

# UT05.- DISEÑO Y REALIZACIÓN DE PRUEBAS.

# Índice

1.- Planificación de las pruebas.

2.- Tipos de prueba.

2.1.- Funcionales.

2.2.- Estructurales.

2.3.- Regresión.

3.- Procedimientos y casos de prueba.

4. Pruebas Funcionales

4.1- Clases de equivalencia

4.2- Valores límite

4.3- Aleatorias

5. Pruebas Estructurales

5.1- Cubrimiento

6. Pruebas de Regresión

7.- Herramientas de depuración.

7.1.- Puntos de ruptura.

7.2.- Tipos de ejecución.

7.3.- Examinadores de variables.

8.- Validaciones.

9.- Normas de calidad.

10.- Pruebas unitarias.

10.1.- Herramientas para Java.

10.2.- Herramientas para otros lenguajes.

11.- Automatización de la prueba.

12.- Documentación de la prueba.

## **1. Planificación de las pruebas.**

# Planificación de las pruebas

- Las pruebas software son esenciales para obtener **software de calidad**.
- Las pruebas constituyen **una de las etapas del desarrollo del software**.
- Es necesario realizar un conjunto de pruebas
  - ➔ software correcto
  - ➔ software que cumple con las **especificaciones** impuestas por el usuario.
- **Errores** del proceso de desarrollo de software:
  - Una incorrecta especificación de los objetivos.
  - Errores producidos durante el proceso de diseño.
  - Errores que aparecen en la fase de desarrollo.

# Planificación de las pruebas

Una prueba es el proceso por el cual ejecutamos un producto software con la intención de verificar y validar su correcto funcionamiento.

- ▣ **Verificación:** comprobación que un sistema o parte de un sistema, **cumple los requisitos funcionales y no funcionales del software**. ¿Se está construyendo correctamente?
- ▣ **Validación:** proceso de evaluación del sistema o de uno de sus componentes, para determinar si **satisface las expectativas del cliente**. ¿El producto es correcto?

No probar de forma adecuada un software puede acarrear consecuencias: pérdidas económicas, pérdida de prestigio, incluso dependiendo del tipo de software, daños en instalaciones o personas.

# Planificación de las pruebas

- Es necesario implementar una **ESTRATEGIA DE PRUEBAS**:
  - La forma en la que mayormente se aplican las pruebas del software se asemejan a una espiral:



# Planificación de las pruebas

- ❑ **Pruebas de unidad:** se centran en la unidad más pequeña del software → módulo de código fuente.
  - ❑ Se centra en probar cada unidad o módulo. Se pretende eliminar errores de lógica interna como en la interfaz, usando **técnicas de caja negra y caja blanca**.
  - ❑ Se utilizan herramientas como **Junit**.
- ❑ **Pruebas de integración:** se prueba la interacción entre los módulos, que aun funcionando por separado pueden fallar al comunicarse.
  - ❑ Se construye una estructura de programa acorde al diseño, con todos los módulos probados.
- ❑ **Pruebas de validación:** prueba el software de acuerdo con lo que el cliente espera, es decir, de acuerdo con lo especificado en el **análisis de requisitos del software**.
  - ❑ Se utilizan pruebas de **caja negra** en el entorno real de trabajo con intervención del usuario final.
- ❑ **Prueba del sistema:** pone a prueba el sistema en conjunto y lo lleva al límite. Verifica que cada elemento encaja de forma adecuada y alcanza la funcionalidad y rendimiento total.

# Reflexión

---

El 40% del esfuerzo de desarrollo se emplea en las pruebas.

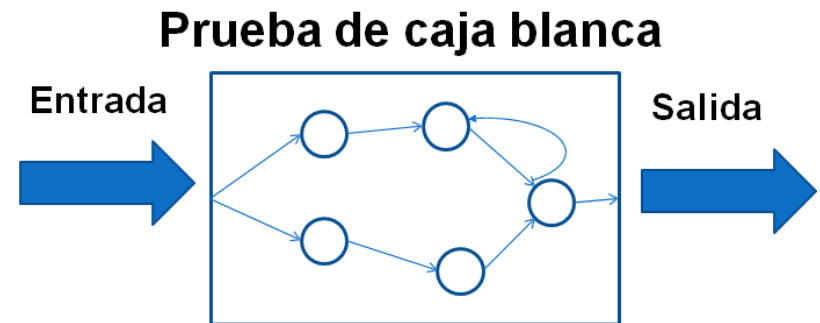
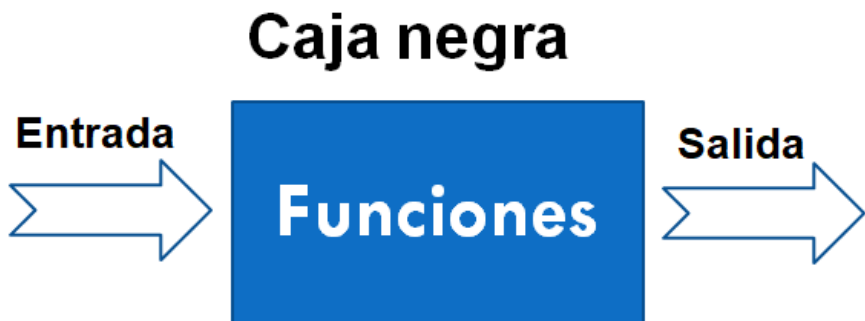
Las pruebas son importantes ya que permiten descubrir errores en un programa, fallos en la implementación, calidad o usabilidad del software, ayudando a garantizar la calidad.



## 2. Tipos de pruebas

# Tipos de pruebas

- La ingeniería del software contempla dos enfoques (no excluyentes):
  - ▣ **Pruebas de Caja Negra:** se centran en validar **los requisitos funcionales** indicados en la especificación de requisitos, sin tener en cuenta el funcionamiento interno del programa.
  - ▣ **Pruebas de Caja Blanca:** se centran en validar el **comportamiento interno** del programa.



# Tipos de pruebas

## Pruebas de Caja Negra (Black Box Testing):

- ❑ También llamadas **Pruebas de comportamiento o funcionales**.
- ❑ Se prueba la aplicación mediante su interfaz externa, sin conocer la implementación de la misma.
- ❑ Comprueban que **los resultados** de la ejecución de la aplicación, **son los esperados**, en función de las entradas que recibe.
- ❑ No es necesario conocer ni la estructura, ni el funcionamiento interno del sistema.
- ❑ Solo se conocen las entradas a recibir por la aplicación, y las salidas que les correspondan, pero no se conoce el proceso mediante el cual la aplicación obtiene esos resultados.

# Tipos de pruebas

- El objetivo es encontrar errores que se encuentran:
  - ▣ Funcionalidades incorrectas o ausentes
  - ▣ Errores en la interfaz E/S
  - ▣ Errores en estructuras de datos o en accesos a BD externas
  - ▣ Errores de rendimiento
  - ▣ Errores de inicialización y finalización

# Tipos de pruebas

## **Prueba de la Caja Blanca (White Box Testing):**

- ▣ También llamadas **Pruebas Estructurales**.
- ▣ Prueban la aplicación desde dentro, usando su lógica de aplicación.
- ▣ Se **analiza y prueba directamente el código de la aplicación**, por lo que es necesario un conocimiento específico del código.

# Tipos de pruebas

- ▣ Se pueden obtener **casos de prueba** que:
  - Garanticen que se ejecutan al menos una vez todos los caminos independientes de cada módulo.
  - Ejecuten todas las sentencias al menos una vez .
  - Ejecuten todas las decisiones lógicas en su parte verdadera y en su parte falsa.
  - Ejecuten todos los bucles en sus límites.
  - Utilicen todas las estructuras de datos internas para asegurar su validez.

### **3. Procedimientos y casos de prueba**

# Pruebas de código

- ❑ Las pruebas de código consisten en la ejecución del programa (o parte de él) para encontrar errores.
- ❑ Para llevar a cabo las pruebas de código hay que definir una serie de **casos de prueba**.
- ❑ Según el IEEE:
  - ❑ *Un caso de prueba es un conjunto de entradas, condiciones de ejecución y resultados esperados, desarrollados para un objetivo particular, como, por ejemplo, ejercitar un camino concreto de un programa o verificar el cumplimiento de un determinado requisito, incluyendo toda la documentación asociada.*



# Pruebas de código

- La complejidad de las aplicaciones informáticas imposibilita probar todas la combinaciones
- Por ello, se debe asegurar que los casos de prueba obtienen un nivel aceptable de probabilidad de detección de errores.
- El diseño de casos de prueba se puede abordar con **3 tipos de técnicas o enfoques:**
  - Pruebas Funcionales.
  - Pruebas Estructurales.
  - Pruebas de Regresión

# Pruebas de código

- **Para llevar a cabo un caso de prueba es necesario:**
  - Definir las precondiciones y postcondiciones,
  - identificar los valores de entrada adecuados,
  - conocer el comportamiento que debería tener el sistema ante dichos valores (resultados).
- Tras realizar el análisis e introducir los datos en el sistema, **se determina si el sistema pasa la prueba** (ver si su comportamiento es el previsto o no y por qué)
- Debe definirse al menos **un caso de prueba para cada requerimiento** a menos que éste tenga requisitos secundarios. En ese caso, cada requerimiento secundario deberá tener por lo menos un caso de prueba.

## 4. Pruebas funcionales

# Pruebas Funcionales.

- Son pruebas de la **caja negra**.
- **Objetivo:** verificar la funcionalidad del software
  - ¿Puede el usuario hacer esto?
  - ¿Funciona esta utilidad de la aplicación?
- Analiza las **entradas** y las **salidas** de cada componente, **verificando que el resultado es el esperado**, sin preocuparnos por la estructura interna del mismo (código).
- Aún así, con estas técnicas no podemos garantizar que el módulo/s de código estén libres de errores.

# Pruebas Funcionales.

- Tipos de pruebas funcionales:

- **Particiones equivalentes o clases de equivalencia.**

- Clasificamos los datos de entrada en grupos que pueden presentar un comportamiento similar y por tanto procesaremos del mismo modo. Habrá clases tanto para datos de entrada válidos como no válidos (que deben ser rechazados por el sistema).

- **Análisis de valores límite.**

- Relacionadas con las anteriores, se crean casos de prueba específicos para los valores máximos y mínimos admitidos para cada dato de entrada y salida.

- **Pruebas aleatorias.**

- Se generan entradas aleatorias para la aplicación a probar. Se pueden usar generadores de pruebas, los cuales generan casos de prueba al azar.
    - Se usan en aplicaciones no interactivas.

# Funcionales: clases de equivalencia

- Consiste en la división del **dominio de datos de entrada** en un número finito de **particiones** o **clases de equivalencia**, válidas y no válidas, que cumplan la siguiente propiedad:

“La prueba con un valor representativo de una clase permite suponer razonablemente que el resultado obtenido será el mismo que el obtenido probando cualquier otro valor de la clase”

- Para identificar las clases de equivalencia se examina cada **condición de entrada** y se divide en 2 o más grupos obteniendo:
  - **Clases válidas:** conjunto de valores de entrada válidos
  - **Clases no válidas:** conjunto de valores de entrada no válidos
- Cada partición se puede subdividir en **subparticiones** si cada valor de entrada tiene diferentes tratamientos.
- Cada valor de entrada pertenece a una y solo una partición.
- **Objetivo:** Reducir el número total de casos de prueba a un conjunto de casos comprobables que cubran el mayor número de entradas posible.

# Funcionales: clases de equivalencia

- ▣ Ejemplo: si se está probando una entrada que acepta números del 1 al 1000, no tiene sentido escribir 1000 casos de prueba para los 1000 posibles datos de entrada válidos, más otros casos para datos de prueba no válidos.
- ▣ Esta técnica permite en este caso, dividir en tres conjuntos de datos de entrada llamados clases. Cada caso de prueba es representativo de una clase.
  - ▣ Una clase para los valores entre 1 y 1000
  - ▣ Otra clase para los valores por debajo de 1
  - ▣ Otra clase para los valores por encima de 1000

# Funcionales. clases de equivalencia

## ■ Reglas para identificar las clase de equivalencia:

- **Rangos:** si un dato de entrada pertenece a un rango de valores (por ejemplo, edades comprendidas entre 18 y 65 años ambos inclusive), se define una **1 clase válida** de equivalencia (valores del rango) y **2 clases no válidas** (2 extremos del rango).
- **Valor específico:** análoga a la anterior, solo que el rango queda restringido a un único valor (la edad es de 18 años). Se define **1 clase de equivalencia válida** y **2 no válidas**.
- **Número de valores:** igual que las anteriores, pero teniendo en cuenta el número de valores introducidos (ej. de 1 a 5 pasajeros sería una clase válida, 0 pasajeros y más de 5 pasajeros serían las no válidas).
- **Valor lógico (boolean o “debe ser”):** se corresponde a la evaluación de una condición (ej. La edad debe ser  $\geq$  a 18 años). Se establece **1 clase válida** (se cumple la condición) y **1 clase no válida** (no se cumple).
- **Conjunto de valores:** una entrada es válida si pertenece a un conjunto (ej. días de la semana). En este caso se define **1 clase válida para cada uno de los valores del conjunto** y **1 clase no válida** para el resto.



# Funcionales. clases de equivalencia

## ■ Ejemplos de clases de equivalencia:

- **Día de la semana numérico:** La condición de entrada refleja que sólo se podrán introducir números del 1 al 7, ambos inclusive.
  - Clase válida:  $1 \leq \text{día} \leq 7$
  - Clase no válida:  $\text{día} < 1$
  - Clase no válida:  $\text{día} > 7$
- **Colores RGB en formato String minúscula:** El valor de entrada sólo puede corresponder a uno de los colores RGB escrito en minúscula: «red», «green», «blue». Se supone que cada una de esas entradas se debería manejar de formas distintas en el programa.
  - 3 Clases válidas una para cada color: “red”, “green”, “blue”
  - 1 clase no válida para el resto de valores
- **Nombre con primera letra en mayúscula:**
  - Clase válida para cadenas cuya primera letra es una mayúscula (se cumple la condición)
  - Clase no válida para valores que no cumplen la condición: cadenas que comienzan en minúscula.

# Funcionales: clases de equivalencia

- **Elaborar la tabla de clases de equivalencia:** (Etiquetar cada clase de equivalencia con un identificador único)

Entrada	Tipo	Clases Válidas	ID	Clases no Válidas	ID
Día de la semana	Rango	1 <= día <= 7	v_dia	día < 1	nv_dia_menor
				día > 7	nv_dia_mayor
Colores	Conjunto	color = "red"	v_color_r	color distinto de los válidos	nv_color
		color = "green"	v_color_g		
		color = "blue"	v_color_b		
Nombre de usuario	Condición lógica	Empieza por mayúscula	v_mayúscula	No empieza por mayúscula	nv_mayúscula

# Funcionales: clases de equivalencia

## ❑ Elaborar los casos de prueba:

- Debemos construir casos de prueba mientras haya **clases de equivalencia válidas por cubrir**.
- Repetimos el proceso para las clases no válidas, cubriendo en cada caso de prueba **una y solo una clase inválida** a la vez, para evitar el **enmascaramiento** de errores.
- El resultado es una **tabla con los casos de prueba**:

Caso de Prueba	Clases de equivalencia	Condiciones de Entrada			Resultado esperado
		Día	Color	Nombre	
CP1	v_dia, v_color_r, v_mayúscula	3	red	User	R1
CP2	v_dia, v_color_g, v_mayúscula	1	green	USER	R2
CP3	v_dia, v_color_b, v_mayúscula	7	blue	UseR	R3
CP4	nv_dia_menor, v_color_r, v_mayúscula	0	red	User	E1
CP5	nv_dia_mayor, v_color_r, v_mayúscula	8	red	User	E2
CP6	nv_dia_menor, v_color_g, v_mayúscula	-1	green	USeR	E1
...	...	...	...	...	...

# Funcionales: clases de equivalencia

- Cuando elabores la tabla de **casos de prueba** trata de tener en cuenta lo siguiente:
  - Siempre que puedas, trata de **utilizar todos los valores límite** de tus clases válidas y si es posible de las inválidas.
  - Cubre mediante los **casos de prueba** válidos tantas clases de prueba como sea posible. Si eliges bien los valores, el número obligatorio puede ser el número máximo de casos de prueba necesarios.
  - **Nunca lo olvides:** los casos de prueba para clases no válidas únicamente pueden cubrir una de estas a la vez, si no puede que enmascare posibles errores.

# Funcionales: clases de equivalencia

- **Ejemplo:** Aplicación bancaria donde el operador debe aportar un código de área, un nombre de operación y una orden que disparará una serie de funciones bancarias
  - Especificación
    - **Código área:** número de 3 dígitos que no empieza por 0 ni por 1.
    - **Nombre de identificación:** 6 caracteres.
    - **Órdenes posibles:** cheque, depósito, pago factura, retirada de fondos.

# Ejemplo clases de equivalencia.

## □ Código área

### ▣ Número

- Clase válida: número
- Clase no válida: no número

### ▣ División de número

- Subclase válida (200-999)
- Dos subclases no válidas
  - Menor de 200
  - Mayor de 999

## □ Nombre operación

### ▣ Clase válida: 6 car.

### ▣ Clases no válidas

- Más de 6 car.
- Menos de 6 car.

## □ Orden

### ▣ Clase válida para cada orden: 4 en total.

### ▣ Clase no válida: cualquier otra cosa

# Ejemplo clases de equivalencia.

## 1. Clases de equivalencia

Condición de entrada	Regla a aplicar	Clases válidas	Clases no válidas
Código área	Condición booleana + rango de valores	(1) $200 \leq \text{código} \leq 999$	(2) Código < 200 (3) Código > 999 (4) No es número
Nombre de operación	Condición	(5) Seis caracteres	(6) Menos de 6 caracteres (7) Más de 6 caracteres
Orden	Conjunto de valores admitidos	(8) Cheque (9) Depósito (10) Pago factura (11) Retirada de fondos	(12) Ninguna orden válida

# Ejemplo clases de equivalencia.

## □ Casos de prueba válidos:

▣ 300	nómina	depósito.	(1) (5) (9)
▣ 400	viajes	cheque	(1) (5) (8)
▣ 500	coche	pago-factura	(1) (5) (10)
▣ 600	comida	retirada de fondos	(1) (5) (11)

## □ Casos no válidos.

▣ 180	Viajes	Pago-factura	(2) (5) (10)
▣ 1032	Nómina	Depósito	(3) (5) (9)
▣ XY	Compra	Retirada-fondos	(4) (5) (11)
▣ 350	A	Depósito	(1) (6) (9)
▣ 450	Regalos	Cheque	(1) (7) (8)
▣ 550	Casa	&%4	(1) (5) (12)



# Ejemplo clases de equivalencia.

## □ 2. Casos de prueba:

Caso de Prueba	Clases de equivalencia cubiertas	Condiciones de entrada Código – Clave - Orden
CP1	1-5-8	300 – Nómina – “cheque”
CP2	1-5-9	400 – Viajes – “depósito”
CP3	1-5-10	500 – Coches – “pago factura”
...		
CPN-1	2-5-10	150 –Viajes – “depósito”
CPN	1-6-9	220 – Luz – “cheque”
...		

# Ejercicio: obtención de los casos de prueba utilizando clases de equivalencia

- Se va a realizar una entrada de datos por teclado en el que se definen tres datos:
  - ✓ **Código de producto:** numérico de 200 a 999
  - ✓ **Proveedor:** se puede dejar en blanco o un número de dos dígitos
  - ✓ **Conservación:** uno de los siguientes: tipo1, tipo2, tipo3, tipo4
- Si la entrada es correcta, se le asigna un almacén atendiendo al tipo de conservación que precisa (frío, congelador, ...)
  - ✓ S1: si la conservación es de tipo1 o tipo2, se asigna almacén A001
  - ✓ S2: si la conservación es de tipo3, se asigna el almacén A002.
  - ✓ S3: si la conservación es de tipo4, se asigna el almacén A003.
- Si la entrada no es correcta, el programa muestra un mensaje indicando qué entrada es la incorrecta.
  - ✓ ER1: si no es un código de producto correcto.
  - ✓ ER2: si no es un proveedor correcto.
  - ✓ ER3: si no es una conservación correcta.

## Ejercicio: obtención de los casos de prueba utilizando las clases de equivalencia (solución)

- Determinar las clases de equivalencia, casos de prueba válidos y no válidos.

### Clases de equivalencia:

Condición de entrada	Clases de equivalencia	Clases válidas	Clases no válidas
Código del producto	Rango	(1) $200 \leq \text{Producto} \leq 999$	(8) $\text{Producto} < 200$ (9) $\text{Producto} > 999$ (10) No es número
Proveedor	Lógica (puede estar o no)	(2) En blanco	(11) No es un número
	Valor	(3) Cualquier número de dos dígitos	(12) Número de más de dos dígitos (13) Número de menos de dos dígitos
Conservación	Miembro de un conjunto	(4) Tipo1 (5) Tipo2 (6) Tipo3 (7) Tipo4	(14) Ningún tipo válido

## Ejercicio: obtención de los casos de prueba utilizando las clases de equivalencia (**solución**)

- A partir de la tabla anterior se genera la tabla de **casos de prueba**. Para ello se utilizan las condiciones de entrada y las clases de equivalencia, con el código asignado

CASO DE PRUEBA	Clases de equivalencia	CONDICIONES DE ENTRADA			Resultado esperado
		Código Producto	Proveedor	Conservación	
CP1	(1) (3) (4)	200	20	Tipo1	S1
CP2	(1) (2) (5)	250		Tipo2	S1
CP3	(1) (3) (6)	450	30	Tipo3	S2
CP4	(1) (2) (7)	220		Tipo4	S3
CP5	(1) (2) (6)	350		Tipo3	S2
CP6	(1) (3) (5)	400	40	Tipo2	S1
CP7	(8) (3) (6)	90	35	Tipo3	ER1
CP8	(1) (11) (5)	220	CC	Tipo2	ER2
CP9	(1) (2) (14)	300		Elige conservación	ER3
CP10	(1) (12) (6)	345	123	Tipo3	ER2
CP11	(9) (2) (4)	1000		Tipo1	ER1
CP12	(1) (13) (6)	300	1	Tipo3	ER2
...	...	...	...	...	...

## Ejercicio: obtención de los casos de prueba utilizando las clases de equivalencia (solución)

- Para rellenar la tabla de casos de prueba, se tiene que tener en cuenta que se tienen que cubrir todas las clases válidas en cada uno de los casos de prueba válidos. En el caso de los casos de prueba no válidos, cada uno de ellos cubre una única clase no válida para no enmascarar que finalice por causas ajenas a los datos no válidos. Por lo que faltarían los casos de prueba válidos (1)(2)(4), (1)(3)(7), y el caso de prueba no válido (10)(3)(4).

# Funcionales: análisis de Valores Límite

- ▣ Esta técnica **complementa** la técnica de las particiones de equivalencia.
- ▣ Se basa en el hecho heurístico de que **los errores tienden a producirse con más probabilidad en los límites** o extremos de los valores de entrada.
- ▣ Los valores elegidos para los casos de prueba son aquellos que están por encima o por debajo de los márgenes de las clases de equivalencia.
  - ▣ En lugar de diseñar una clase válida con un elemento representativo de la partición equivalente, se escogen los valores en los límites de la clase.
- ▣ Los valores límite no solo se exploran en los datos de entrada, sino también en las **condiciones de salida**.

# Funcionales: análisis de Valores Límite

- ▣ En cada prueba se elige los valores que se encuentran **justo por encima** y por **debajo** de los márgenes de la clase de equivalencia.
- ▣ El análisis se concentra sobre la clase de equivalencia válida, identificando:
  - ▣ El valor límite
  - ▣ El valor por debajo del límite
  - ▣ El valor por encima del límite

# Funcionales: análisis de Valores Límite

## ▣ Valores límite con un rango de valores

- Si la **condición de entrada** especifica un **rango de valores** definido como
$$n \leq i \leq m$$

- los casos de prueba resultantes corresponden a  $n-1, n, n+1, m-1, m$  y  $m+1$

dando lugar a 4 clases válidas y 2 no válidas.

- **Ejemplo** para  $1 \leq i \leq 99$ :
  - Los valores a testear son: 0, 1, 2, 98, 99 y 100 (2 y 98 son opcionales)
  - Obtenemos **4 clases válidas**: para los extremos (1, 99) y para los valores que se sitúan junto al límite (2 y 98, que son opcionales)
  - Obtenemos **2 clases no válidas**: para los valores más allá de los extremos (0 y 100).



# Funcionales: análisis de Valores Límite

## ▣ **Valores límite en entradas con varios valores**

- Si la **condición de entrada** especifica **un número de valores** (ej.- de 1 a 6 caracteres):
  - Valores a testear: valores máximo y mínimo (1 y 6), y los valores por encima del máximo y por debajo del mínimo (0 y 7)
  - **2 clases válidas:** para los valores mínimo y máximo (1, 6)
  - **2 clases no válidas:** para los valores por encima de máximo (7) y otro por debajo del mínimo (0)

# Funcionales: análisis de Valores Límite

## ▣ Valores límite en los datos de salida:

- Se debe aplicar las dos reglas anteriores a los **datos de salida**.
- Por ej.- si un programa obtiene como salida la nota media de un alumno y debe estar entre 0 y 10, se debería usar el análisis de valores límite con el fin de crear casos de prueba que generen los siguientes valores como notas medias: -1,0,1, 9,10 y 11
- Por ej.- si la salida del programa es una tabla de temperaturas de 1 a 10 elementos, se deben diseñar casos de prueba para que la salida produzca 0,1, 10 y 11 elementos. Tanto en esta regla como en la anterior, se debe tener en cuenta que no siempre se podrá generar resultados fuera del rango de salida.
- Aplicar reglas para estructuras tipo array, con límites establecidos, es decir, diseñar casos de prueba que sobre los límites del array.

# Funcionales: análisis de Valores Límite

- **Ejemplo:** siguiendo con el ejemplo anterior de la aplicación bancaria.

Condición de entrada	Regla a aplicar	Clases válidas	Clases no válidas
Código área	Condición booleana + rango de valores [200...999]	(1) código = 200 (límite) (2) código = 201 (opcional) (3) código = 999 (límite) (4) código = 998 (opcional)	(5) código = 199 (valor por debajo) (6) código = 1000 (valor por encima)
Nombre de operación	Conjunto finito de valores	(7) seis caracteres	(8) cinco caracteres (9) siete caracteres
Orden	Conjunto de valores admitidos	(10) Cheque (11) Depósito (12) Pago factura (13) Retirada de fondos	(14) Ninguna orden válida

Ahora definimos los **casos de prueba** para cubrir todas las clases válidas e inválidas (igual que antes)

# Funcionales: análisis de Valores Límite

- **Ejercicio:** aplicamos esta técnica al ejercicio anterior.

Condición de entrada	Regla a aplicar	Clases válidas	Clases no válidas
Código producto	rango de valores [200...999]	(1) código = 200 (límite) (2) código = 201 (opcional) (3) código = 999 (límite) (4) código = 998 (opcional)	(5) código = 199 (valor por debajo) (6) código = 1000 (valor por encima)
Proveedor	Lógica + valor	(7) 2 dígitos	(8) 1 dígito (9) 3 dígitos
Conservación	Conjunto de valores admitidos	(10) Tipo1 (11) Tipo2 (12) Tipo3 (13) Tipo4	(14) Ninguna orden válida

Ahora definimos los **casos de prueba** para cubrir todas las clases válidas e inválidas (igual que antes)

# Funcionales. Análisis de Valores Límite

## Algunas recomendaciones al elegir valores de límite:

- Variable flotante  $[-1.0 \text{ a } 1.0]$   $\rightarrow$  probar  $-1.0$ ,  $1.0$ ,  $-1.001$  y  $1.001$ .
- Variable entera  $[10 \text{ a } 100]$   $\rightarrow$  probar  $9$ ,  $10$ ,  $100$ ,  $101$ .
- letra en mayúsculas  $\rightarrow$  probar de la A a la Z de límite. Probar también  $@$  y  $[$  puesto que, en el código ASCII,  $@$  está junto debajo de la A y  $[$  está justo después de la Z.
- La entrada es un conjunto ordenado, conviene prestar atención al primer y al último elemento del conjunto.
- La suma de las entradas debe ser un número específico ( $n$ ), probar el programa donde la suma sea  $n-1$ ,  $n$ , o bien,  $n+1$ .
- El programa acepta una lista, probar los valores de la lista. Todos los demás valores no son válidos.
- Al leer o escribir un archivo, comprobar los caracteres primero y último del archivo.
- Para una variable con varios rangos, cada rango es una clase de equivalencia. Si los subrangos no están solapados, probar los valores de los límites, después del límite superior y debajo del límite inferior.

# Funcionales. Análisis de Valores Límite

## Valores especiales

- Un verificador con experiencia puede observar las entradas de programa para descubrir todos los casos de "**valores especiales**" que, una vez más son, potencialmente, fuentes valiosas para revelar anomalías de software. Estos son algunos ejemplos:
  - Para un tipo de entero, se debe probar siempre **cero** si se encuentra en la clase de equivalencia válida.
  - Al probar la hora (hora, minuto y segundo), se debe probar siempre **59** y **0** como el límite superior e inferior para cada campo, sin tener en cuenta la restricción que tenga la variable de entrada. Así, excepto los valores de límite de la entrada, -1, 0, 59 y 60 siempre deben ser casos de prueba.
  - Al probar la **fecha** (año, mes y día), se deben incluir numerosos casos de prueba como, por ejemplo, un número de días en un mes específico, el número de días de febrero en año bisiesto o el número de días en años no bisiestos.

# Funcionales. Análisis de Valores Límite

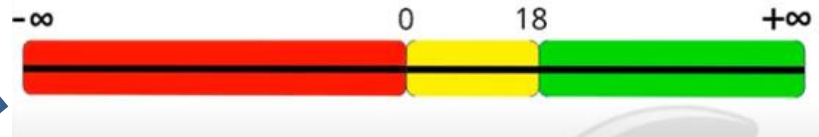
	Condiciones de entrada y salida	Casos de prueba
Código	Entero de 1 a 100	Valores: 0, 1, 100, 101
Puesto	Alfanumérico de hasta 4 caracteres	Longitud de caracteres: 0, 1, 4, 5
Antigüedad	De 0 a 25 años (Real)	Valores: 0, 25, -0.1, 25.1
Horas semanales	De 0 a 60	Valores: 0, 60, -1, 61
Fichero de entrada	Tiene de 1 a 100 registros	Para leer 0,1, 100, 101
Fichero de salida	Podrá tener de 0 a 10 registros	Para generar 0, 10 y 11 registros (no se puede generar -1 registro)
Array interno	De 20 cadenas de caracteres	Para el primer y último elemento

# Funcionales. Análisis de Valores Límite

```
public class Persona {  
    private int edad;  
  
    public Persona(int edad) {  
        this.edad = edad;  
    }  
  
    public boolean isMayorDeEdad() {  
        assert edad >= 0;  
        return edad >= 18;  
    }  
}
```

## ▣ Otro Ejemplo.

- **Función:** `isMayorDeEdad()`
- Clases de equivalencia válidas:
  - $[18, +\infty)$  → resultado true
  - $[0, 18)$  → resultado false
- Clase de equivalencia no válida:
  - $(-\infty, 0)$  → no tiene sentido
- Valores límite:  $-100, -1, 0, 17, 18, 100$   
Atributo: edad



## Clases de equivalencia

Atributo: edad				
Dominio	Clase	Tipo	Límite inferior	Límite superior
Números enteros	$(-\infty, 0)$	NO VÁLIDA	$-\infty$	-1
	$[0, 18)$	VÁLIDA	0	17
	$[18, +\infty)$	VÁLIDA	18	$+\infty$

## Casos de prueba

isMayorDeEdad		
Nº	edad	Salida esperada
1	-1	ERROR
2	-100	ERROR
3	0	false
4	17	false
5	18	true
6	100	true



# Funcionales. Análisis de Valores Límite

- Esta técnica permite extender los casos de prueba obtenidos con la técnica de las particiones equivalentes y hacer hincapié en los límites operacionales del módulo a probar, donde se concentran la mayoría de errores.

# Funcionales. Pruebas aleatorias

- Consiste en generar entradas aleatorias para la aplicación que hay que probar.
- Se suelen utilizar herramientas (generadores automáticos de pruebas), capaces de crear un volumen de casos de prueba al azar.
- Se suelen utilizar en aplicaciones no interactivas.

## 5. Pruebas estructurales

# Pruebas Estructurales.

- Conjunto de pruebas de la **caja blanca**.
- Se pretende verificar la estructura interna de cada componente.
- **Función:** obtener casos de prueba que comprueben
  - ▣ que se van a ejecutar todas la instrucciones del programa,
  - ▣ no hay código no usado,
  - ▣ se ejecutan todas las condiciones a true y a false,
  - ▣ que los caminos lógicos del programa se van a recorrer, etc.
- Es impracticable probar todos los caminos de un programa → este tipo de pruebas se basan en **criterios de cobertura lógica**, que permiten decidir qué sentencias o caminos se deben examinar en los casos de prueba.

# Estructurales. Criterios de cobertura

## □ Criterios de cobertura:

### ▣ Cobertura de sentencias.

- Generar casos de pruebas suficientes para que cada instrucción del programa sea ejecutada, al menos, una vez.

### ▣ Cobertura de decisiones:

- Crear los suficientes casos de prueba para que cada opción resultado de una prueba lógica del programa, se evalúe al menos una vez a cierto y otra a falso.

### ▣ Cobertura de condiciones:

- Cada elemento de una condición de una decisión se evalúa al menos una vez a falso y otra a verdadero.

```
if (numero>0 && repetir==true)
```

- Condiciones:

- num>0
- repetir==true

- Decisiones

- if (true or false)

Decisión = condición1 + condición2

# Estructurales. Criterios de cobertura (cont)

- **Cobertura de condiciones y decisiones:**

- Consiste en cumplir simultáneamente las dos anteriores.

- **Cobertura de caminos:** (es el más importante)

- Ejecutar todos los caminos de un programa → Ejecutar al menos una vez cada sentencia del programa, desde la sentencia inicial, hasta su sentencia final y evaluar todas las condiciones tanto a V como a F.
- La ejecución de un conjunto de sentencias se denomina **camino**.
- Para realizar esta prueba se **reduce el número de caminos** a lo que se conoce como **camino de prueba**.

- Una técnica empleada para aplicar el criterio de cobertura de caminos es la **Prueba del Camino Básico**.

# Estructurales. Prueba del camino básico.

- Técnica de prueba de caja blanca
- Esta técnica se basa en obtener **una medida de la complejidad lógica** del diseño procedimental de un programa denominada **Complejidad ciclomática de McCabe**, que representa el límite superior para el número de **casos de prueba** que se deben realizar para asegurar que se ejecuta cada camino del programa, es decir, que se ejecutan todas las sentencias al menos una vez.
- Para la obtención de la **complejidad lógica** o **complejidad ciclomática** se usa un **grafo de flujo** o grafo del programa.

# Estructurales. Prueba del camino básico.

- Pasos a desarrollar en la técnica prueba del camino básico:
  - Numerar las sentencias de código
  - Obtener el **grafo de flujo** a partir del paso anterior
  - Calcular la **complejidad ciclomática** o  $V(G)$  a partir del grafo anterior.
  - Determinar los **caminos independientes** del grafo.
  - Obtener los casos de prueba de cada camino.



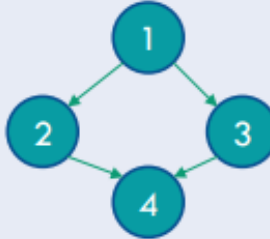

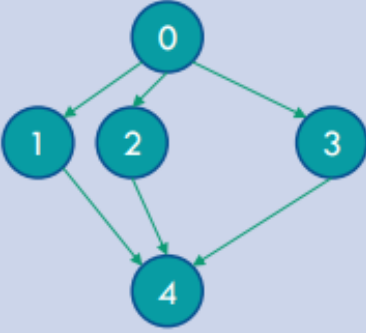
[Video: Prueba del camino básico](#)



# Estructurales. Prueba del camino básico.

- **Grafo de flujo:** representa el programa de modo que se observan los puntos donde existen cambios de dirección (debidos a un salto, el fin de de una iteración o la evaluación de una condición) permitiendo analizar los diferentes caminos de ejecución desde el inicio al fin del programa.
- El grafo de flujo contiene los siguientes elementos:
  - **Nodos:** representan cero, una o varias instrucciones secuenciales. Las instrucciones secuenciales (asignación, E/S) pueden representarse cada una en un nodo o varias de ellas en un solo nodo.
  - **Aristas:** líneas que unen dos nodos y representan cambios de dirección.
  - **Regiones:** áreas delimitadas por aristas y nodos (área encerrada entre aristas y nodos). Cuando se contabilizan las regiones de un programa debe incluirse el área externa como una región más.
  - **Nodos predicado:** nodo del que parten dos o más caminos.
- Se recomienda incluir un nodo inicial y final únicos.

# Estructurales. Prueba del camino básico.

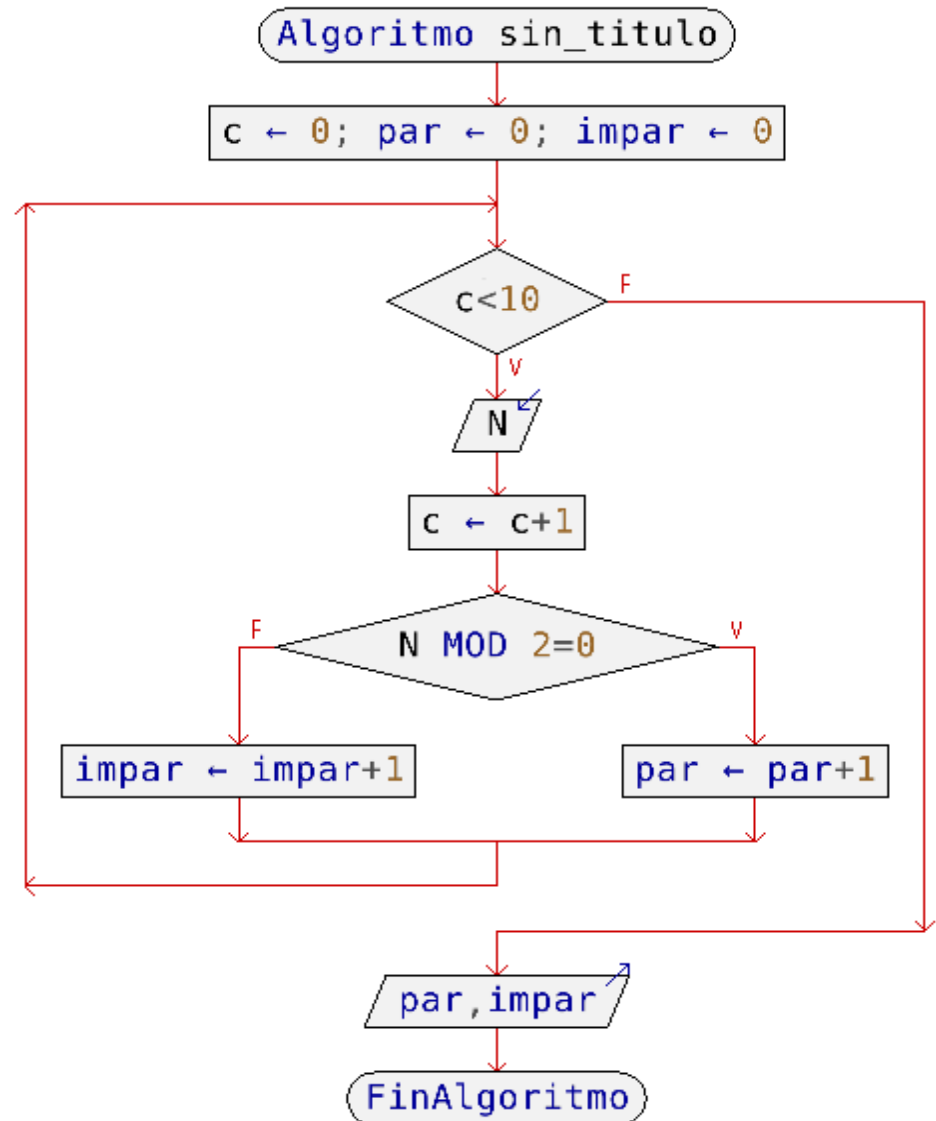
Estructura	Grafo de flujo	Estructura	Grafo de flujo
<b>Secuencial</b> instrucción1 instrucción2 ... instrucciónN	 <pre> graph TD     1((1)) --&gt; 2((2)) </pre>	<b>Hacer mientras</b> mientras <condición1> <inst2> fin mientras3	 <pre> graph TD     1((1)) --&gt; 2((2))     2 --&gt; 3((3))     3 --&gt; 1 </pre>
<b>Condicional</b> si <condición1> entonces <instruc2> si no <instruc3> fin si4	 <pre> graph TD     1((1)) --&gt; 2((2))     1 --&gt; 3((3))     2 --&gt; 4((4))     3 --&gt; 4 </pre>	<b>Repetir hasta</b> Repetir <inst1> Hasta que <condición2>	 <pre> graph TD     1((1)) --&gt; 2((2))     2 --&gt; 3((3))     3 --&gt; 1 </pre>
<b>Condición múltiple</b> según sea <variable0> hacer caso opción1: <inst1> caso opción2: <inst2> otro caso3: <inst3> fin según4	 <pre> graph TD     0((0)) --&gt; 1((1))     0 --&gt; 2((2))     0 --&gt; 3((3))     1 --&gt; 4((4))     2 --&gt; 4     3 --&gt; 4 </pre>	Videos: <a href="#">Grafo de flujo de un programa</a> <a href="#">Grafo de flujo de un programa</a>	

# Estructurales. Prueba del camino básico.

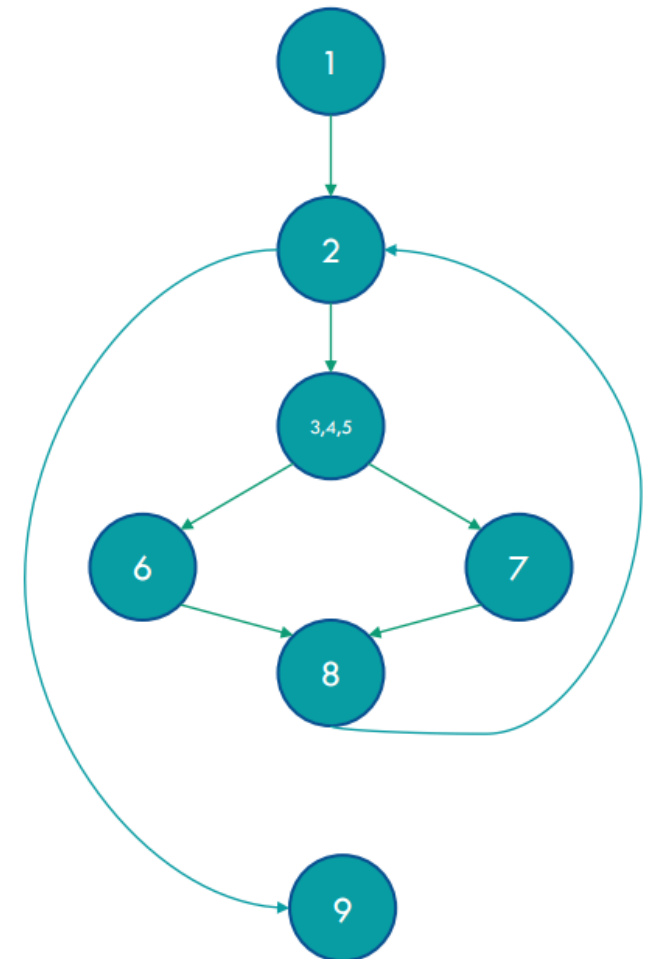
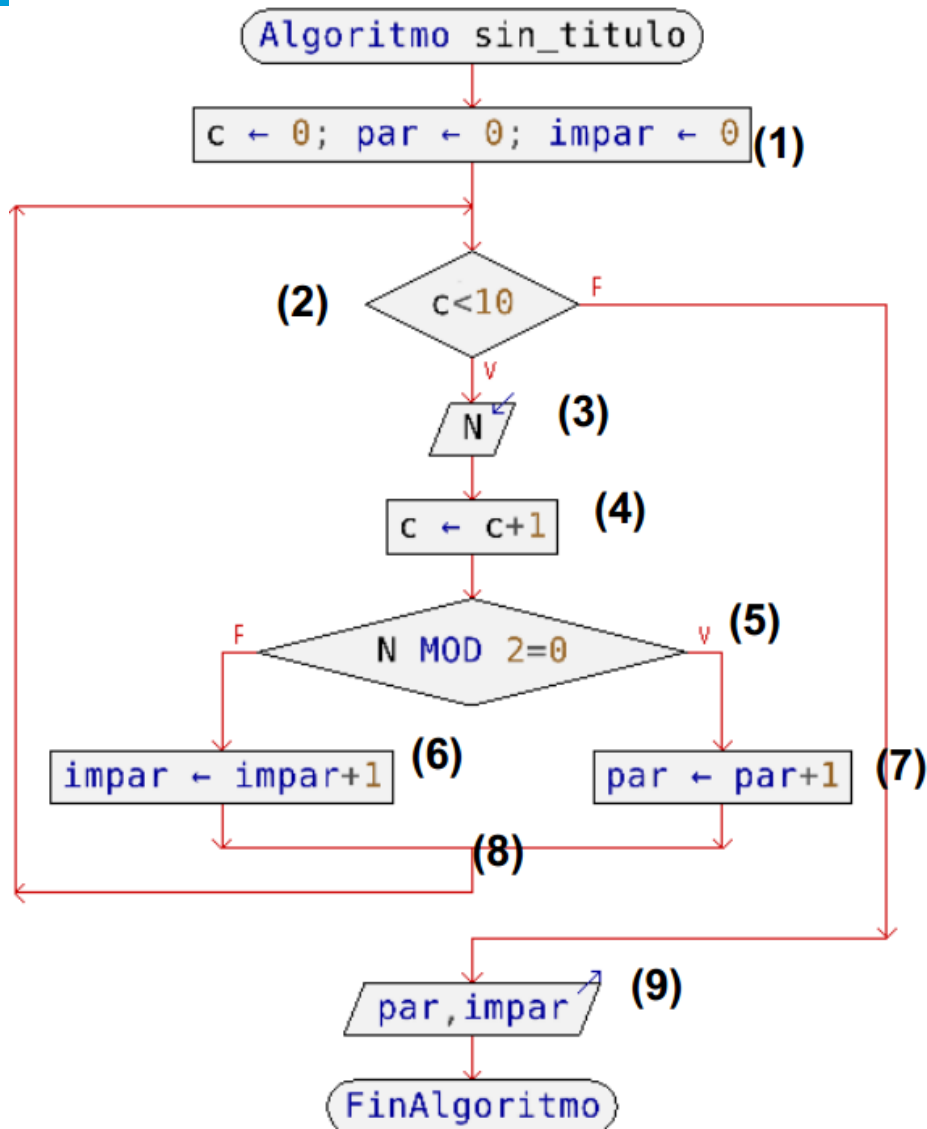
- **Paso 1.** Se numera cada sentencia en el código , teniendo en cuenta que si una sentencia tiene dos o más condiciones, se numerará cada una de las condiciones por separado. Es decir, si la sentencia es:  
$$\text{if } (a > 0 \ \&\& \ b > 0) \rightarrow (a > 0) \text{ se numera como } 1$$
$$(b > 0) \text{ se numera como } 2$$
- Se tendrán en cuenta los finales de las estructuras de control (if, while ...)
- **Paso 2.** Construcción del grafo

# Estructurales. Prueba del camino básico.

- Ejercicio 1. Obtener el grafo de flujo a partir del ordinograma siguiente:

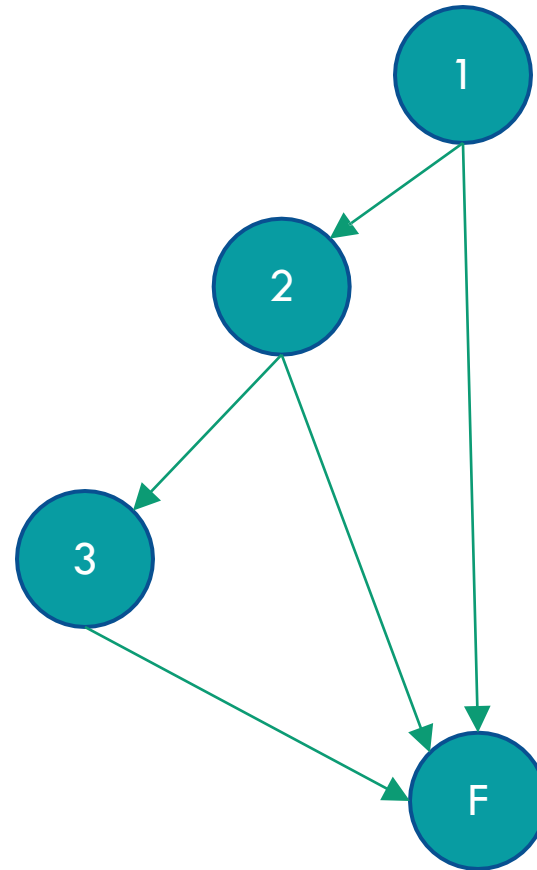


# Estructurales. Prueba del camino básico.



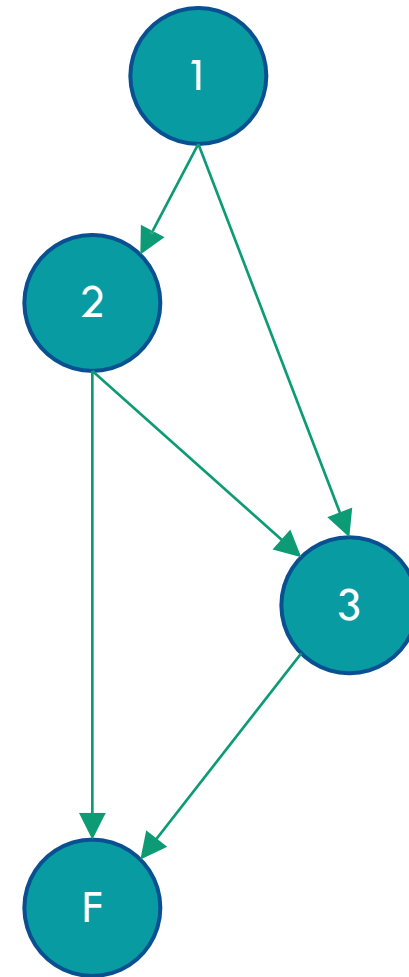
# Ejemplo de lógica compuesta: and

① si ② (a and b)  
③ sentencias1  
F fin si



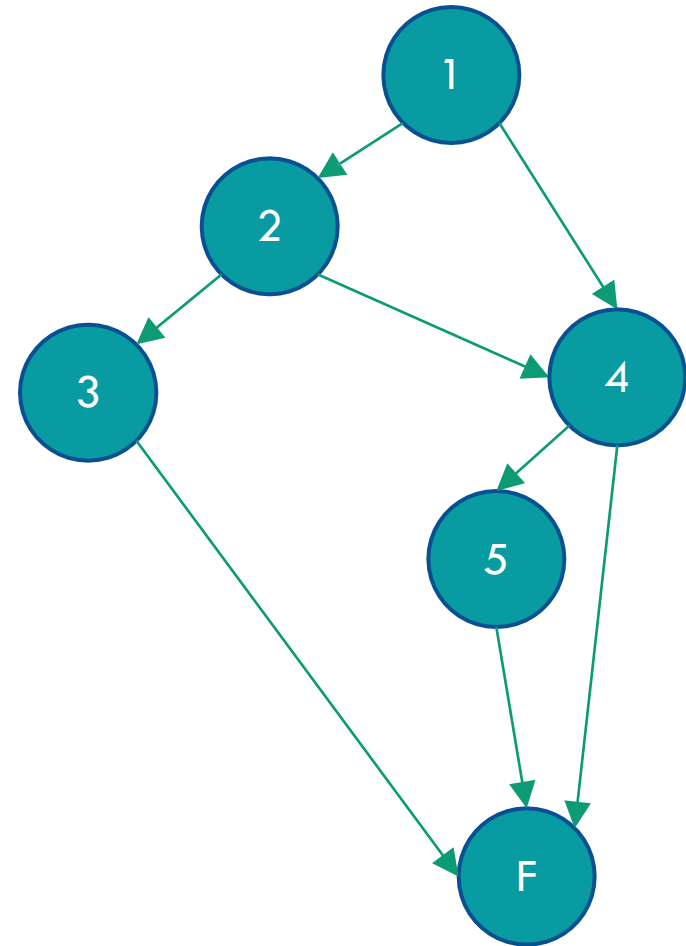
# Ejemplo de lógica compuesta: or

① si ② (a or b)  
③ sentencias1  
F fin si



# Ejemplo de lógica compuesta

① si ② (a and b)  
③ sentencias1  
si no  
④ si c  
⑤ sentencias2  
fin si  
F fin si





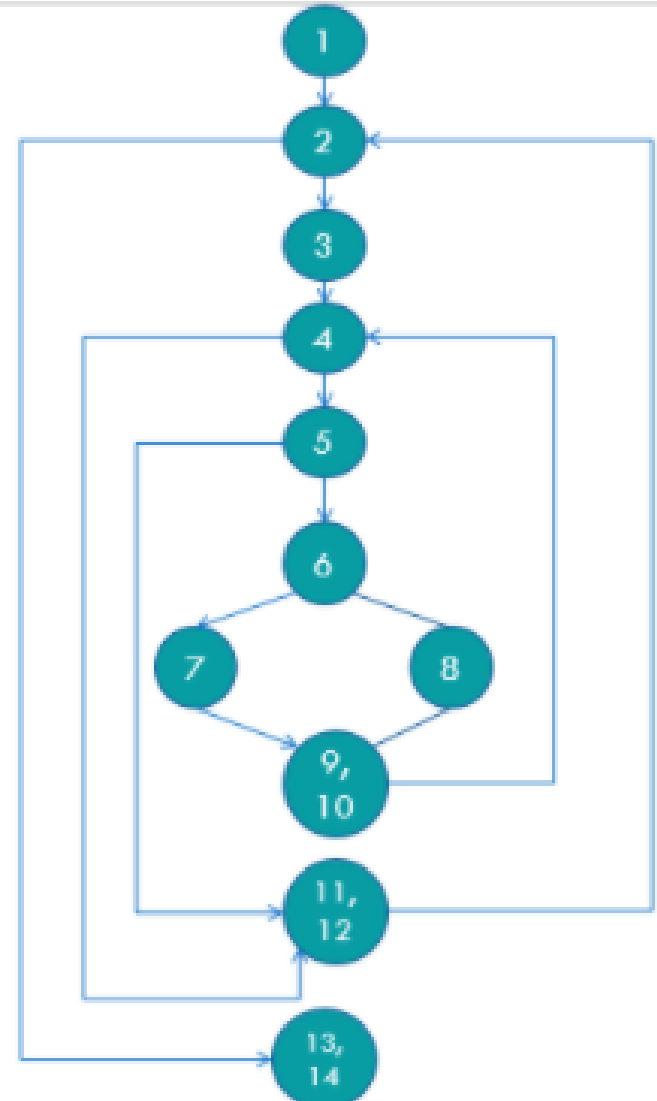
# Estructurales. Prueba del camino básico.

- Ejercicio 2. Obtener el grafo de flujo a partir del pseudocódigo siguiente:

```
Abrir archivo ventas;  
Leer venta (producto, tipo_venta, total);  
MIENTRAS (haya registros) HACER  
    total_nacional=0;  
    total_extranjero=0;  
    MIENTRAS (haya registros) Y (mismo producto)  
        SI (tipo_venta="nacional") ENTONCES  
            total_nacional= total_nacional+ total;  
        SI NO  
            total_extranjero=total_extranjero+ total;  
        FIN SI;  
    Leer venta (producto, tipoVenta, total);  
    FIN MIENTRAS;  
    Escribir Producto, total_nacional, total_extranjero;  
FIN MIENTRAS;  
Cerrar archivo ventas;
```

# Estructurales. Prueba del camino básico.

```
Abrir archivo coches; (1)
Leer venta (modelo, año_fabricacion); (2)
MIENTRAS (haya registros) HACER (2)
    masde10=0; (3)
    menosde10=0; (4) (5)
    MIENTRAS (haya registros) Y (mismo modelo) (6)
        SI (año_fabricación >= 2007) (6) ENTONCES
            menosde10 = menosde10 + 1; (7)
        SI NO
            masde10 = masde10 + 1; (8)
        FIN SI; (9)
    Leer venta (producto, tipoVenta, total); (10)
    FIN MIENTRAS; (11)
    Escribir Modelo, menosde10, masde10; (12)
FIN MIENTRAS; (13)
Cerrar archivo coches; (14)
```



# Estructurales. Prueba del camino básico.

## ▣ Paso 3. Cálculo de la **complejidad ciclomática** de McCabe $v(G)$ :

- ▣ Métrica del software que proporciona una medida cuantitativa de la complejidad lógica de un programa.
- ▣ Determina el número de **caminos independientes** de la ejecución de un programa, por lo que también **determina el número de casos de prueba que se deben ejecutar** para asegurar que cada sentencia se ejecuta al menos una vez.
- ▣  **$V(G)$  se calcula de tres formas:**
  - $V(G) = \text{numero de aristas} - \text{numeros nodos} + 2 = a - n + 2$
  - $V(G) = \text{número de regiones cerradas del grafo} = r$
  - $V(G) = \text{número de nodos de condición} + 1 = c + 1$

[Video: Cálculo complejidad ciclomática](#)

# Estructurales. Prueba del camino básico.

- Ejercicio 1: calcula el número de caminos a través de la complejidad ciclomática de McCabe:

- $V(G) = a - n + 2 = 8 - 7 + 2 = 3;$

- $V(G) = r = 3;$

- $V(G) = c + 1 = 2 + 1 = 3$

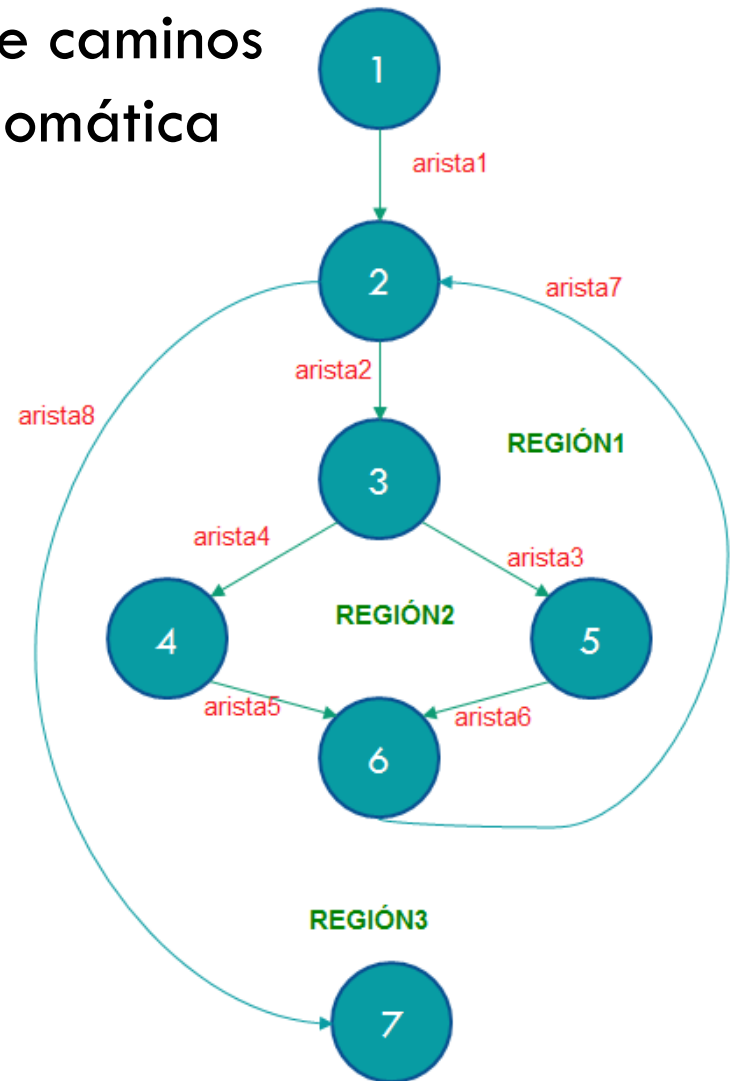
- Son 3 el número de caminos indep.

- Camino1: 1 2 7

- Camino2: 1 2 3 4 6 2 7

- Camino3: 1 2 3 5 6 2 7

- Casos de prueba para los caminos  
= 3



# Estructurales. Prueba del camino básico.

- ▣ Ejercicio2: calcula el número de caminos a través de la complejidad ciclomática de McCabe:

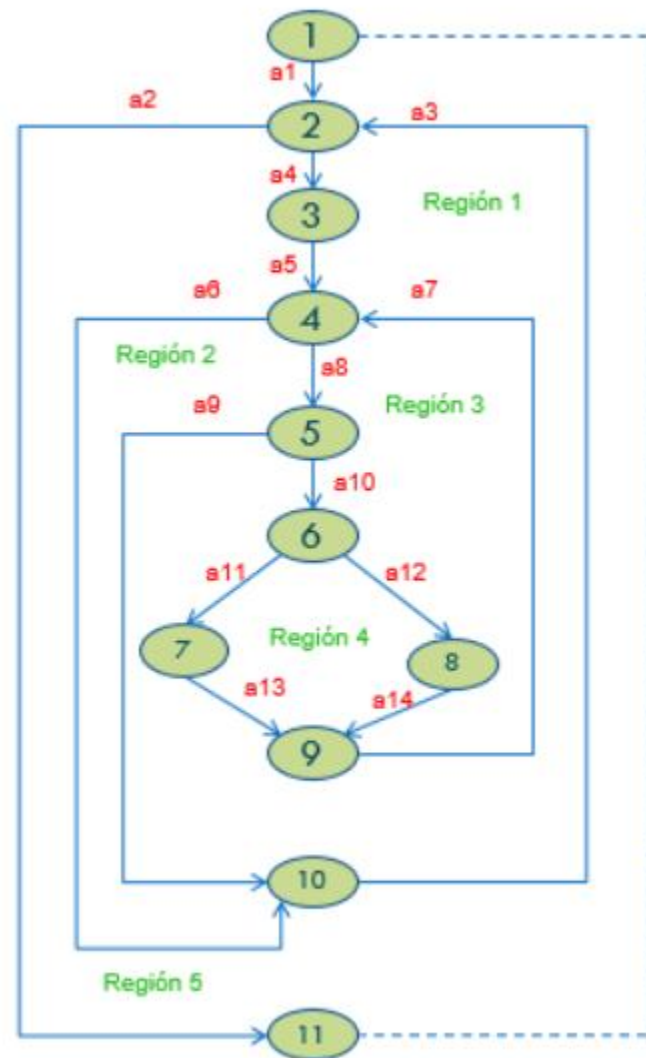
$$V(G) = 14 - 11 + 2 = 5.$$

$$V(G) = \text{Número de regiones} = 5$$

$$V(G) = 4 + 1 = 5 \text{ (Nodos de condición: 2, 4, 5, 6)}$$

Posible conjunto de caminos:

- 1-2-11
- 1-2-3-4-10-2-11
- 1-2-3-4-5-10-2-11
- 1-2-3-4-5-6-7-9-4-10-2-11
- 1-2-3-4-5-6-8-9-4-10-2-11



# Estructurales. Prueba del camino básico.

- Se establecen los siguientes valores de la complejidad ciclomática.

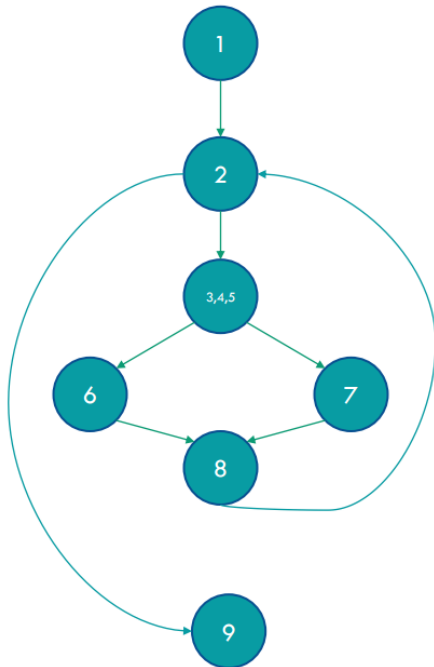
Complejidad ciclomática	Evaluación de riesgo
Entre 1 y 10	Programas o métodos sencillos, sin mucho riesgo
Entre 11 y 20	Programas o métodos más complejos, riesgo moderado
Entre 21 y 50	Programas o métodos complejos, alto riesgo
Mayor que 50	Programas o métodos no testeables, muy alto riesgo

# Estructurales. Prueba del camino básico.

- **Paso 4.** Una vez obtenidos los **caminos independientes**, podemos determinar el conjunto de casos de prueba que ejecutan cada uno de los caminos (uno por cada camino).
- Se trata de escoger los casos de prueba de forma que las condiciones de los nodos predicado estén adecuadamente establecidas.

# Estructurales. Prueba del camino básico.

## □ Paso 5. Obtención de los casos de prueba:



Camino1: 1 2 9

Camino2: 1 2 3 4 5 6 8 2 9

Camino3: 1 2 3 4 5 7 8 2 9

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



## 6. Pruebas Regresión

# Regresión

- Durante el proceso de prueba se tiene **éxito** si se detectan fallos o errores → supone **cambios** en el componente analizado → supone posibles **errores colaterales** que no existían antes.
- **Consecuencia:** repetición de las pruebas realizadas.
- **Objetivo** de las pruebas de regresión es comprobar que los cambios realizados sobre un componente no introducen un comportamiento no deseado o errores adicionales en otros componentes no modificados.
- Las pruebas de regresión se pueden hacer manualmente o con herramientas automáticas.
  - Estas pruebas se deben realizar cada vez que se haga un cambio en el sistema, tanto para corregir un error, como para realizar una mejora → no es suficiente probar solo los componentes modificados o añadidos, sino que hay que controlar que las modificaciones no produzcan efectos negativos sobre el mismo u otros componentes.

# Regresión

- El conjunto de pruebas de regresión contiene tres clases diferentes de clases de prueba:
  - ▣ Una **muestra representativa** de pruebas que ejercite todas las funciones del software;
  - ▣ **Pruebas adicionales** que se centran en las funciones del software que se van a ver probablemente afectadas por el cambio;
  - ▣ **Pruebas** que se centran en los componentes del software que han **cambiado**.
- Para evitar que el número de pruebas de regresión crezca demasiado, se deben de diseñar para incluir sólo aquellas pruebas que traten una o más clases de errores en cada una de las funciones principales del programa. No es práctico ni eficiente volver a ejecutar cada prueba de cada función del programa después de un cambio.