

## UNIDAD 1. INTRODUCCIÓN Y CONCEPTOS

### Hardware - Software

**Partes o elementos** de un dispositivo informático:

- **Hardware (HW):** Conjunto de componentes físicos o materiales que componen un dispositivo o PC.
- **Software (SW):** Conjunto de programas y aplicaciones que residen y se ejecutan sobre el HW.

Ambos elementos **se complementan** para que un dispositivo pueda funcionar.

### Tipos de Software

Según la función:

- **Sistemas operativos (S.O.):** Software base que ha de estar instalado y configurado en un ordenador para que las aplicaciones puedan ejecutarse y funcionar.
- **Software de programación:** Herramientas para desarrollar otros programas informáticos.
- **Aplicaciones:** Conjunto de programas con finalidad más o menos concreta.

### Desarrollo de software

Es el **proceso** desde que se detecta una necesidad o se concibe una idea, hasta que se dispone de una herramienta software en funcionamiento que la resuelve, o más bien, hasta que esta herramienta o solución deja de utilizarse:

1. **Inicio:** Se detecta una necesidad o se concibe una idea.
2. **Objetivo:** Desarrollar e implantar una solución software, que resuelva la necesidad y cumpla los acuerdos previstos (tiempo, forma y coste).
3. **Final:** La herramienta o solución deja de utilizarse y mantenerse.

### Fases clásicas de un proyecto de desarrollo de SW

#### 1.- Análisis:

Es la **primera fase** de un proyecto. **Las demás dependen de su precisión.**

Se recopilan los requerimientos y se analizan en profundidad.

¿Qué hay que hacer en esta fase? **Objetivos:**

- ❖ Definir los **requerimientos**, funcionales o no funcionales.
- ❖ Definir una **estrategia**.
- ❖ Definir una **planificación** de tareas e hitos.

La **comunicación e implicación** de todo el personal implicado en el proyecto es un factor clave para definir **¿qué debe realizar la solución?**

#### 2.- Diseño:

Se definen en detalle las funcionalidades necesarias para cubrir los requerimientos.

**¿Cómo se van a resolver los requerimientos?** Dependerá de la naturaleza de los requerimientos, de la estrategia y de las circunstancias del proyecto.

¿Qué hay que hacer en esta fase?

- **Análisis técnico de los requerimientos:** Mediante gráficos, diagramas, algoritmos, textos, etc.
- **Decisiones para organizar y cuantificar recursos:** Personas, infraestructura, Entornos de desarrollo, lenguaje/s de programación, gestor de bases de datos.

#### 3.- Desarrollo o codificación:

Se codifica la solución en base al diseño realizado.

¿Qué hay que hacer en esta fase? **Objetivos:**

**Codificar** mediante uno o varios lenguajes de programación.

El código puede pasar por diferentes **estados**:

- ✚ **Código fuente**: Lo generan los **programadores**. Habitualmente mediante el uso de lenguajes de programación de **alto nivel**.
- ✚ **Código objeto**: Código binario resultado de **compilar** el código fuente. **Código intermedio**, no es inteligible por las personas, y todavía no es ejecutable por un sistema operativo.
- ✚ **Código ejecutable o código máquina**: Código binario resultante de **enlazar o linkar** el código objeto con rutinas y bibliotecas utilizadas. El sistema operativo lo podrá cargar en memoria RAM y **ejecutarlo directamente**.

#### 4.- Pruebas:

Se verifican y testean de los resultados del desarrollo de la solución.

Conforme se va disponiendo de elementos desarrollados, es crucial realizar pruebas.

¿Qué hay que hacer en esta fase? **Objetivos**:

Se distinguen **múltiples tipos** de pruebas:

1. **Unitarias**: Testean pequeños elementos aislados.
2. **Pruebas de integración**: Testea que los elementos se integran correctamente.
3. **Pruebas sistema**: Testea el funcionamiento de un proceso completo.

El resultado de las pruebas puede provocar que algunos elementos retrocedan, incluso a la fase de análisis.

#### 5.- Implantación:

Etapas de puesta en marcha de la solución. Instalaciones. Capacitación de usuarios.

Cuando una solución alcanza cierta fiabilidad, se prepara el escenario donde se usará.

¿Qué hay que hacer en esta fase?:

- ❖ **Instalación** de la solución en la infraestructura donde se ejecutará.
- ❖ **Configuración** de la solución.
- ❖ **Capacitación** de usuarios.
- ❖ **Explotación/producción**.

#### 6.- Mantenimiento y evolución:

La naturaleza del software es cambiante, necesitará actualizarse, adaptarse y evolucionar (puede durar toda la vida).

¿Qué hay que hacer en esta fase?:

El **mantenimiento** de una solución software puede durar toda la vida del producto.

Existen varios tipos de mantenimiento:

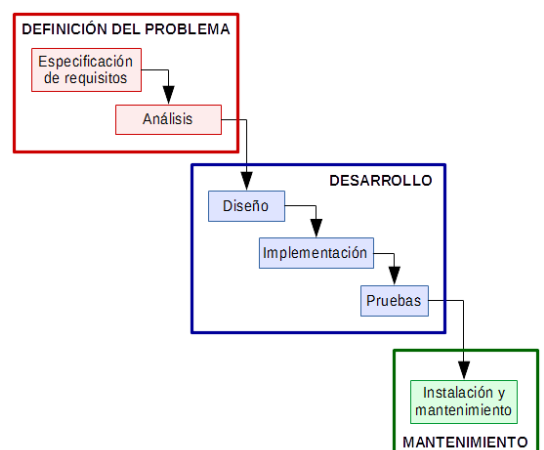
- **Correctivo**: Corrección de errores y fallos de funcionamiento.
- **Adaptativo**: Modificaciones y actualizaciones de la funcionalidad existente.
- **Evolutivo**: Nuevas funcionalidades, nuevas versiones.

#### Modelos del ciclo de vida del software

En función de la secuencia que se establezca entre las fases o etapas y de las reglas que establezcan entre ellas, podemos encontrar diferentes **modelos**:

##### En cascada:

- Es el modelo más clásico y menos flexible.
- Una variante muy interesante y flexible es el **modelo en cascada con retroalimentación**.



Modelos más actuales, que tienen en cuenta la **naturaleza altamente cambiante del SW**, son los **modelos evolutivos**:

## En espiral:

- Es un modelo flexible.
- El software se va construyendo **repetidamente** en forma de **versiones mejoradas**.
- Se apoya en las fases clásicas.
- Puede ser **complejo** en su implementación.
- Permite **reducir los riesgos** en cada versión.

Existen más modelos (modelo en V, modelo por prototipos, etc.), y más que surgirán...

No existen modelos “buenos” y “malos”, dependerá de las circunstancias del proyecto.

Es habitual utilizar una combinación de modelos, en mayor o menor medida.

## Documentación

La documentación es una labor muy importante.

Se puede considerar como una fase clásica más.

Desde la antigüedad, la documentación ha permitido **universalizar** los detalles de un proyecto.

## Mejora e integración continua

### Mejora continua:

Consiste en repetir este procedimiento de forma permanente:

- **Analizar procesos** y estudiarlos en detalle.
- **Realizar adecuaciones** para mejorarlos y minimizar los errores.

### Integración continua:

- Consiste en **integrar las mejoras** de varios equipos o desarrolladores (contribuidores) en un único proyecto de software, de forma periódica.

## Gestión ágil de proyectos

Consiste en realizar **entregas** de forma **continua e iterativa**.

Las **personas** y su interacción se posicionan como **núcleo** del proyecto. Por encima de las herramientas y los procedimientos.

La estrecha **colaboración y comunicación** entre los componentes del proyecto es una necesidad primordial para obtener éxito.

La **unidad básica** de ejecución es la **iteración**. En cada iteración se debe marcar una serie de objetivos, de forma que **cada iteración va aportando valor a la solución**.

Existen **definiciones** específicas de **pautas de trabajo** basadas en la gestión ágil de proyectos, como:

- ❖ Scrum.
- ❖ Kanban.

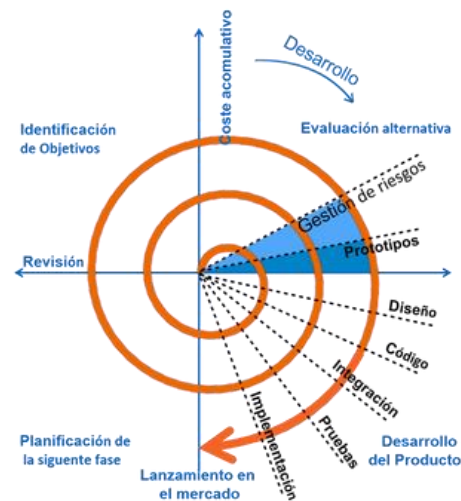
También existen **herramientas** orientadas a este tipo de gestión:

- 🚧 Trello.
- 🚧 Jira.

## Lenguajes de programación

Son **idiomas** creados artificialmente para que una máquina los pueda ejecutar.

Formados por un conjunto de **símbolos** y unas **normas** aplicables.



### Clasificación

**Según la proximidad con el lenguaje humano:**

**Alto nivel:** Próximos al lenguaje humano. Los usados habitualmente.

**Bajo nivel:** Más próximos al funcionamiento interno de una máquina:

🔧 Ensamblador.

🔧 Código máquina.

**Según la técnica de programación utilizada:**

**Estructurados:** Conjunto ordenado de sentencias con inicio y final (Pascal, C, Fortran, Cobol, BASIC, etc.).

**Modulares:** Permiten dividir un código en fragmentos o módulos reutilizables (Pascal, C, Cobol, RPG, etc.).

**Orientados a objetos:** Los programas son un conjunto de objetos que interactúan entre sí. Es el paradigma más extendido (C++, C#, Delphi, Java, JavaScript, PHP, etc.).

**Visuales:** Permiten generar programas mediante la manipulación de elementos gráficos (Scratch, Bolt, MakeCode, etc.).

### Entornos de desarrollo

En inglés **Integrated Development Environment (IDE)**.

Aplicación con un conjunto de **herramientas para facilitar el desarrollo de software**.

**Características:**

1. Diseñados para maximizar la productividad.
2. Ofrecen utilidades para la creación y depuración de software.
3. Facilitan la reutilización de componentes.
4. Ayudan a reducir la configuración.
5. Algunos soportan múltiples lenguajes de programación.
6. Permiten personalización mediante la activación o desactivación de herramientas y componentes.

**Ventajas:**

Facilita el aprendizaje.

Dispone de mecanismos de ayuda avanzada, como auto completar.

Proporciona mensajes de alerta o error durante todo el proceso.

Incrementa la productividad del desarrollador mediante herramientas y plantillas.

**Inconvenientes:**

Suelen consumir muchos recursos.

Generan dependencia, cierta cautividad.

**Ejemplos:**

**Visual Studio:**

- Desarrollado por Microsoft y comercializado bajo licencia privativa.
- Es compatible con lenguajes como C++, C#, Visual Basic .NET, F#, Java, Python, Ruby, PHP, etc.

**Eclipse:**

- Actualmente lo gestiona una fundación que fomenta el código abierto.
- Es multiplataforma (Windows, Linux y Mac).
- Es posible desarrollar usando Java, C, C++, JSP, perl, Python, Ruby, PHP, etc.

**NetBeans:**

- Desarrollado por Apache (Oracle), basado en código abierto.
- Es el “oficial” de Java, también podemos desarrollar en otros lenguajes como PHP, C, C++, etc.

## UNIDAD 2. UML COMPORTAMIENTO: CASOS DE USO

### INTRODUCCIÓN

#### ¿Qué es UML?

**Lenguaje de modelado unificado** (Unified Modeling Language).

Según la RAE....

- **Modelar:** Configurar o conformar algo.
- **Unificar:** Hacer de muchas cosas, una o un todo.

Es el lenguaje de modelado de sistemas de software más conocido y utilizado en la actualidad. Es un estándar desde 2004.

Es un **lenguaje gráfico** utilizado para analizar, especificar, diseñar, visualizar, construir y documentar un sistema, servicio, aplicación, desarrollo o solución.

**No es un lenguaje de programación.**

Describe objetos, procesos, estados y relaciones en un sistema.

**Su elaboración se realiza durante las fases de análisis y/o diseño y su uso se extiende durante el resto del proyecto**, como casi toda la documentación importante.

Se basa en el uso de **diagramas**.

#### ¿Qué es un diagrama?

Una **representación de la realidad**.

En nuestro caso, dicha realidad podría ser un proceso, un sistema o las relaciones existentes entre ciertos objetos de un sistema.

Se trata de una **representación gráfica**, habitualmente mediante **dibujos geométricos**.

Es necesario conocer el significado de las formas y demás caracterización utilizada para poder elaborar o interpretar un diagrama.

#### Tipos de diagramas UML

En UML encontramos 2 tipologías principales de diagramas:

**Diagramas de comportamiento:** Definen el comportamiento dinámico de un sistema (video).

1. Diagramas de casos de uso.
2. Diagramas de actividades.
3. Diagramas de secuencias.
4. Diagramas de tiempos.

**Diagramas estructurales:** Sirven para definir la estructura estática de un sistema (foto).

1. Diagramas de clases.
2. Diagramas de componentes.

### ACTORES Y CASOS DE USO

#### ¿Qué es un diagrama de casos de uso?

Es un tipo de **diagrama de comportamiento**, por lo tanto, define los aspectos dinámicos de un sistema.

Dentro del ciclo de vida del SW, los diagramas de casos de uso formarían parte de la **fase de análisis**.

Se recomienda que sean de los primeros diagramas a elaborar.

Facilitan la especificación de **requerimientos**.

**Representan un sistema** desde los distintos puntos de vista de los **distintos perfiles de usuario que interactuarán** con él.

Un diagrama de casos de uso describe un sistema, teniendo en cuenta sus **distintas variedades de interacción**.

Cada una de estas variedades, equivaldría a un **caso de uso**.

Los casos de uso son ideas sencillas y prácticas que no requieren habilidades ni conocimientos tecnológicos avanzados para ser analizadas.

Cada caso de uso se puede componer internamente de una **secuencia de eventos y es iniciada por un tipo de usuario o agente, llamado actor**.

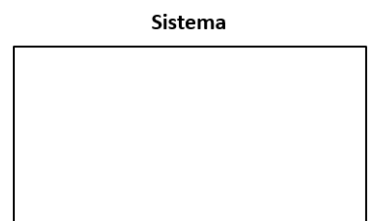
Permiten **modelar los requisitos funcionales y los perfiles de usuario** involucrados en su utilización, para **favorecer el análisis**.

## Elementos: Sistema

Es uno de los elementos más importantes en un diagrama de casos de uso.

Hace referencia a **solución software integral que vamos a estudiar o analizar**, para implementarla.

Se representa mediante un **rectángulo**.

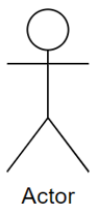


## Elementos: Actores

Cada uno de los agentes o tipos de **usuario que interactúa con un sistema**.

Un actor representa un **rol externo al sistema que interactúa** con el propio sistema.

**No son únicamente humanos**, un actor podría ser otro sistema, dispositivo o el tiempo, por ejemplo:



- Un usuario que solicita la compra de un producto.
- Un dispositivo externo que mediante una aplicación intercambia datos con nuestro sistema.
- Un temporizador que ejecuta un proceso cada 10 segundos.

Un actor representa un perfil concreto de usuario, de modo que:

- Un usuario puede representar a varios actores.
- Un actor puede ser representado por varios usuarios.



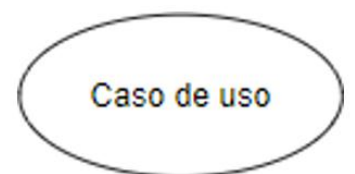
## Elementos: Casos de uso

Los casos de uso representan un **proceso o acción** (crear pedido, mover ficha...).

Gráficamente, los casos de uso se representan en UML mediante una **elipse**.

**No podrán estar aislados** en un sistema, siempre estarán enlazados con uno o varios actores y/o con uno o varios casos de uso.

Para identificar los casos de uso existentes en un sistema, debemos averiguar qué **funciones e interacciones realiza cada actor** en el sistema.



Los casos de uso deben **abstraer** en gran medida lo fundamental o principal respecto a las interacciones de un sistema, por muy grande que éste sea.

## ¿Cómo generar diagramas UML?

Para poder elaborar un diagrama UML, como **primer paso**, debemos **estudiar y analizar** con el mayor detalle posible el escenario o supuesto sobre el que vamos a trabajar.

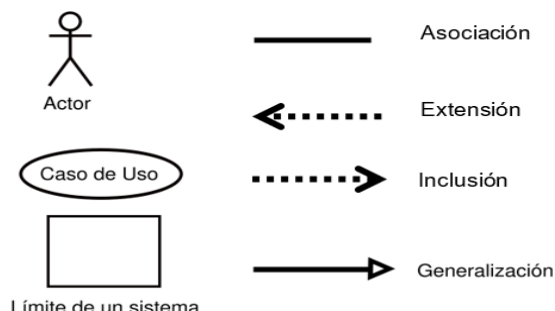
La **precisión** en cada uno de los pasos favorece la evolución del proyecto.

Si nos equivocamos u olvidamos algún aspecto importante al principio, nos costará un esfuerzo considerable durante las fases posteriores del proyecto.

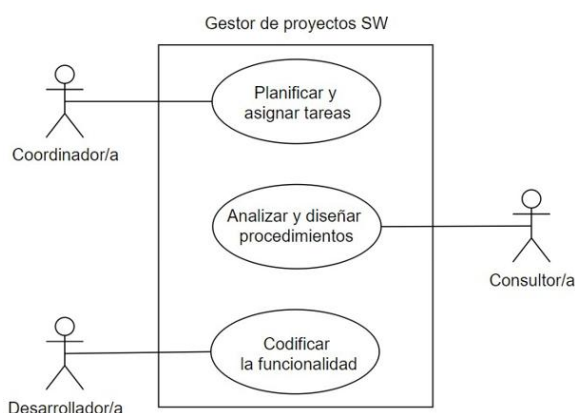
## Análisis y diseño de diagramas de casos de uso

**Identificación de elementos:** Uno de los resultados del análisis realizado, debe ser el inventario de elementos que formarán el diagrama. Identificar:

- **Sistema:** Tener claro límites y objetivos de la solución software a analizar.
- **Actores:** Se representarán como entidades externas al sistema, que interactúan con él.
- **Casos de uso:** Elementos internos del sistema, para identificarlos habrá que estudiar las necesidades e interacciones que llevará a cabo cada actor.
- **Relaciones**



### Ejemplo:



Dado un sistema para la gestión de proyectos software.

Habrà una persona que se encargará de planificar y asignar tareas en cada proyecto, ejerciendo las funciones de coordinación.

También habrá un equipo de consultoría que analizará y diseñará los procedimientos del proyecto.

Por último, un equipo de desarrollo codificará la funcionalidad del proyecto.

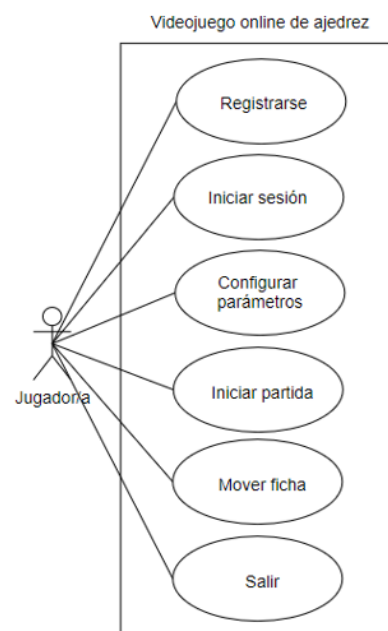
### Ejemplo:

Pensemos ahora en un videojuego de ajedrez online.

Los únicos que interactúan son los Jugadores.

Las interacciones que pueden realizar son:

1. Registrarse.
2. Iniciar sesión.
3. Configurar parámetros.
4. Iniciar partida:
5. Mover ficha.
6. Salir.



## RELACIONES Y PARTICULARIDADES

### ¿Qué es una relación?

**Forma en que interactúan y se comunican los elementos** que forman los diagramas de casos de uso.

Es una **conexión entre elementos** del modelo o diagrama.

Existen varios tipos de relaciones o conexiones.

Relación	Notación
<b>Asociación</b>	—————
<b>Generalización</b>	—————>
<b>Inclusión</b>	-----> «include»
<b>Extensión</b>	-----> «extend»



### Tipos de relación: Asociación

Conecta **actores y casos de uso**.

Indica que un actor hará uso de la función definida en cada caso de uso asociado.

Es la **relación más básica y sencilla**, y probablemente **la más habitual**.

Cada asociación se simboliza mediante **una línea**, que une un actor y un caso de uso.

### Tipos de relación: Inclusión

Conecta **distintos casos de uso**.

La relación de inclusión se usa cuando **la ejecución de un caso de uso origen implica la ejecución obligatoria de otro caso de uso destino**.

Se simboliza mediante una **flecha discontinua** de punta abierta, con inicio en el caso de uso origen y final en el destino, y con la etiqueta **<<include>>**

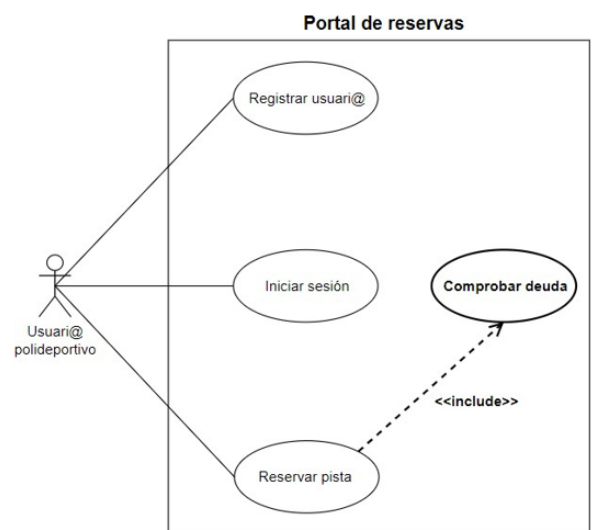
El caso de uso origen incluye la ejecución del caso de uso destino.

#### Ejemplo:

Vamos a suponer que nos solicitan realizar un análisis para desarrollar un portal web que gestione las reservas de pistas de un polideportivo.

Nos indican que las posibles interacciones de cualquier usuario en el portal serán:

1. Registrarse.
2. Iniciar sesión.
3. Reservar pista: al intentar reservar una pista, será obligatorio comprobar el estado de deuda del usuario.



### Tipos de relación: Extensión

Conecta **distintos casos de uso**.

La relación de extensión se usa cuando **la ejecución de un caso de uso origen, bajo determinadas circunstancias, puede implicar la ejecución condicional de otro caso de uso destino, que extiende o modifica su funcionalidad**.

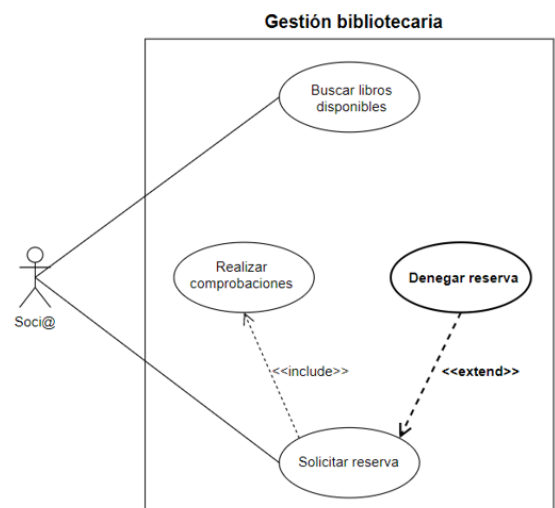
Se simboliza mediante una **flecha discontinua** con punta abierta, con inicio en el caso de uso destino y final en el origen, y con la etiqueta **<<extend>>**

#### Ejemplo:

Vamos a suponer ahora que nos solicitan realizar un análisis para desarrollar una solución para gestionar una biblioteca.

Entre los requerimientos, nos indican que cualquier socio de la biblioteca podrá realizar las siguientes interacciones:

1. Buscar libros disponibles.
2. Solicitar una reserva:
  - a. Habrá un máximo de libros que se puedan reservar, por lo que cuando se solicita una reserva, se deben realizar comprobaciones.
  - b. Bajo determinadas circunstancias, se podrá denegar la reserva solicitada.





## Inclusión y extensión

Tomando como base el un ejemplo anterior del videojuego de ajedrez en red, nos piden ampliar los requerimientos. De modo que:

1. Iniciar sesión implicará validar los datos del usuario.
2. Durante la configuración de los parámetros, podremos elegir jugar contra una máquina o contra otro usuario, y en ese caso habrá que buscar oponente.
3. Al mover ficha, tendremos la posibilidad de pedir consejo y de rendirnos directamente.

## Generalización/especialización

Es una relación que se da, o bien **entre casos de uso** o bien **entre actores**.

La relación de generalización **se aplicará a un componente padre**, que se usa para generalizar **uno o más componentes hijos especializados**.

Se simboliza mediante una **flecha con la cabeza vacía**.

Conocida también como **Herencia**.

### Ejemplo:

Vamos a suponer ahora que nos solicitan realizar un análisis para gestionar un cajero automático. De modo que, un cliente del banco puede llevar a cabo diferentes interacciones:

1. Validar sus credenciales, por ejemplo, con su tarjeta y su clave asociada.
2. Indicar la realización de cualquier operación disponible.

Cada operación tendrá unas características comunes con el resto de operaciones y otras diferentes o especializadas. Las comunes serán:

- En cualquier operación se puede pedir ayuda en línea (on-line), si lo necesitas.
- Cada operación requiere una comprobación obligatoria de los permisos del cliente.
- Toda operación podrá ser cancelada.

En función de la opción que seleccione, el cliente podrá realizar las siguientes operaciones:

1. Retirar efectivo
2. Realizar una transferencia
3. Pagar un recibo
4. Ingresar efectivo
5. Consultar el saldo disponible
6. Salir del Sistema

## Documentación asociada

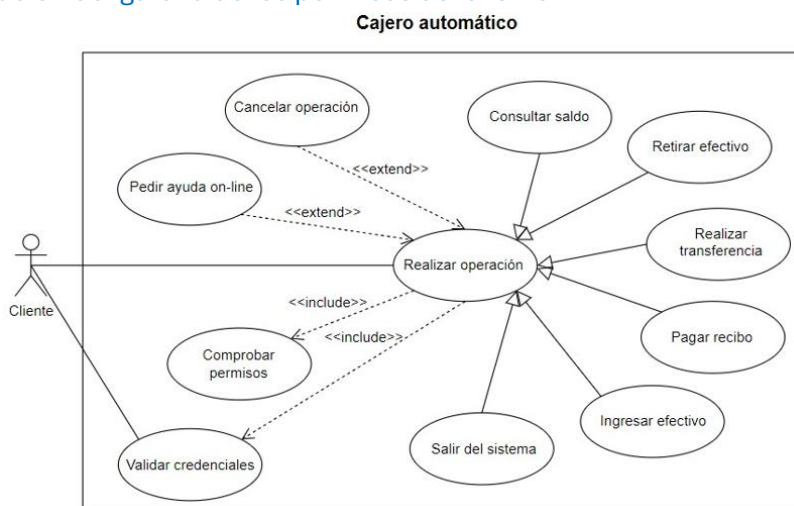
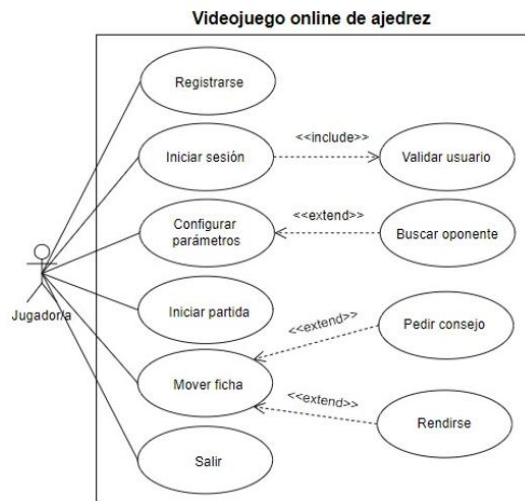
### Ejemplo de descripción técnica del caso de uso “Validar credenciales”:

**Diagrama:** Cajero automático

**Nombre del caso de uso:** Validar credenciales.

**Actores:** Cliente del cajero automático.

**Descripción:** Este caso de uso permite al cliente validar sus credenciales en el sistema.



### Flujo básico:

- 1) Si el cliente no ha validado sus credenciales, se muestra un mensaje indicándole las instrucciones para validarse en el sistema.
- 2) Se muestra al cliente el formulario para validar sus credenciales.
- 3) Si las credenciales no son correctas, se vuelve al paso 2 del flujo básico.
- 4) En caso contrario, se muestra el menú estándar de operaciones que se pueden realizar desde un cajero automático.

### Ejemplo de descripción técnica del caso de uso “Realizar operación”:

**Diagrama:** Cajero automático

**Nombre del caso de uso:** Realizar operación.

**Actores:** Cliente del cajero automático.

**Descripción:** Este caso de uso permite al cliente realizar una operación bancaria.

### Flujo básico:

- 1) Si el cliente no ha validado sus credenciales, se muestra un mensaje indicándole que debe validarse en el sistema.
- 2) Si el cliente está validado, puede seleccionar una opción del menú estándar de operaciones que se pueden realizar desde un cajero automático.
- 3) Se muestra al cliente el formulario que corresponda para que introduzca los datos necesarios para realizar la operación.
- 4) Se comprueban los permisos del cliente para llevar a cabo la operación indicada.
- 5) Si la comprobación no es satisfactoria, se muestra un mensaje indicando al cliente que no se puede realizar la operación.
- 6) En caso contrario, el cliente puede confirmar la operación.
- 7) Se vuelve al paso número 2 del flujo básico.

### Extensiones:

- 1) El cliente puede hacer uso de la ayuda en línea, si lo considera adecuado.
- 2) La operación puede ser cancelada durante el proceso.

### Inclusiones:

- 1) Para poder realizar cualquier operación, es obligatorio confirmar los permisos del cliente.
- 2) Para poder realizar cualquier operación, es obligatorio que el cliente haya validado sus credenciales.

**Herencia** (generalización/especialización). Se consideran casos de uso hijos:

1. Retirar efectivo
2. Consultar saldo
3. Realizar transferencia
4. Pagar recibo
5. Ingresar efectivo
6. Salir del sistema

## UNIDAD 3. UML COMPORTAMIENTO: ACTIVIDADES

### COMPONENTES

#### ¿Qué significa el término actividades?

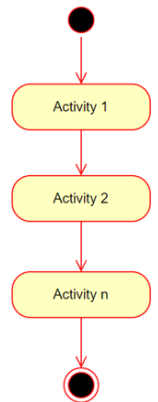
**R.A.E.:** Conjunto de operaciones o tareas propias de una persona o entidad.

Por lo tanto, se puede considerar a las actividades como un **conjunto de acciones que modelan un proceso** o un sistema.

#### ¿Qué es un diagrama UML de actividades?

Los diagramas de actividades **modelan el comportamiento dinámico** de un procedimiento, transacción, proceso, sistema, algoritmo o **caso de uso**, haciendo énfasis en las acciones que se llevan a cabo.

Permiten describir, durante las fases de análisis y diseño, cómo un sistema desarrolla su **funcionalidad de forma secuencial**, estableciendo un **flujo de trabajo paso a paso**.



#### Ejemplos de actividades

Los diagramas de actividades se utilizan, además de en el desarrollo de software, en otras disciplinas como pueden ser economía, procesos industriales o psicología cognitiva.

#### Diferencia entre UML de Actividades y DFD

**Simbología:** Utilizan símbolos diferentes para representar sus componentes.

**Semántica o significado:**

- En un **DFD**:
  - Lo que conecta a los diferentes componentes es el flujo de información.
  - Tiene una aplicación específica en el desarrollo de software, ya que nos lleva a una traducción bastante directa de cara a desarrollar código fuente.
- En un **diagrama de actividades**:
  - Lo que conecta a los diferentes componentes es la secuencia de operaciones.
  - Es más general y abstracto.

#### Simbología y notación

Inicio del diagrama o del proceso + flujo de secuencia.

Fin del diagrama o proceso.

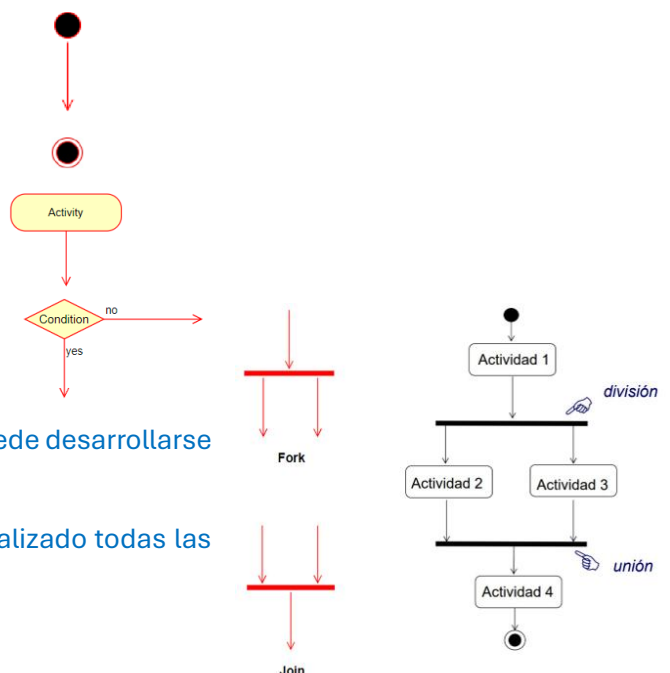
Actividad + flujo de secuencia.

Decisión condicional + flujos de secuencia.

#### Rutas concurrentes

**Fork (división):** Indica que un conjunto de actividades puede desarrollarse en paralelo (concurrentemente).

**Join (unión):** Indica que, para continuar, deben haber finalizado todas las actividades paralelas.



### Ejemplo:

Una empresa de desarrollo de software se encuentra en proceso de negociación con un cliente para cerrar un proyecto y deciden preparar una reunión conjunta para concretar los requisitos del mismo. Una persona del equipo de consultoría de la empresa se encarga de concertar una cita.

Si la cita es en la oficina de la empresa: Una persona del equipo técnico de sistemas preparará una sala de conferencias para mantener la reunión.

Si la cita es en la oficina del cliente: La persona de consultoría debe solicitar un portátil para llevar a las instalaciones del cliente.

Posteriormente, la persona de consultoría y el cliente mantendrán la reunión en el lugar acordado.

Tras mantener la reunión: El consultor redactará una propuesta con los requerimientos y los acuerdos establecidos durante la reunión y se la enviará al cliente.

Si el cliente no está de acuerdo en algún aspecto, el consultor volverá a redactar o preparar la propuesta y enviarla al cliente.

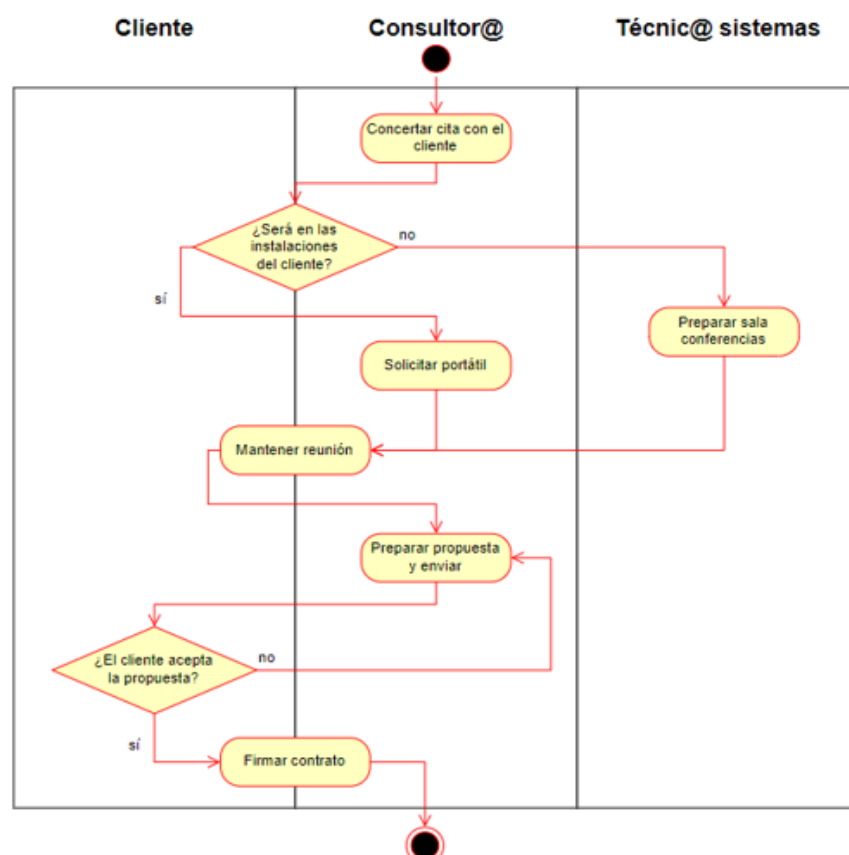
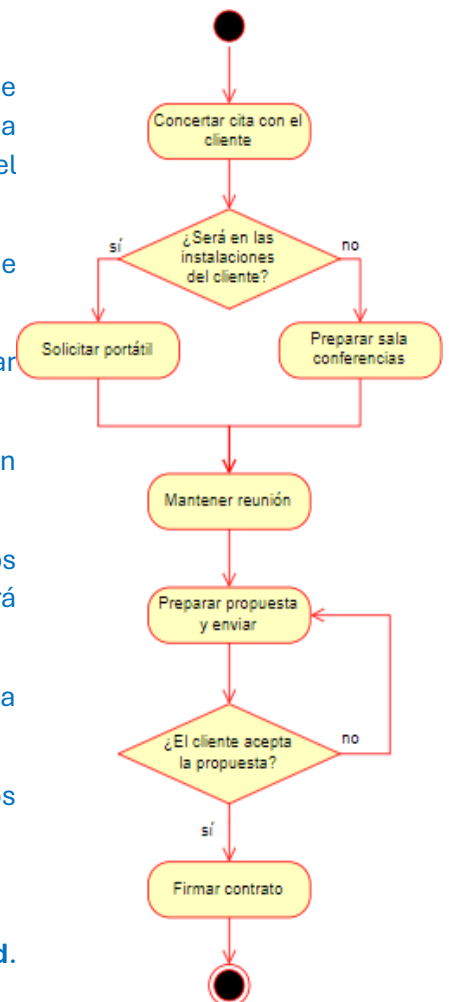
Si el cliente está de acuerdo en todo (acepta la propuesta comercial), ambos firmarán un contrato para llevar a término el proyecto.

### Marcos de responsabilidad (swimlanes)

Nos sirven para **saber qué actor es el encargado de realizar cada actividad**.

Se **representan mediante rectángulos o segmentos paralelos verticales**, permitiendo ver gráficamente quién realiza cada actividad.

Cada marco muestra el nombre del actor en la parte superior y las actividades que le corresponden. Las transiciones pueden llevarse a cabo de un marco a otro.



## UNIDAD 3. UML COMPORTAMIENTO: ACTIVIDADES

### ESTRUCTURAS DE CONTROL

#### ¿Qué son estructuras de control?

Se entiende por estructura de control aquel conjunto de componentes que nos permiten **controlar el flujo de ejecución** de las instrucciones o actividades de un sistema con un objetivo.

#### Tipos de estructuras de control

**Secuencial:** Las instrucciones (o actividades) se ejecutan en orden de aparición. Es el funcionamiento más básico.

**Condicional:** En función de si se cumple o no cierta condición, se ejecutarán unas instrucciones (o actividades) u otras.

**Iterativa:** Se repite la ejecución de un conjunto de instrucciones (o actividades) un número controlado de veces.

#### Estructura de control secuencial

Inicio

Actividades

...

...

Fin



#### Estructura de control condicional simple

##### Pseudocódigo:

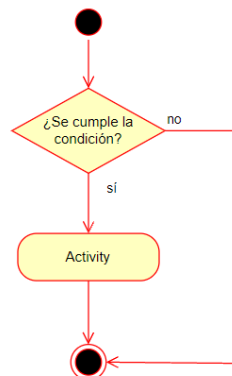
Inicio

Si <condición>

Actividad

Fin Si

Fin



##### JavaScript:

```
if (condición)
```

```
{
```

```
  actividad
```

```
}
```

##### Pseudocódigo:

Inicio

Si <condición>

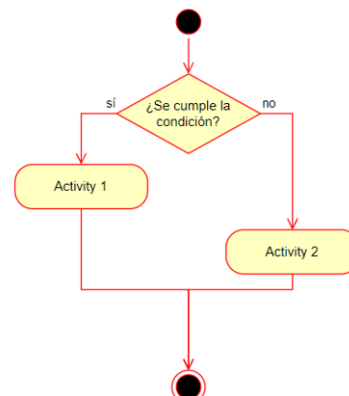
Actividad 1

Sino

Actividad 2

Fin Si

Fin



##### JavaScript:

```
if (condición)
```

```
{
```

```
  actividad_1
```

```
}
```

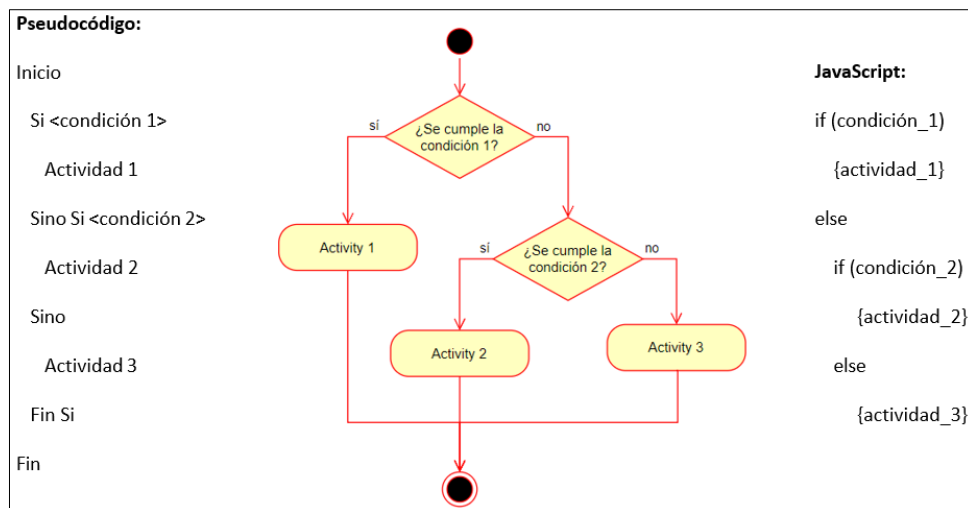
```
else
```

```
{
```

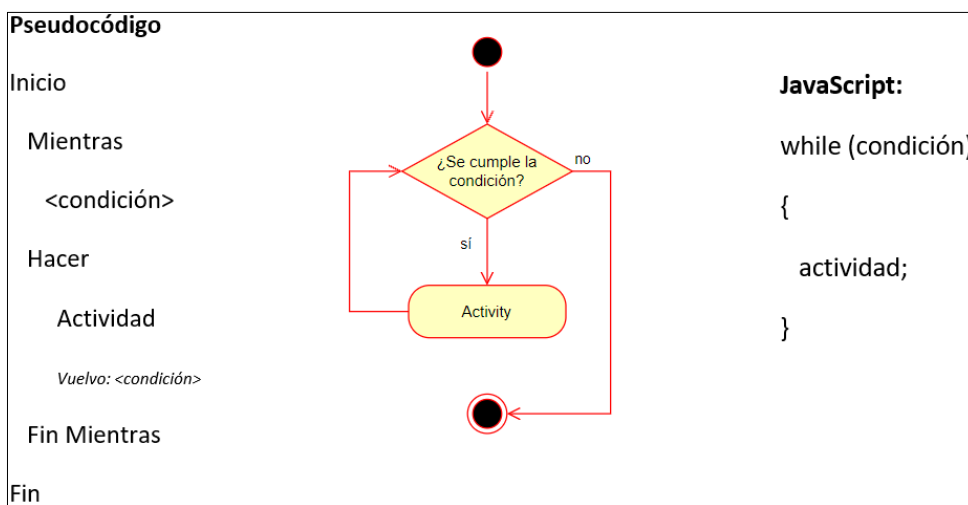
```
  actividad_2
```

```
}
```

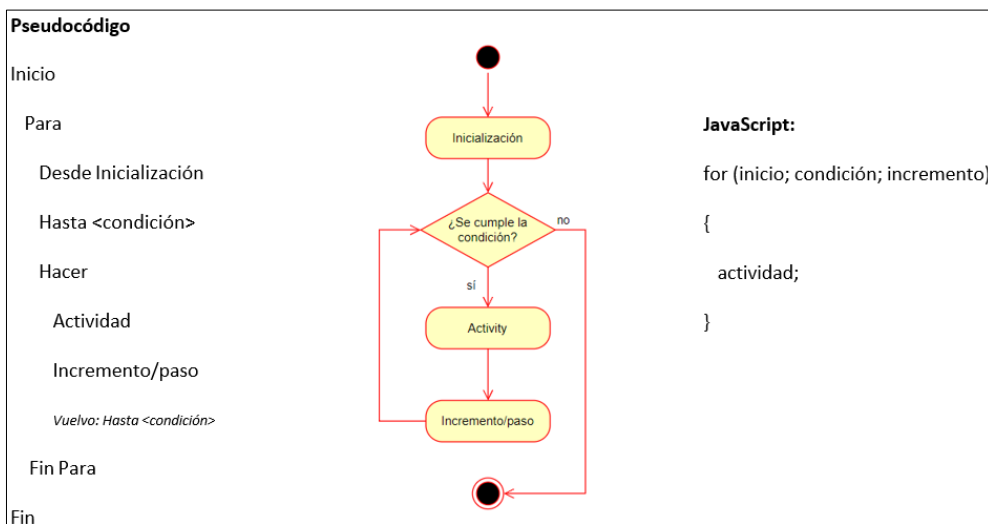
## Estructura de control condicional anidada



## Estructura de control iterativa. WHILE



## Estructura de control iterativa. FOR






## ARRAYS

### ¿Qué es un array?

Es una **estructura de datos** o, dicho de otro modo, un **tipo de dato estructurado**.

Consta de una **serie o colección de elementos**.

En función de cuántas **dimensiones** tenga, puede ser:

-  **Unidimensional:** Vector.
-  **Bidimensional:** Matriz (tabla).
-  **N-dimensionales**

### Declaración de arrays. Posiciones

Un array se declara:

\*Por defecto, un array **empieza en la posición o índice 0**.

**Unidimensional:** Ejemplo en JavaScript:

```
let nombre_array = [valor_0, valor_1, ...,
                    valor_n];
```

**Bidimensional:** Ejemplo en JavaScript (array de arrays):

```
let nombre_array = [
    [valor_0_0, valor_0_1, ..., valor_0_n],
    [valor_1_0, valor_1_1, ..., valor_1_n],
    [..., ..., ..., ...],
    [valor_n_0, valor_n_1, ..., valor_n_n]
];
```

### Asignación/Consulta de valores en arrays

**Asignar** un valor en esa posición:

**Unidimensional:** Ejemplo en JavaScript:

```
vector[0] = 8;
vector[0] = "Jorge";
vector[5] = 14;
```

**Bidimensional:** Ejemplo en JavaScript:

```
matriz[0][0] = 8;
matriz[0][0] = "Jorge";
matriz[5][3] = 14;
```

**Consultar** el valor de esa posición:

**Unidimensional:** Ejemplo en JavaScript:

```
console.log(vector);
console.log(vector[0]);
variable = vector[5];
```

**Bidimensional:** Ejemplo en JavaScript:

```
console.log(matriz);
console.log(matriz[0][0]);
variable = matriz[5][3]
```

### Longitud de arrays

**Unidimensional:**

```
longitud = vector.length;
ultimo_elemento = vector.length - 1;
```

**Bidimensional:**

```
numero_filas = matriz.length;
longitud_columna = matriz.length;
longitud_fila_0 = matriz[0].length;
// ...
longitud_fila_n = matriz[n].length;
longitud_total = Σ(longitud_fila_x)
```



Si longitud\_fila\_0 = longitud\_fila\_1 = .... = longitud\_fila\_n  $\rightarrow$  longitud\_total = longitud\_columna \* longitud\_fila\_x

### Recorrido de arrays unidimensionales

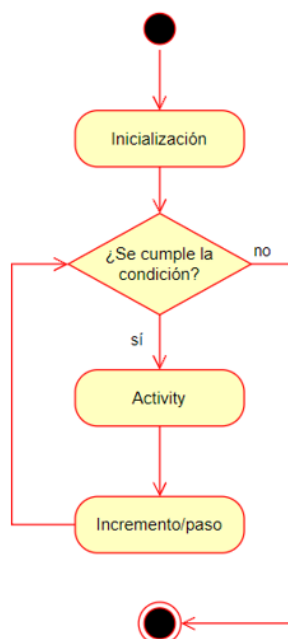
Utilizamos la estructura de control FOR, donde:

- **Inicialización:** Indicamos en qué componente del array comenzará el recorrido. (ejemplo:  $i = 0$ )
- **Condición:** Mientras se cumpla la condición, continuamos el recorrido. Si no se cumple, finalizamos. (ejemplo:  $i < \text{vector.length}$ )
- **Incremento o paso:** Avanzamos al siguiente elemento del recorrido. (ejemplo:  $i = i + 1$ )

**Ejemplo en JavaScript:**

```
for (i = 0; i < vector.length; i++) {
    actividad;
}
```

7	9	1	0	12	50
0	1	2	3	4	5



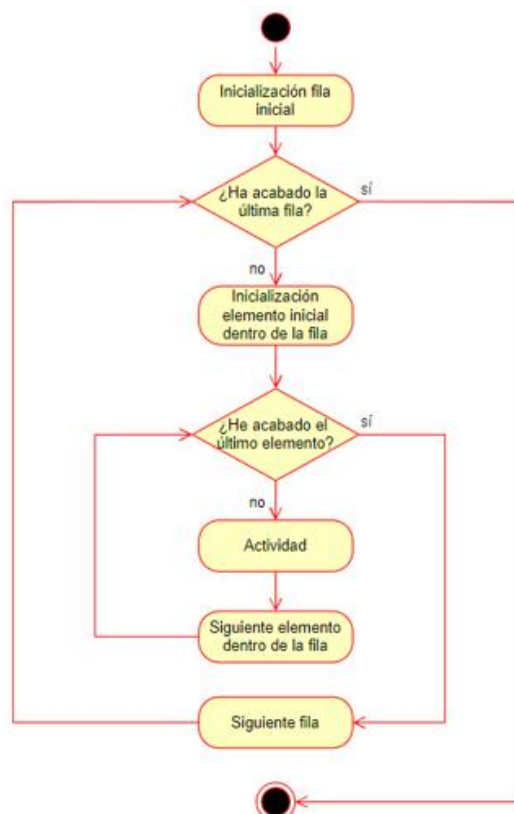
### Recorrido de arrays bidimensionales

Utilizamos dos estructuras de control FOR anidadas.

**Ejemplo en JavaScript:**

```
// Recorremos cada fila
// matriz.length = número de filas = longitud columna
for (i = 0; i < matriz.length; i++) {
    // Recorremos cada elemento dentro de una fila
    // matriz[i].length = longitud de cada fila
    for (j = 0; j < matriz[i].length; j++) {
        actividad;
    }
}
```

	0	1	2
0			
1			
2			
3			



## UNIDAD 4. DEPURACION (DEBUGGING)

### ¿Qué significa depurar código?

**Depurar** según la **R.A.E.**: Limpiar, purificar.

#### Depurador de código:

- ✓ Es una **herramienta** que permite **ejecutar código paso a paso** (instrucción a instrucción).
  - Su objetivo suele ser ayudar en la identificación y corrección de errores.
  - Suele estar a disposición de los desarrolladores en los editores de código y en los entornos de desarrollo.

### Puntos de interrupción

Sirven para poder indicar al depurador **dónde debe detenerse**.

Generaremos un punto de interrupción en aquella instrucción en la que queremos que se inicie el proceso de depuración.

Podremos generar tantos puntos de interrupción como necesitemos.

### Navegación

**Continuar:** Ejecuta el código hasta el final.

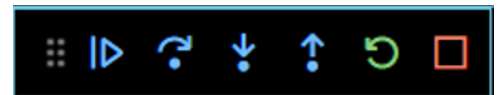
**Depurar paso a paso por procedimientos:** Si la línea actual contiene una llamada de función, ejecuta el código de la misma, y suspende la ejecución en la primera línea de código después de que se devuelva la función a la que se ha llamado.

**Depurar paso a paso por instrucciones:** Ejecuta cada instrucción y suspende la ejecución en la siguiente.

**Depurar paso a paso para salir:** Continúa ejecutando el código y suspende la ejecución cuando se devuelve la función actual.

**Reiniciar:** Reinicia la ejecución del código.

**Detener:** Detiene el depurador.



### Utilidades

**Inspección de variables:** Una de las utilidades más importantes, permite inspeccionar los valores de las variables en tiempo real para cada instrucción.

**Consola de depuración:** Permite interactuar con la aplicación en tiempo de ejecución, mientras se está depurando.

