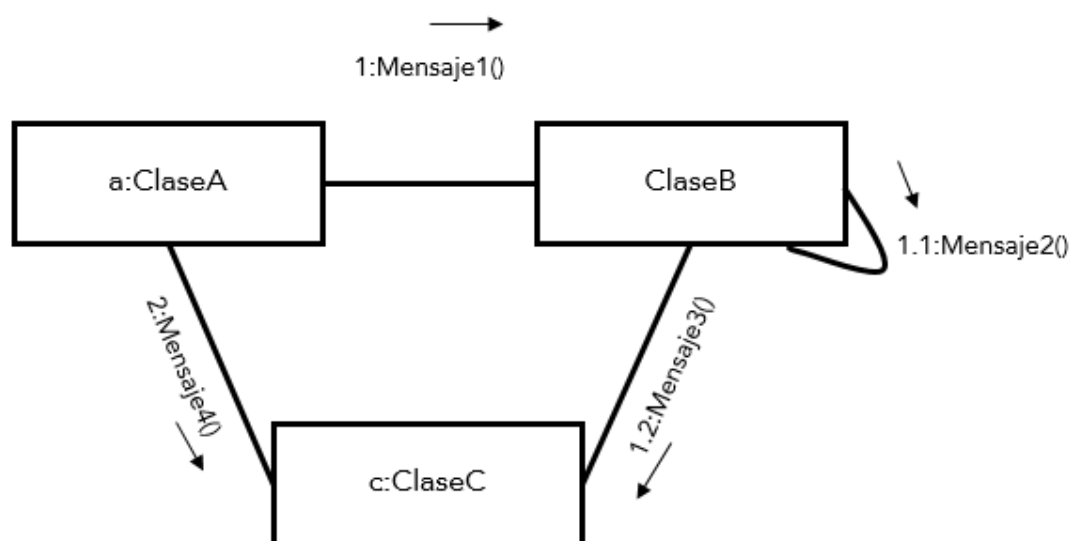
El diagrama de colaboración posee los siguientes **elementos**:

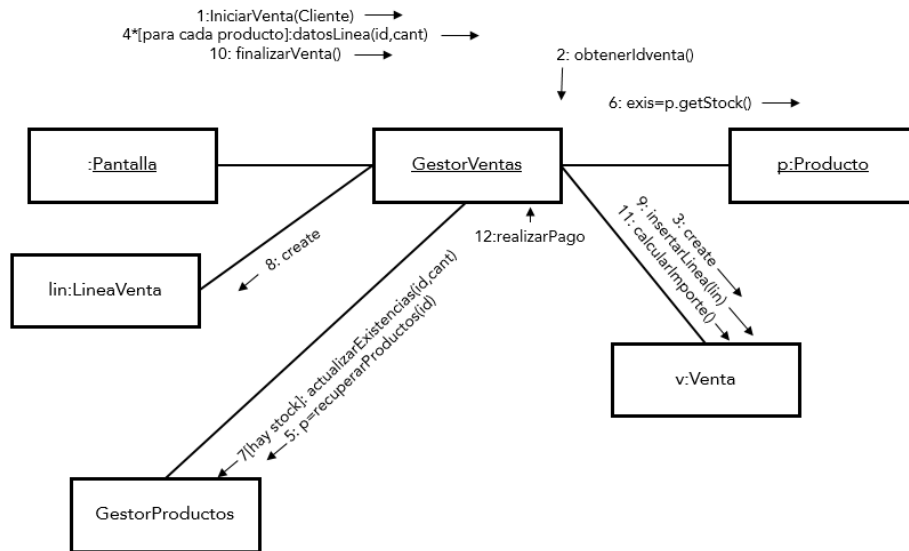
SÍMBOLO	FUNCIÓN	NOTACIÓN
Objetos o roles	Se representa con un rectángulo que contiene el nombre y la clase del objeto en el siguiente formato objeto:Clase.	
Enlaces	Arcos del grafo que conecta ambos objetos. Se podrán mostrar muchos mensajes en un mismo enlace, pero cada uno con un número de secuencia único	
Mensajes	Se representa mediante una flecha dirigida con un nombre y un número de secuencia	

Número de secuencia	Indica el orden de un mensaje dentro de la iteración (repetición). Comenzará con el 1 y se incrementará conforme se envíen mensajes. El primer mensaje no lleva número de secuencia.
Iteración	Se representa colocando un * después del número de secuencia y una condición encerrada entre corchetes.
Alternativa	Se indican con condiciones entre corchetes. Los caminos alternativos tendrán el mismo número de secuencia, seguido del número de subsecuencia.
Anidamiento	Se puede mostrar el anidamiento de mensajes con números de secuencia y subsecuencia.

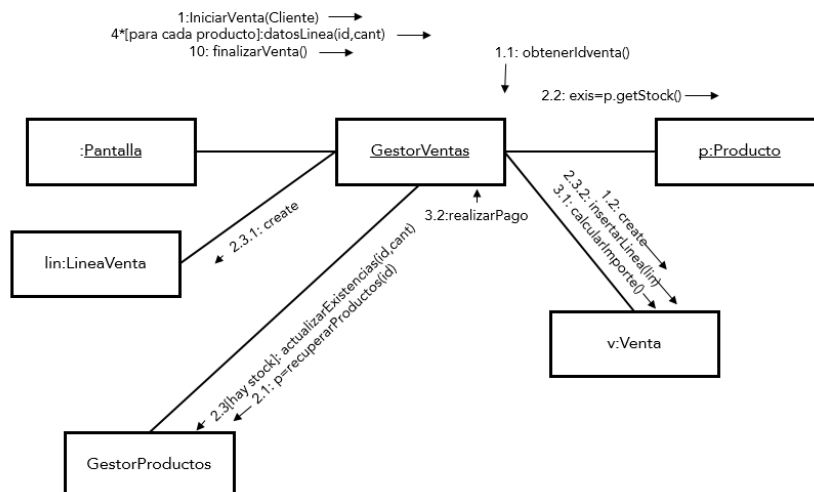
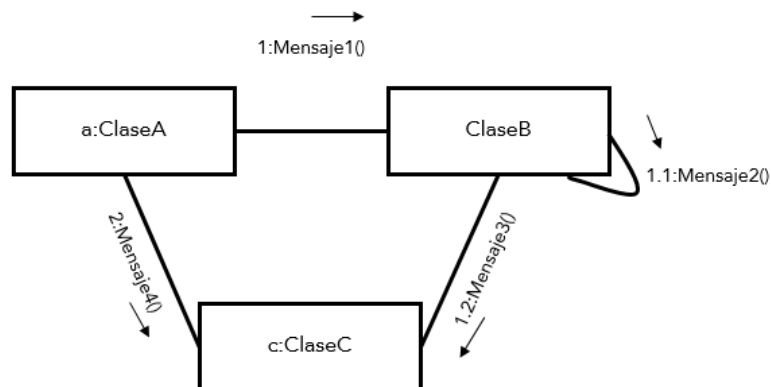
Para numerar los mensajes se puede usar varios esquemas:

- **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.



Bibliografía

RAMOS MARTÍN, Alicia (2014). *Entornos de desarrollo*. Madrid: Editorial Garceta.

```
function updatePhotoDescription() {  
    if (descriptions.length > (page * 9) + (currentImageSubsting() - 1)) {  
        document.getElementById("bigImageDesc").innerHTML = descriptions[page * 9 + (currentImageSubsting() - 1)];  
    }  
}  
  
function updateAllImages() {  
    var i = 1;  
    while (i < 10) {  
        var elementId = "foto" + i;  
        var elementIdBig = "bigImage" + i;  
        if (page * 9 + i - 1 < photos.length) {  
            document.getElementById(elementId).src = "images/min/" + photos[page * 9 + i - 1].src;  
            document.getElementById(elementIdBig).src = "images/max/" + photos[page * 9 + i - 1].src;  
        } else {  
            document.getElementById(elementId).src = "images/min/default.jpg";  
            document.getElementById(elementIdBig).src = "images/max/default.jpg";  
        }  
        i++;  
    }  
}
```