

**Responsable del Tratamiento:** Universitat Politècnica de València (UPV)

**Finalidad:** Prestación del servicio público de educación superior en base al interés público de la UPV (Art. 6.1.e del RGPD).

**Ejercicio de derechos y segunda capa informativa:** Podrán ejercer los derechos reconocidos en el RGPD y la LOPDGDD de acceso, rectificación, oposición, supresión, etc., escribiendo al correo [dpd@upv.es](mailto:dpd@upv.es).

Para obtener más información sobre el tratamiento de sus datos puede visitar el siguiente enlace: <https://www.upv.es/contenidos/DPD>.

**Propiedad Intelectual:** Uso exclusivo en el entorno del aula virtual.

Queda prohibida la difusión, distribución o divulgación de la grabación de las clases y particularmente su compartición en redes sociales o servicios dedicados a compartir apuntes.

La infracción de esta prohibición puede generar responsabilidad disciplinaria, administrativa y/o civil.

# PRUEBAS

## Tema 8. Pruebas. Técnicas del camino básico y de la partición equivalente

Ingeniería del Software

ETS Ingeniería Informática

DSIC – UPV

# Objetivos

- Conocer las técnicas básicas de prueba de programas
- Ser capaces de diseñar casos de prueba para un módulo o función utilizando las técnicas del **camino básico** y de la **partición equivalente**.

# Contenidos

- Introducción a la prueba de software.
- Técnicas de diseño de casos de prueba.
- Prueba de caja blanca: el camino básico.
- Prueba de caja negra: la partición equivalente.
- Herramientas automáticas de prueba.

# Bibliografía

- SOMMERVILLE, I., Software Engineering, 7ª Edición. Addison Wesley, 2005
- PFLEEGER, S. L., Ingeniería del Software: Teoría y Práctica. Prentice Hall, 2002.
- PRESSMAN, R. Ingeniería del software. Un enfoque práctico. 6ª Edición, McGraw-Hill, 2006.
- COLLARD, J.F, BURNSTEIN, I, Practical Software Testing: A Process-Oriented Approach, Springer. 2003
- EVERETT, D., McLEOD, R. Software Testing. Testing Across the Entire Software Development Life Cycle, IEEE Press
- BEIZIER, B., Testing and quality assurance, von Nostrand Reinhold, New York, 1984

# INTRODUCCION

---

# Introducción

- **Prueba de Software:** actividad en la cual un sistema o uno de sus componentes se ejecuta en circunstancias previamente especificadas, los resultados se observan y registran, y se realiza una evaluación de algún aspecto: corrección, robustez, eficiencia, etc.
- **Caso de prueba** (test case): un conjunto de entradas, condiciones de ejecución y resultados esperados desarrollados para un objetivo particular

# Principios básicos

1. **Testing** es el proceso de ejecutar un componente software utilizando un conjunto básico de casos de prueba, con la intención de (i) revelar defectos, y (ii) evaluar la calidad
2. Cuando el objetivo de la prueba es detectar defectos, entonces un buen caso de pruebas será aquel con una mayor probabilidad de detectar un defecto todavía no detectado
3. Los resultados de las pruebas deberían ser meticulosamente inspeccionados
4. Un caso de pruebas tiene que contener los resultados esperados
5. Los casos de prueba deberían ser definidos tanto para condiciones de entrada válidas como no válidas

# Principios básicos

6. La probabilidad de la existencia adicional de defectos en un componente software es proporcional al número de defectos ya encontrados en ese componente
7. Las pruebas deberían ser llevadas a cabo por un grupo que sea independiente del grupo de desarrollo
8. Las pruebas tienen que ser repetibles y reutilizables
9. Las pruebas deben ser planeadas
10. Las actividades de testing deberían estar integradas con el resto del ciclo de vida
11. Testing es una actividad creativa y desafiante



# Visión Google de las pruebas



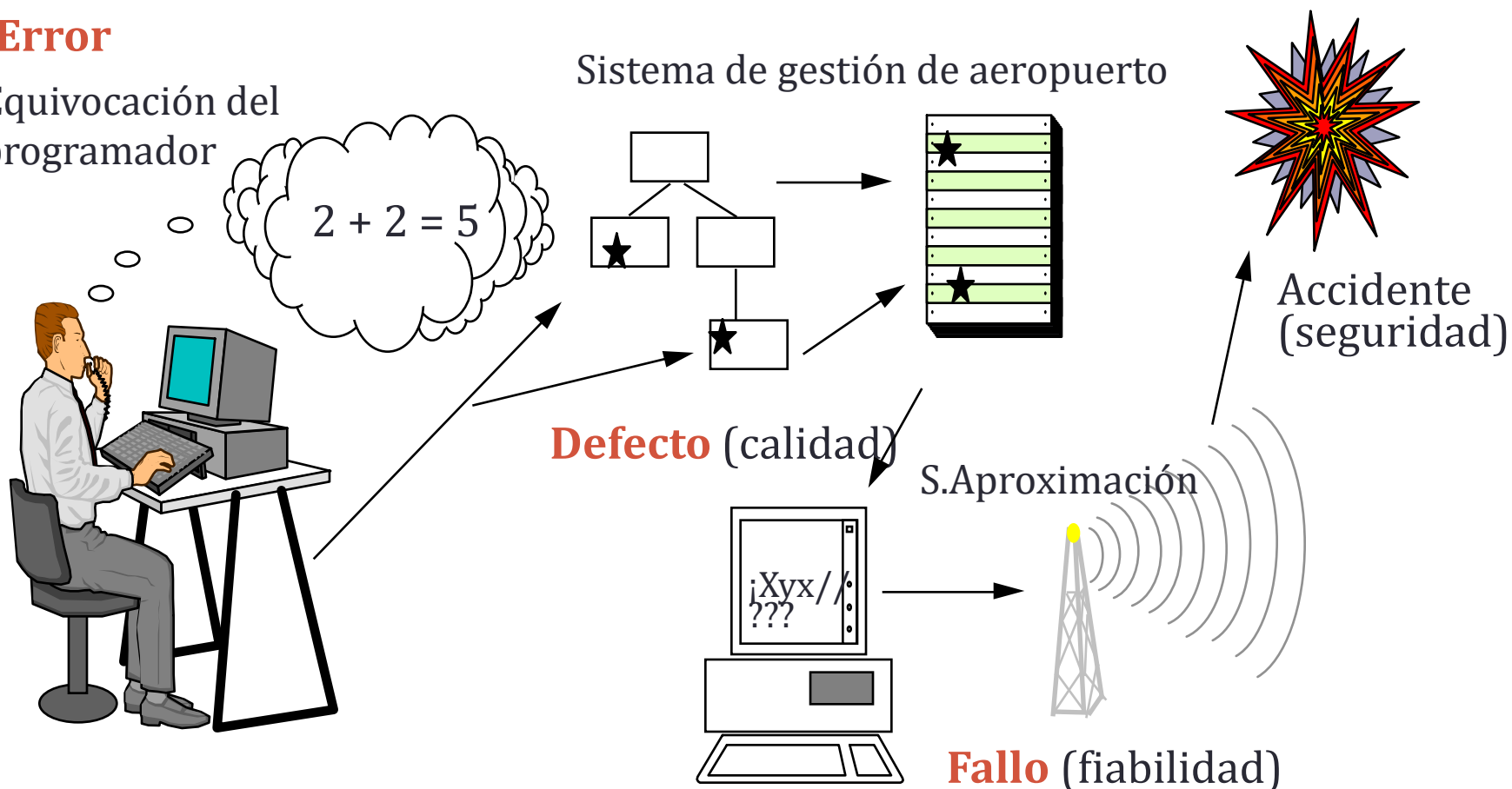
# Definiciones

- **Verificación:** ¿estamos construyendo correctamente el producto? ¿el software se ha construido de acuerdo a las especificaciones?
- **Validación:** ¿estamos construyendo el producto correcto? ¿el software hace lo que el usuario realmente requiere?
- **Error** (de programación): acción humana que conduce a un resultado incorrecto
- **Defecto** (bug): una anomalía en el software como, por ejemplo, un proceso, una definición de datos o un paso de procesamiento incorrectos en un programa
- **Fallo** (failure): cuando el sistema, o alguno de sus componentes, es incapaz de realizar las funciones requeridas dentro de los rendimientos especificados

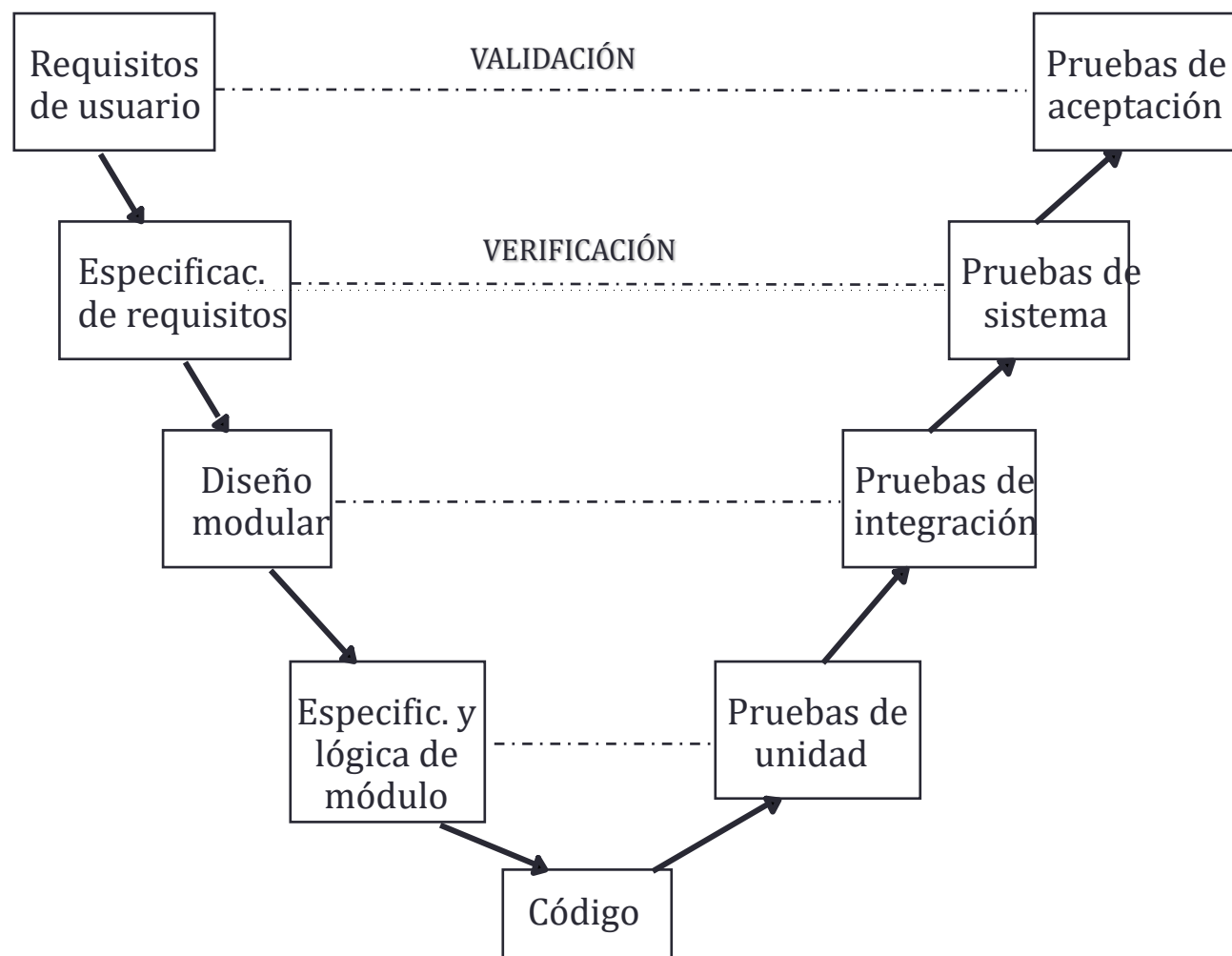
# Relación entre error, defecto y fallo

## Error

Equivocación del programador



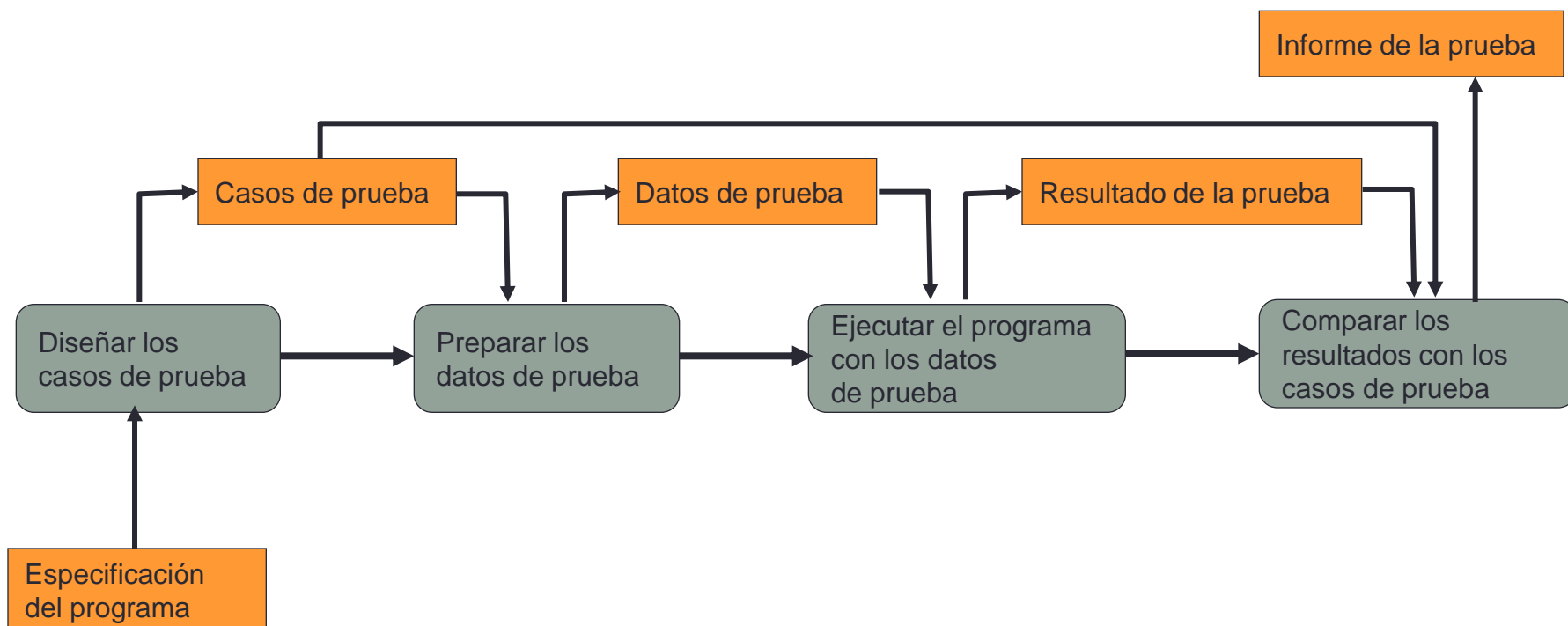
# El proceso de prueba



# El proceso de prueba

- **Pruebas de unidad:** se prueba cada módulo individualmente.
- **Prueba funcional o de integración:** el software totalmente ensamblado se prueba como un conjunto, para comprobar si cumple o no tanto los requisitos funcionales como los requisitos de rendimiento, seguridad, etc.
- **Prueba del sistema:** el software ya validado se integra con el resto del sistema (por ejemplo, elementos mecánicos, interfaces electrónicas, etc.) para probar su funcionamiento conjunto .
- **Prueba de aceptación:** el producto final se comprueba por el usuario final en su propio entorno de explotación para determinar si lo acepta como está o no (*validación*).

# Flujo de Información de la Prueba



# Proceso de depuración

- En cuanto a la **localización** del error:
  - Analizar la información y pensar.
  - Al llegar a un punto muerto, pasar a otra cosa.
  - Al llegar a un punto muerto, describir el problema a otra persona.
  - No sólo usar herramientas de depuración como único recurso.
  - No experimentar cambiando el programa.
  - Se deben atacar los errores individualmente.
  - Se debe fijar la atención también en los datos.

# Proceso de depuración

- En cuanto a la **corrección** del error:
  - Donde hay un defecto, suele haber más.
  - Debe fijarse el defecto, no sus síntomas.
  - La probabilidad de corregir perfectamente un defecto no es del 100%.
  - Cuidado con crear nuevos defectos.
  - La corrección debe situarnos temporalmente en la fase de diseño.



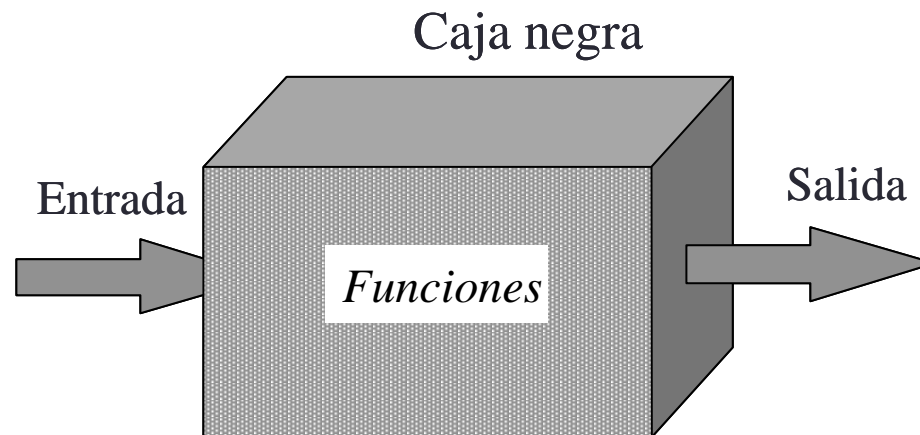
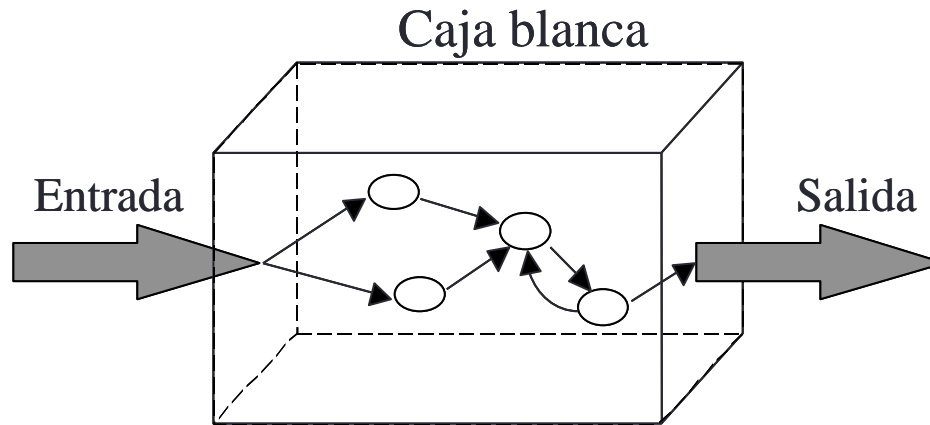
# DISEÑO DE CASOS DE PRUEBA

---

# Diseño de casos de Prueba

- El diseño de casos de prueba para la verificación del software puede significar un esfuerzo considerable (cerca del 40% del tiempo total de desarrollo)
- Existen fundamentalmente tres enfoques:
  - Prueba de caja blanca
  - Prueba de caja negra
  - Pruebas aleatorias
- Combinar ambos enfoques permite lograr mayor fiabilidad.

# Caja Blanca y Caja Negra



# Prueba de caja blanca o prueba estructural

- Se basa en el estudio minucioso de toda la operatividad de una parte del sistema, considerando los detalles procedimentales.
  - Se plantean caminos de ejecución alternativos y se ejecutan para observar los resultados y contrastarlos con lo esperado.
  - En la práctica NO es viable la verificación mediante prueba de caja blanca de la totalidad de caminos de un procedimiento (número de combinaciones posibles crece exponencialmente)
    - Se establecen distintos niveles de cobertura de los casos de prueba

## Prueba de caja negra o prueba funcional

- Analiza la compatibilidad en cuanto a las interfaces de cada uno de los componentes software entre sí.

## Prueba aleatorias

- Utiliza modelos estadísticos para crear los casos de prueba simulando las secuencias y frecuencias habituales de los datos de entrada.
  - La probabilidad de descubrir un error mediante pruebas aleatoriamente elegidas es casi la misma que si se siguen criterios de cobertura.
    - Sin embargo, pueden permanecer ocultos errores que solamente se descubren ante entradas muy concretas.
  - Este tipo de pruebas puede ser suficiente en programas poco críticos.

# PRUEBAS DE CAJA BLANCA

---

# Pruebas de caja blanca

- Usan la estructura de control del diseño procedural para derivar los casos de prueba.
- No es posible probar todos los caminos de ejecución distintos, pero sí confeccionar casos de prueba que garanticen que se verifican todos los caminos de ejecución **independientes**.
- Verificaciones para cada camino independiente:
  - Probar sus dos facetas desde el punto de vista lógico, es decir, verdadera y falsa.
  - Ejecutar todos los bucles en sus límites operacionales
  - Ejercitar las estructuras internas de datos.

# Pruebas de caja blanca

- Los errores lógicos y las suposiciones incorrectas son inversamente proporcionales a la probabilidad de que se ejecute un camino del programa.
- A menudo creemos que un camino lógico tiene pocas posibilidades de ejecutarse cuando, de hecho, se puede ejecutar de forma regular.
- Los errores tipográficos son aleatorios.
- Tal como apuntó Beizer, “los errores se esconden en los rincones y se aglomeran en los límites”.



# Tipos de cobertura

- **Cobertura de sentencias.** cada sentencia o instrucción del programa se ejecute al menos una vez.
- **Cobertura de decisiones.** cada decisión tenga, por lo menos una vez, un resultado verdadero y, al menos una vez, uno falso.
- **Cobertura de condiciones.** cada condición de cada decisión adopte el valor verdadero al menos una vez y el falso al menos una vez
- **Criterio de decisión/condición.** Consiste en exigir el criterio de cobertura de condiciones obligando a que se cumpla también el criterio de decisiones
- **Criterio de condición múltiple.** Debe garantizar todas las combinaciones posibles de las condiciones dentro de cada decisión.

# Tipos de cobertura

EJEMPLO 1:

```
si (a>b)
    entonces a=a-b
    si no b=b-a
fin_si
```

EJEMPLO 2:

```
si (a>b)
    entonces a=a-b
fin_si
```

EJEMPLO 3:

```
i=1
mientras (v[i]<>b)
y(i<>5)
hacer
    i=i+1;
fin_mientras
```

EJEMPLO 4:

```
si (a>b) y (b es
primo)
    entonces a=a-b
fin_si
```

# Tipos de cobertura

1 `if (a>b) and (b is prime)`  
 2 `then a=a-b`  
`//without else`  
`end_if`

Cobertura de sentencia: 2 fragmentos

1 `if (a>b) and (b is prime)`  
 2 `then a=a-b`  
 3 `//without else`  
`end_if`

Cobertura de Decisión: 3 fragmentos

1, 2 `if (a>b) and (b is prime)`  
 3, 4  
 5 `then a=a-b`  
 6 `//without else`  
`end_if`

Cobertura Condición: 6 fragmentos

$2^2 = \{TT, TF, FT, FF\}$  True table  
 1, 2 `if (a>b) and (b is prime)`  
 3, 4  
 5 `then a=a-b`  
 6 `//without else`  
`end_if`

Multiple condición: 6 fragmentos

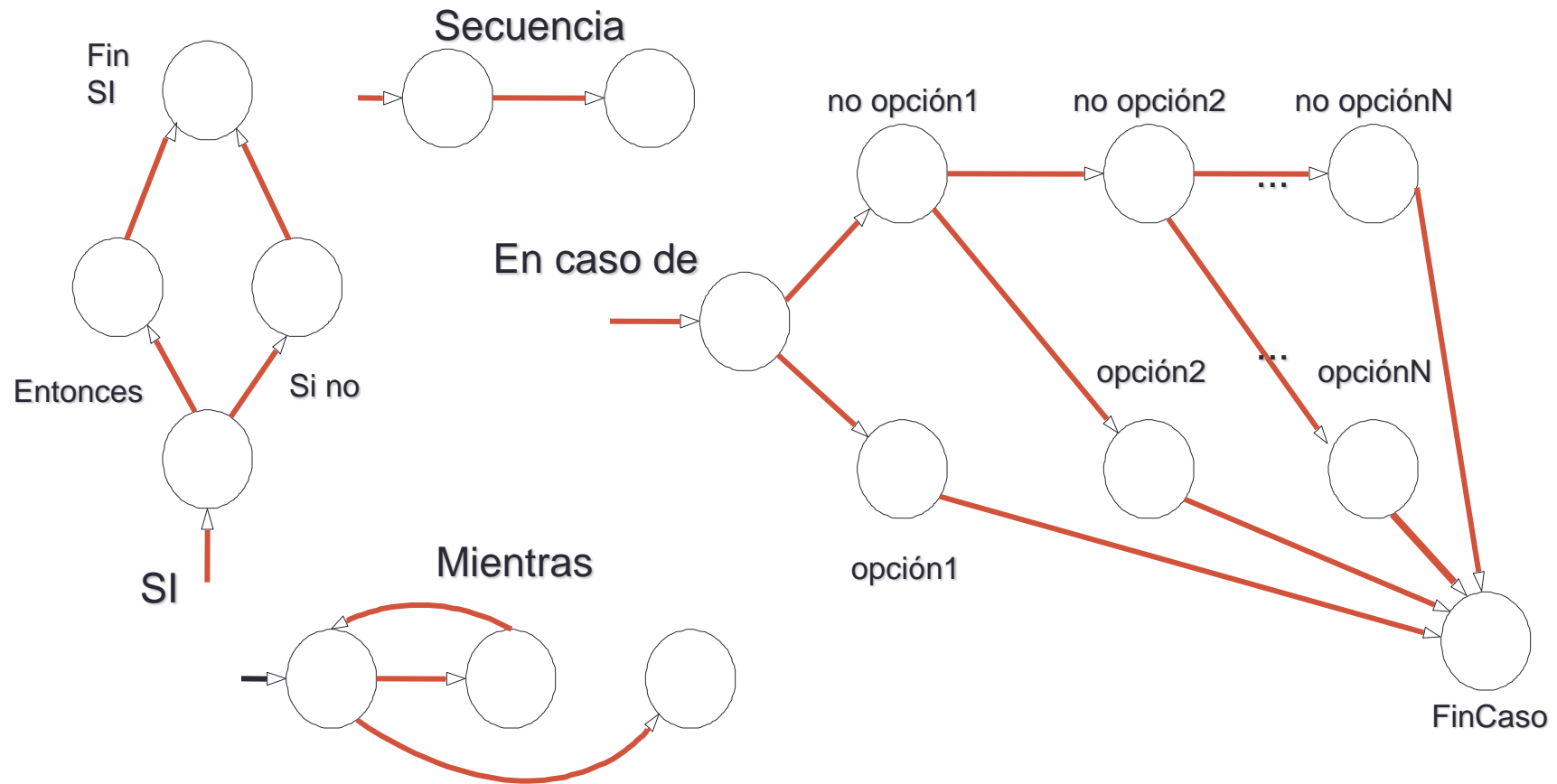
# Prueba del camino básico

- Técnica de caja blanca propuesta por Tom McCabe
- La idea es derivar casos de prueba a partir de un conjunto dado de caminos independientes por los cuales puede circular el flujo de control.
- **Camino independiente** es aquel que introduce por lo menos una sentencia de procesamiento (o condición) que no estaba considerada en el conjunto de caminos independientes calculados hasta ese momento.
- Para obtener un conjunto de caminos independientes se construirá el Grafo de Flujo asociado y se calculará su Complejidad Ciclomática.

## Prueba del camino básico: Grafos de Flujo

- El flujo de control de un programa puede representarse por un **grafo de flujo**.
- Cada **nodo del grafo** corresponde a una o más sentencias de código fuente
- Cada nodo que representa una condición se denomina **nodo predicado**.
- Cualquier representación del diseño procedural se puede traducir a un grafo de flujo.
- Un **camino independiente** en el grafo es el que incluye alguna **arista nueva**, es decir, que no estaba presente en los caminos definidos previamente.

# Grafos de Flujo



# Prueba del camino básico

- Si se diseñan casos de prueba que cubran los caminos básicos, se garantiza la ejecución de cada sentencia al menos una vez y la prueba de cada condición en sus dos posibilidades lógicas (verdadera y falsa) -> *Cobertura de Condición*
- El conjunto básico para un grafo dado puede no ser único, depende del orden en que se van definiendo los caminos.
- Cuando aparecen condiciones lógicas compuestas, la confección del grafo es más compleja.

# Ejemplo Grafo de Flujo

- Ejemplo:

SI a 0 b

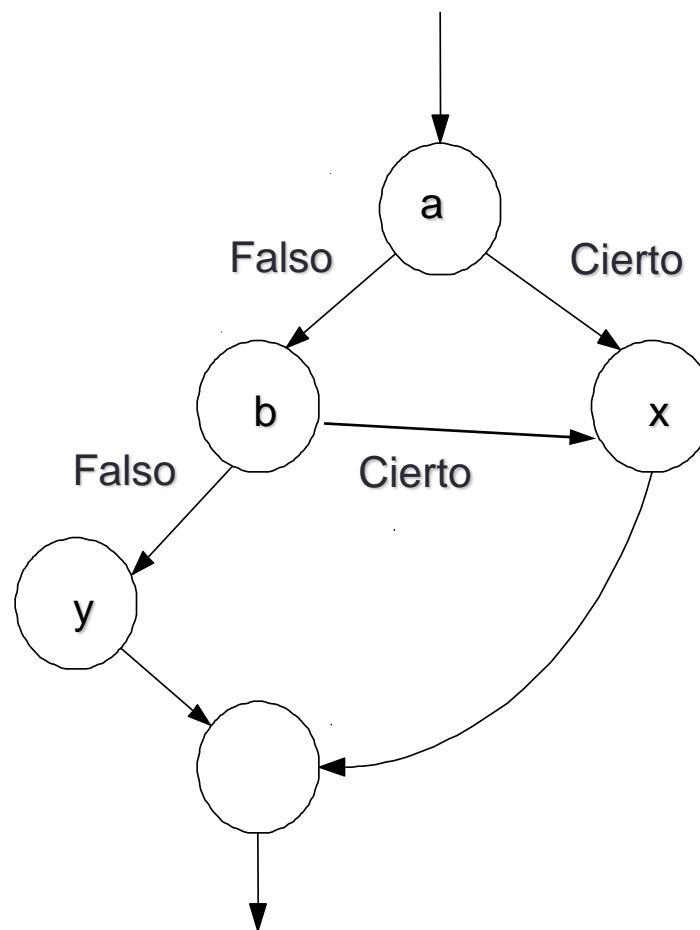
**Entonces**

hacer x

**Si No**

hacer y

FinSI



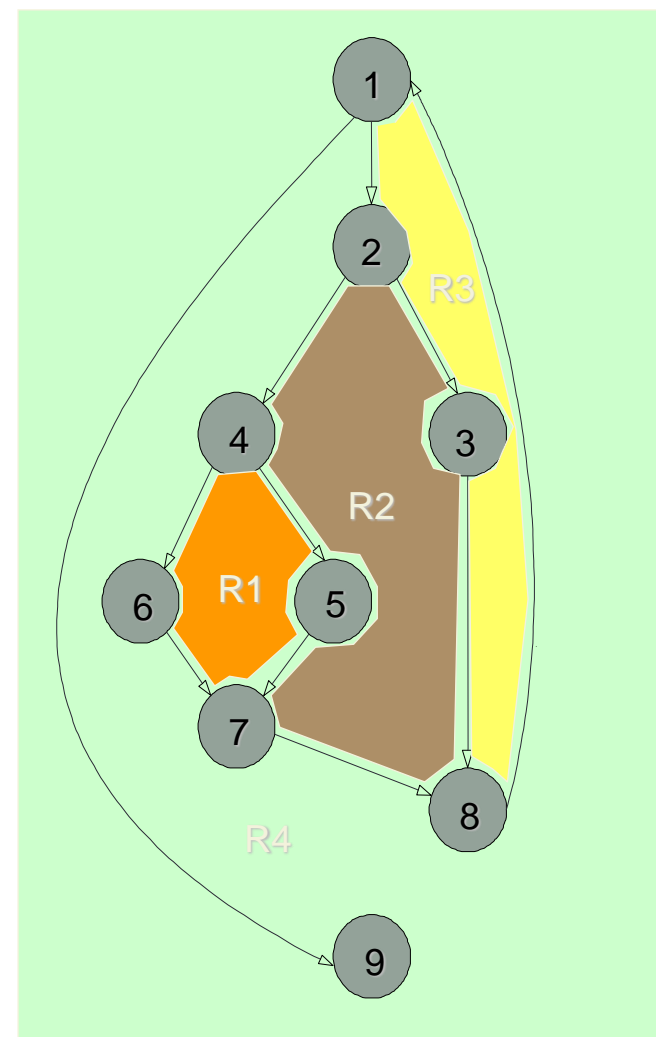


# Complejidad Ciclomática

- La **Complejidad ciclomática** de un grafo de flujo,  $V(G)$ , es el número máximo de caminos independientes en el grafo.
- Puede calcularse de tres formas alternativas:
  - El número de regiones en que el grafo de flujo divide el plano.
  - $V(G) = A - N + 2$ , donde  $A$  es el número de aristas y  $N$  es el número de nodos.
  - $V(G) = P + 1$ , donde  $P$  es el número de nodos predicado.

# Complejidad Ciclomática: Ejemplo

- $V(G) = 4$ 
  - El grafo de la figura delimita cuatro regiones.
  - $11 \text{ aristas} - 9 \text{ nodos} + 2 = 4$
  - $3 \text{ nodos predicho} + 1 = 4$



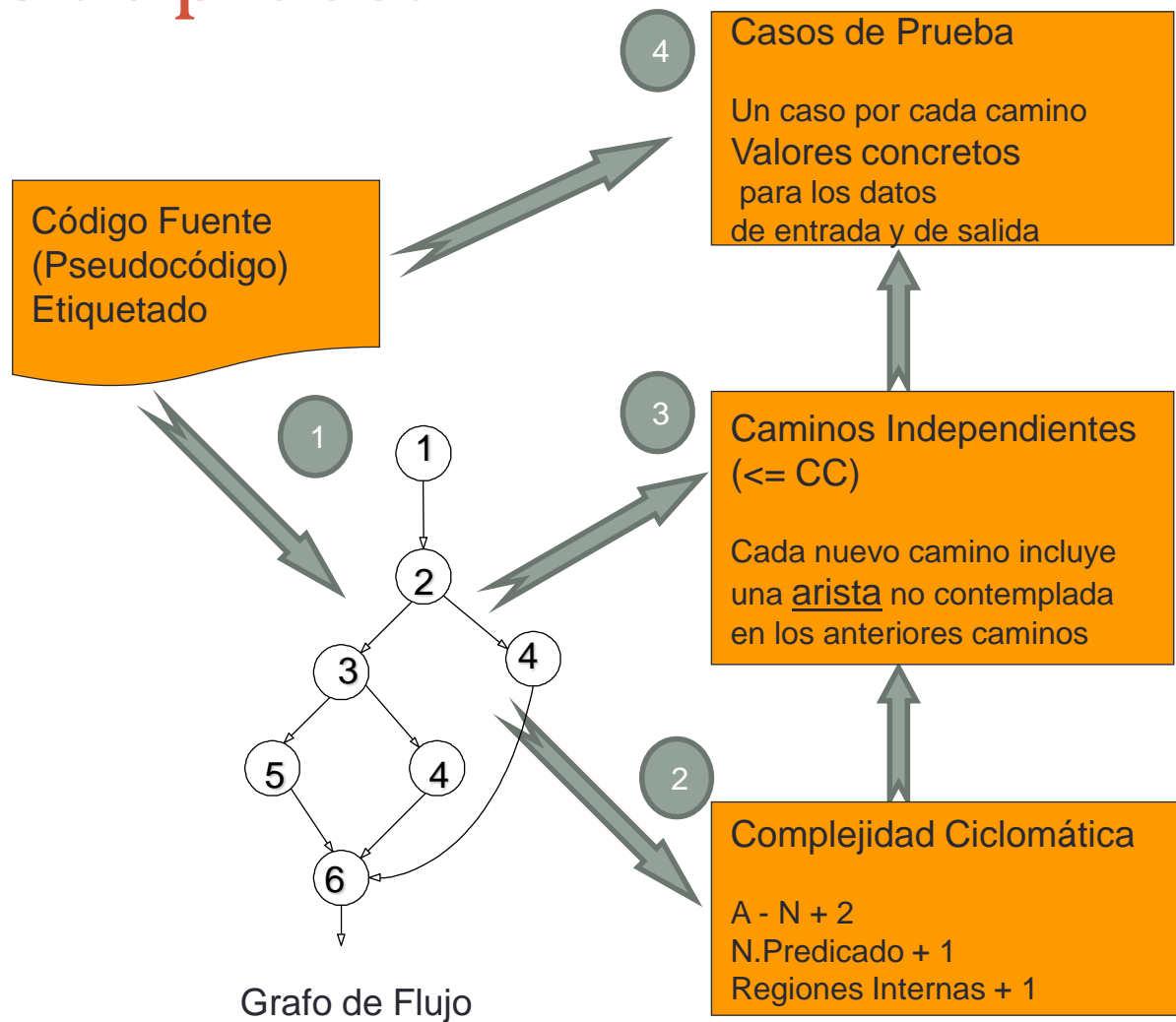
# Prueba del camino básico: Complejidad ciclomática

- El conjunto de caminos independientes del grafo será 4.
  - Camino 1: 1-9
  - Camino 2: 1-2-4-5-7-8-1-9
  - Camino 3: 1-2-4-6-7-8-1-9
  - Camino 4: 1-2-3-8-1-9
- Cualquier otro camino no será un camino independiente,  
p.e., 1-2-4-5-7-8-1-2-3-8-1-2-4-6-7-8-1-9  
ya que es simplemente una combinación de caminos ya especificados
- Los cuatro caminos anteriores constituyen un **conjunto básico** para el grafo

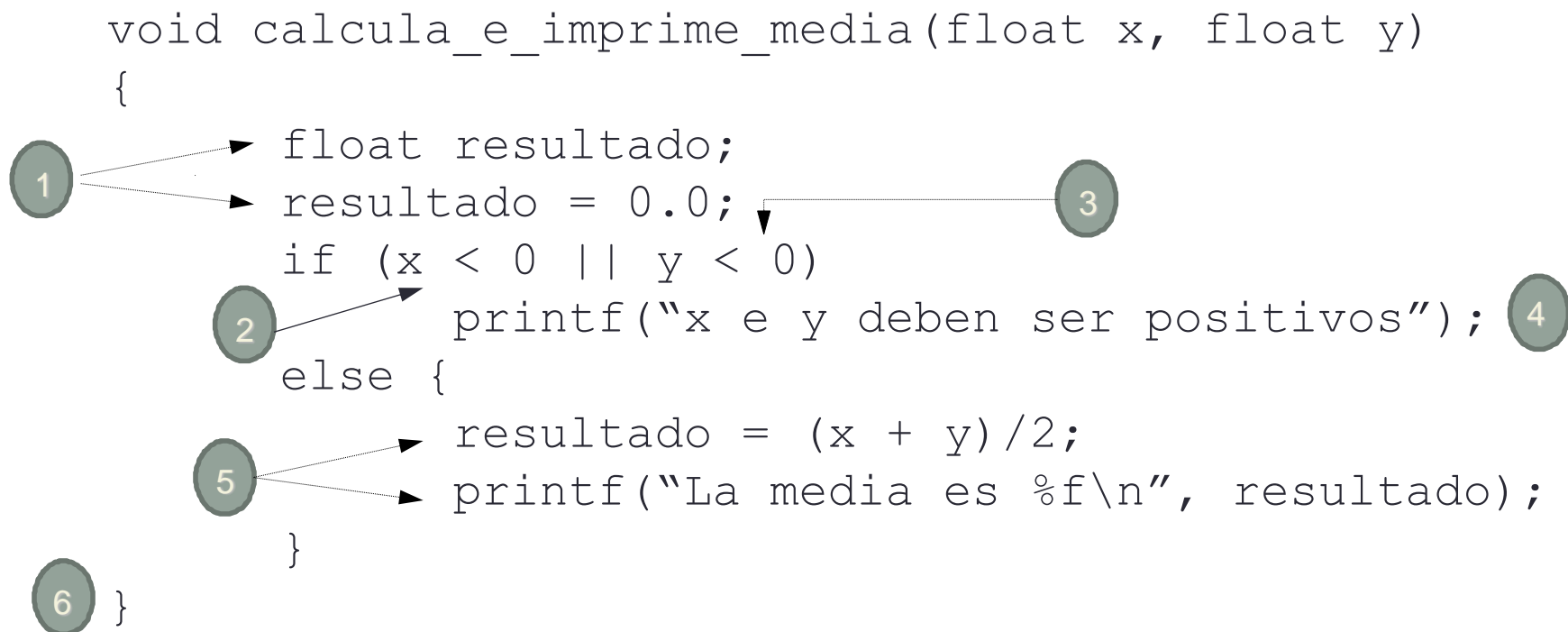
# Derivación de casos de prueba

- Pasos para diseñar los casos de prueba:
  1. Se etiqueta el código fuente (o pseudocódigo), enumerando cada instrucción y cada condición simple.
  2. A partir del código fuente etiquetado, se dibuja el grafo de flujo asociado.
  3. Se calcula la complejidad ciclomática del grafo.
  4. Se determina un conjunto básico de caminos independientes.
  5. Se preparan los casos de prueba que obliguen a la ejecución de cada camino del conjunto básico.

# Prueba del camino básico: Derivación de casos de prueba



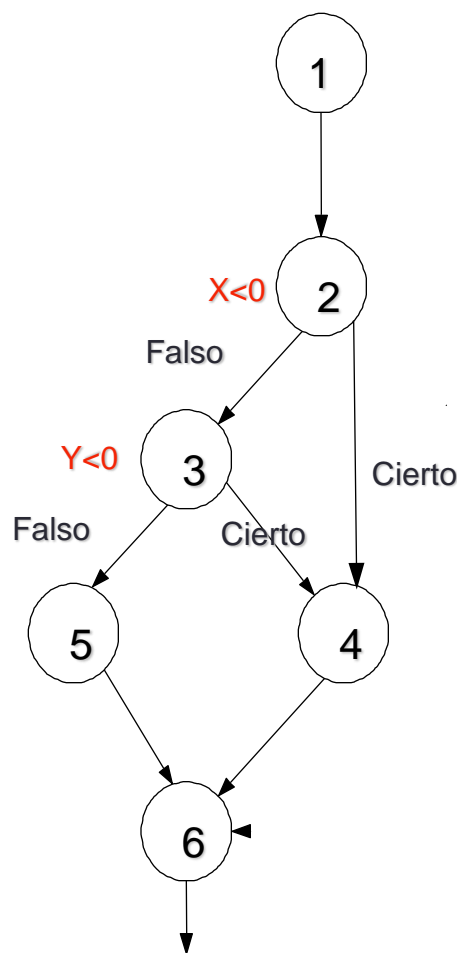
# Prueba del camino básico: Ejemplo



# Prueba del camino básico: Ejemplo

- 0: asegurarse que haya sólo un nodo inicial y sólo un nodo final
- 1: seleccionar el camino más corto entre inicio y fin y agregarlo a la base
- 2: fijar num-caminos en 1
- 3: Mientras existan nodos de decisión con salidas no utilizadas y num-caminos  $< v(G)$ ,
  - 3.1: seguir un camino básico hasta uno de tales nodos
  - 3.2: seguir la salida no utilizada y buscar regresar al camino básico tan pronto sea posible
  - 3.3: agregar el camino a la base e incrementar num-caminos en uno.

# Prueba del camino básico: Ejemplo



$V(G) = 3$  regiones. Por lo tanto, hay que determinar tres caminos independientes.

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

Casos de prueba para cada camino:

Camino 1:  $x=3$ ,  $y=5$ ,  $rdo=4$

Camino 2:  $x=-1$ ,  $y=3$ ,  $rdo=0$ , error

Camino 3:  $x=4$ ,  $y=-3$ ,  $rdo=0$ , error



## Ejercicio Camino Básico

```
int contar_letra(char cadena[10], char letra)
{
    int contador, n, lon;
    n=0; contador=0;
    lon = strlen (cadena);
    if (lon > 0) {
        do {
            if (cadena[contador] == letra) n++;
            contador++;
            lon--;
        } while (lon > 0);
    }
    return n;
}
```

# Código Fuente Etiquetado

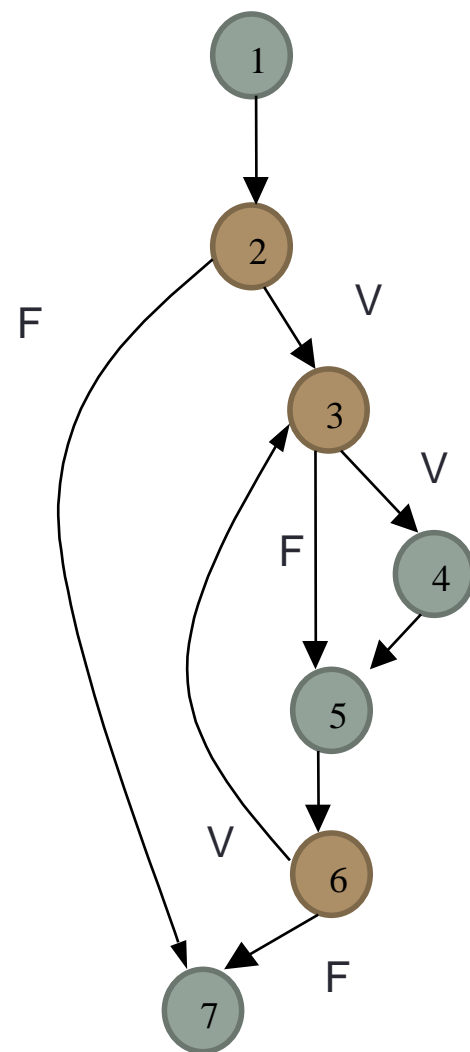
```
int contar_letra(char cadena[10], char letra)
{
    int contador, n, lon;
    n=0; contador=0;
    lon = strlen (cadena);
    if (lon > 0) {
        do {
            if (cadena[contador] == letra)
                n++;
            contador++;
            lon--;
        } while (lon > 0);
    }
    return n;
}
```

Diagrama de etiquetado del código fuente:

- 1: Grupo de declaración de variables y inicialización de `n` y `contador`.
- 2: Condición de entrada `if (lon > 0)`.
- 3: Condición de comparación `if (cadena[contador] == letra)`.
- 4: Incremento de `n` (`n++`).
- 5: Incremento de `contador` y decremento de `lon`.
- 6: Condición de salida del bucle `while (lon > 0)`.
- 7: Retorno de `n`.

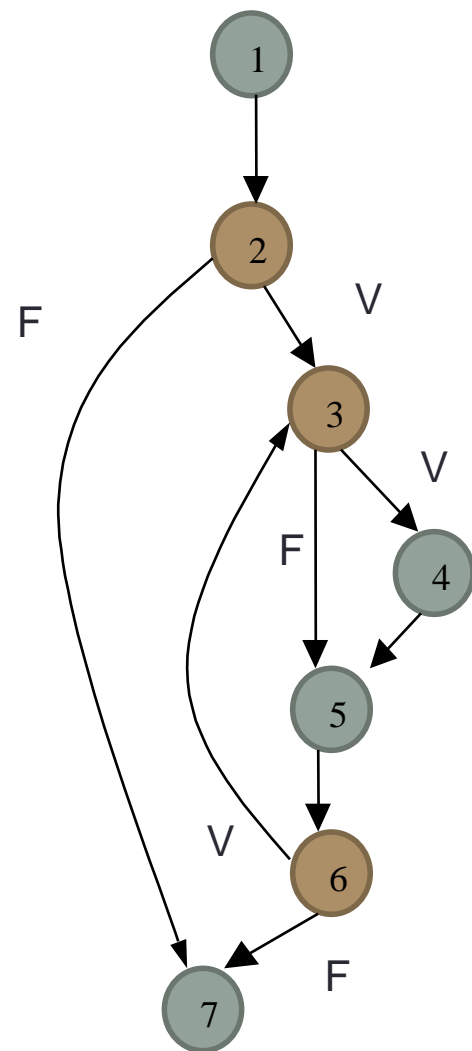
# Grafo de Flujo

```
int contar_letra(char cadena[10], char letra)
{
    int contador, n, lon;
    n=0; contador=0;
    lon = strlen (cadena);
    if (lon > 0) {
        do {
            if (cadena[contador] == letra)
                n++;
            contador++;
            lon--;
        } while (lon > 0);
    }
    return n;
}
```



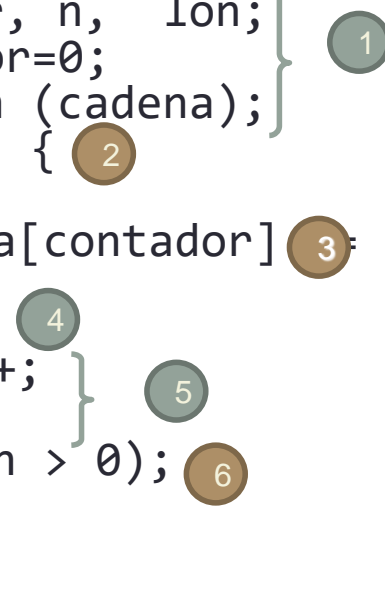
# Complejidad Ciclomática y Caminos Independientes

- $V(G) = 4$ ;
  - Nodos=7; Aristas=9;
  - Nodos Predicado=3;
  - Regiones = 4
- Conjunto de caminos independientes:
  - 1-2-7
  - 1-2-3-4-5-6-7
  - 1-2-3-5-6-7
  - 1-2-3-4-5-6-3-5-6-7 (No es el único)



# Casos de Prueba

```
int contar_letra(char
cadena[10], char letra)
{
    int contador, n, lon;
    n=0; contador=0;
    lon = strlen (cadena);
    if (lon > 0) {
        do {
            if (cadena[contador]
letra)
                n++;
            contador++;
            lon--;
        } while (lon > 0);
    }
    return n;
}
```



## Caso 1 (1-2-7)

cadena = "", letra = 'a'  
n = 0;

## Caso 2 (1-2-3-4-5-6-7)

cadena = "a", letra = 'a'  
n = 1;

## Caso 3 (1-2-3-5-6-7)

cadena = "b", letra = 'a'  
n = 0;

## Caso 4 (1-2-3-4-5-6-3-5-6-7)

cadena = "ab", letra = 'a'  
n = 1;

# PRUEBAS DE CAJA NEGRA

---

# Pruebas de caja negra

- Los métodos de caja negra se centran sobre los requisitos funcionales del software.
- La prueba de la caja negra intenta encontrar errores de los siguientes tipos:
  - Funciones incorrectas o inexistentes.
  - Errores relativos a las interfaces.
  - Errores en estructuras de datos o en accesos a bases de datos externas.
  - Errores debidos al rendimiento.
  - Errores de inicialización o terminación.

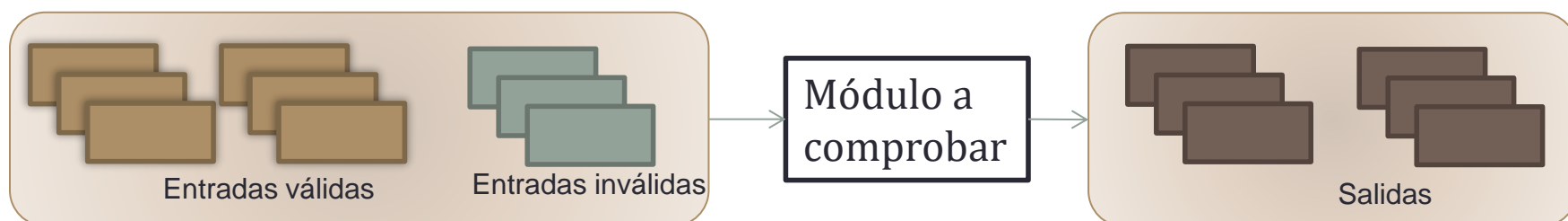
# Partición Equivalente

- Consiste en derivar casos de prueba mediante la **partición del dominio de entrada** en clases de equivalencia, evaluando su comportamiento para un **valor representativo** de cada clase.
- Una **clase de equivalencia** representa el conjunto de estados válidos o inválidos para todas las condiciones de entrada que satisfagan dicha clase
- La prueba sobre el valor representativo de una clase permite suponer «razonablemente» que el resultado de éxito/fallo será el mismo que para cualquier otro valor de la clase



# Partición Equivalente

- Condiciones de entrada (restricciones de formato o contenido)
  - De datos válidos
  - De datos no válidos o erróneos
- Objetivo: descubrir clases de errores para cada clase de equivalencia identificada
- Al reducir el número de casos de prueba a un conjunto más pequeño nos arriesgamos a perder información relevante



# Partición Equivalente

- **Técnica para identificación** de casos de prueba:
  1. Para cada condición de entrada del software bajo pruebas, **identificar sus clases de equivalencia** utilizando ciertas **heurísticas**.
  2. Asignar un **número único** a cada clase de equivalencia identificada.
  3. Hasta que todas las clases de equivalencia válidas hayan sido cubiertas (incorporadas a casos de prueba), **escribir un caso de prueba que cubra tantas clases válidas no incorporadas como sea posible**.
  4. Hasta que todas las clases de equivalencia no válidas hayan sido cubiertas por un caso de prueba, **escribir un caso de prueba para cada clase no válida sin cubrir**.

El proceso anterior también puede aplicarse para las condiciones de salida del software bajo pruebas

# Partición Equivalente

- **Heurísticas de identificación** de clases de equivalencia:
  - a) Si se especifica un **rango de valores** para los datos de entrada, se creará una clase válida y dos clases no válidas
  - b) Si se especifica un **número finito de valores**, se creará una clase válida y dos no válidas
  - c) Si se especifica una situación del tipo «**debe ser**» o **booleana** por ejemplo, «el primer carácter debe ser una letra», se identifican una clase válida «es una letra» y una no válida «no es una letra»
  - d) Si se especifica **un conjunto de valores** admitidos y se sabe que el programa trata **de forma diferente cada uno de ellos**, se identifica una clase válida por cada valor y una no válida
  - e) Si se sospecha que ciertos elementos de una clase no se tratan igual que el resto de la misma, deben dividirse en **Clases Menores**

# Partición Equivalente

Heurísticas	Número de clases válidas	Número de clases inválidas
<b>Rango de valores</b> <b>Ej: [20..30]</b>	<b>1:</b> valor dentro del rango <b>(25)</b>	<b>2:</b> uno bajo el límite inferior y otro sobre el límite superior <b>(15,41)</b>
<b>Valor finito</b> <b>Ej: {2,4,6,8,10}</b> <b>Ej2: código de 8 char</b>	<b>1:</b> valor dentro del conjunto <b>(4) (8 char)</b>	<b>2:</b> uno inferior del menor valor y otro superior al valor más alto <b>(1,12) (7,9 char)</b>
<b>«debe ser » condición booleana</b> <b>Ej: Debe ser una letra</b>	<b>1:</b> valor que cumple la condición <b>(‘m’)</b>	<b>1:</b> valor que hace la condición falsa
<b>Conjunto de valores aceptados. Trata de forma diferente a cada uno.</b> <b>Ej: { blue, white, black}</b>	<b>Número de valores aceptados (3: { blue, white, black})</b>	<b>1:</b> Un valor no aceptado <b>(red)</b>
<b>Clases menores</b> <b>0&lt;value&lt;5; 10&lt;=value&lt;20</b>	Cuando podemos transformar las clases válidas en subclases de los previos ejemplos (ej: <b>una para [0,5[ y otra para [10,20[</b> ), y cada subclase válida provoca un comportamiento o salida diferente	

# Partición Equivalente

Un programa toma como entrada un fichero cuyo formato de registro consta de los siguientes campos:

- Numero-empleado es un campo de números enteros positivos de 3 dígitos excluido el 000.
- Nombre-empleado es un campo alfanumérico de 10 caracteres.
- Meses-Trabajo es un campo que indica el número de meses que lleva trabajando el empleado; es un entero positivo incluye el 000 de 3 dígitos.
- Directivo es un campo de un solo carácter que puede ser «+» para indicar que el empleado es un directivo y «-» para indicar que no lo es.

El programa asigna una prima que se imprime en un listado a cada empleado según las normas siguientes:

- P1 a los directivos con, al menos, 12 meses de antigüedad
- P2 a los no directivos con, al menos, 12 meses de antigüedad
- P3 a los directivos sin un mínimo de 12 meses de antigüedad
- P4 a los no directivos sin un mínimo de 12 meses de antigüedad

Se pide:

- a) Crear una tabla de clases de equivalencia las clases deberán ser numeradas en la que se indiquen las siguientes columnas en cada fila: 1 Condición de entrada que se analiza; 2 Clases válidas y 3 Clases no válidas que se generan para la condición 4 Regla heurística que se aplica para la generación de las clases de la fila
- b) Generar los casos de prueba

# Partición Equivalente

Condición de entrada	Clases válidas	Clases no válidas	Regla heurística
Numero-empleado			
Nombre-empleado			
Meses-Trabajo			
Directivo			

CP C.Validos	Clases válidas	Entrada	Salida
	1 – 5 – 8 - 13	123, gumersindo, 009, +	P3
	1 – 5 – 9 – 14	456, sebastiano, 013, -	P2

[illegible]

# Partición Equivalente

Condición de entrada	Clases válidas	Clases no válidas	Regla heurística
Numero-empleado	[001-999] <b>1</b>	<=000 <b>2</b> >999 <b>3</b> No es un número <b>4</b>	Rango Booleana
Nombre-empleado	10 caracteres <b>5</b>	<10 <b>6</b> >10 <b>7</b>	Nº finito de valores
Meses-Trabajo	[000-011] <b>8</b> [012-999] <b>9</b>	<000 <b>10</b> >999 <b>11</b> No número <b>12</b>	Rango Clases menores Booleana
Directivo	+ <b>13</b> - <b>14</b>	Otro caracter <b>15</b>	Conjunto de valores

CP C. Validos	Clases válidas	Entrada	Salida
	1 – 5 – 8 - 13	123, gumersindo, 009, +	P3
	1 – 5 – 9 – 14	456, sebastiano, 013, -	P2

	c. no válidas	Entrada	Salida
Casos de Prueba No Validos	2 - 5 - 9 - 13	000, gumersindo, 014, +	
	3 - 5 - 9 - 14	1024, minotauros, 016, -	
	4 - 5 - 8 - 13	abc, sebastiano, 008, +	
	1 - 6 - 8 - 13	123, cobos, 006, +	
	1 - 7 - 8 - 13	123, torreceballos, 003, +	
	1 - 5 - 10 - 13	123, margaritos, -01, +	
	1 - 5 - 11 - 14	123, margaritos, 1024, -	
	1 - 5 - 12 - 14	123, margaritos, abc, -	
	1 - 5 - 9 - 15	123, margaritos, 013, *	

# Ejercicio Partición Equivalente

- Aplicación bancaria. Datos de entrada:
  - **Código de área:** número de 3 dígitos que no empieza por 0 ni por 1
  - **Nombre de identificación de operación:** 6 caracteres
  - **Órdenes posibles:** “cheque”, “depósito”, “pago factura”, “retirada de fondos”. El programa ofrece un comportamiento distinto para cada una de estas entradas.



# Ejercicio Partición Equivalente

## Código de área:

Lógica:

1 clase válida: número

Rango:

1 clase válida:  $200 < \text{código} < 999$

2 clases no válidas:  $\text{código} < 200$ ;  $\text{código} > 999$

1 clase no válida: no es número

## Nombre de identificación:

Valor específico:

1 clase válida: 6 caracteres

2 clases no válidas: más de 6 caracteres; menos de 6 caracteres

## Órdenes posibles:

Conjunto de valores:

1 clase válida: 4 órdenes válidas

1 clase no válida: orden no válida

# Ejercicio Partición Equivalente

<b>Datos de Entrada</b>	<b>Clases Válidas</b>	<b>Clases No Válidas</b>
Código de área	(1) $200 \leq \text{código} \leq 999$	(2) código $< 200$ (3) código $> 999$ (4) no es numérico
Identificación	(5) 6 caracteres	(6) menos de 6 caracteres (7) más de 6 caracteres
Orden	(8) "cheque" (9) "depósito" (10) "pago factura" (11) "retirada de fondos"	(12) ninguna orden válida

# Ejercicio Partición Equivalente

Código	Identificación	Orden	Clases Cubiertas
300	Nómina	"Depósito"	(1) <sup>C</sup> (5) <sup>C</sup> (9) <sup>C</sup>
400	Viajes	"Cheque"	(1) (5) (8) <sup>C</sup>
500	Coches	"Pago-factura"	(1) (5) (10) <sup>C</sup>
600	Comida	"Retirada-fondos"	(1) (5) (11) <sup>C</sup>

# Ejercicio Partición Equivalente: Casos de Prueba NO Válidos

Código	Identificación	Orden	Clases Cubiertas
180	Viajes	“Pago-factura”	(2) <sup>C</sup> (5) (10)
1032	Nómina	“Depósito”	(3) <sup>C</sup> (5) (9)
XY	Compra	“Retirada-fondos”	(4) <sup>C</sup> (5) (11)
350	A	“Depósito”	(1) (6) <sup>C</sup> (9)
450	Regalos	“Cheque”	(1) (7) <sup>C</sup> (8)
550	Casita	&%4	(1) (5) (12) <sup>C</sup>

# Ejemplo Crear Reserva de Vehículo

- El usuario introducirá los siguientes datos para realizar la reserva de un vehículo:
  - El DNI (DNI: 9 caracteres, 8 dígitos y 1 letra),
  - La Fecha de Recogida del vehículo (Día, Mes y Año),
  - La Categoría del vehículo: valor entero entre 1 y 10
  - La Modalidad de Alquiler: “por km” o “ilimitado”. El programa ofrece un comportamiento distinto para cada una de estas entradas.

# Ejemplo Crear Reserva

Condición de entrada	Clases válidas	Clases no válidas	Regla heurística
DNI	9 caracteres DDDDDDDDL 1	<9 2	Nº finito de valores
		>9 3	
		No son dígitos 4	Booleana
		No es letra 5	Booleana
Fecha Recogida	dd-mm-aaaa existe, correspondiente a hoy o el futuro 6	Fecha no existe 7	Booleana
		Fecha en el pasado 8	Booleana
Categoría	1 < num cat <= 10 9	Num cat < 1 10	Rango
		Num cat > 10 11	
Modalidad de Alquiler	Km 12	No seleccionado 13	Conjunto de valores
	Ilimitado 14		

# Ejemplo Crear Reserva

CP C. Válidos	Clases válidas	Entrada	Salida
	1 – 6 – 9- 12	"12345678A" "15-01-2019" 3 "km"	Reserva OK
	1 – 6 – 9- 14	"12345678A" "15-01-2019" 3 "Ilimitado"	Reserva OK

Casos de Prueba No Válidos	c. no válidas	Entrada	Salida
	2 – 6 – 9 – 12	"1234567A" "15-12-2019" 3 "km"	Error DNI < 9 car.
	3 - 6 – 9 – 12	"123456789A" "15-12-2019" 3 "km"	Error DNI > 9 car.
	4 - 6 – 9 – 12	"Z2345678A" "15-12-2019" 3 "km"	Error DNI No dígitos
	5 - 6 – 9 – 12	"123456789" "15-12-2019" 3 "km"	Error DNI No letra
	1 - 7 – 9 – 12	"12345678A" "30-02-2019" 3 "km"	Error Fecha no existe
	1 - 8 – 9 – 12	"12345678A" "15-12-2010" 3 "km"	Error Año pasado
	1 – 6 – 10 – 12	"12345678A" "15-12-2019" 0 "km"	Error Categoría <1
	1 – 6 – 11 – 12	"12345678A" "15-12-2019" 11 "km"	Error Categoría > 10
	1 – 6 – 9 – 13	"12345678A" "15-12-2019" 3 ""	Error Modalidad vacía

# Análisis de valores límite

- La técnica de **Análisis de Valores Límites** selecciona casos de prueba que ejerciten los valores límite dada la tendencia de la aglomeración de errores en los extremos.
- Complementa la de la partición equivalente. En lugar de realizar la prueba con cualquier elemento de la partición equivalente, se escogen los valores en los bordes de la clase.
- 
- Se derivan tanto casos de prueba a partir de las condiciones de entrada como con las de salida.



# Análisis de valores límite

- Directrices:

- Si una condición de entrada especifica un rango delimitado por los valores a y b, se deben diseñar casos de prueba para los valores a y b y para los valores justo por debajo y justo por encima de ambos.
- Si la condición de entrada especifica un número de valores, se deben desarrollar casos de prueba que ejerciten los valores máximo y mínimo además de los valores justo encima y debajo de aquéllos.

# Análisis de valores límite

- Directrices:

- Aplicar las directrices anteriores a las condiciones de salida. Componer casos de prueba que produzcan salidas en sus valores máximo y mínimo.
- Si las estructuras de datos definidas internamente tienen límites prefijados (p.e., un array de 10 entradas), se debe asegurar diseñar un caso de prueba que ejercite la estructura de datos en sus límites.

Heurísticas	Número de clases válidas	Número de clases inválidas
<b>Rango de valores</b> <b>Ex: [20..30]</b>	<b>4:</b> valores en los límites ( <b>20,21,29,30</b> )	<b>2:</b> uno <b>justo</b> por debajo del límite inferior y otro <b>justo</b> por encima del límite superior ( <b>19,31</b> )
<b>Conjunto finito de valores</b> <b>Ex: {2,4,6,8,10}</b>	<b>4:</b> valores mínimos y máximos del conjunto ( <b>2,4, 8,10</b> )	<b>2:</b> valores fuera del conjunto, uno <b>justo</b> por debajo del valor menor y otro <b>justo</b> por encima del valor superior ( <b>1,11</b> )

# Otros tipos de prueba

- Recorridos (“walkthroughs”).
- Pruebas de robustez (“robustness testing”)
- Pruebas de aguante (“stress”)
- Prestaciones (“performance testing”)
- Conformidad u Homologación (“conformance testing”)
- Interoperabilidad (“interoperability testing”)
- Regresión (“regression testing”)
- Prueba de comparación

# Herramientas automáticas de prueba

- **Analizadores estáticos.** Soportan pruebas de las sentencias que consideran más débiles dentro de un programa.
- **Auditores de código:** filtros para verificar que el código fuente cumple determinados criterios de calidad (dependerá fuertemente del lenguaje en cuestión).
- **Generadores de archivos de prueba:** confeccionan automáticamente ficheros con datos que servirán de entrada a las aplicaciones.
- **Generadores de datos de prueba.** Confeccionan datos de entrada determinados que conlleven un comportamiento concreto del software.
- **Controladores de prueba:** Confeccionan y alimentan los datos de entrada y simulan el comportamiento de otros módulos para restringir el alcance de la prueba.

# Herramientas automáticas de prueba

- Analizadores estáticos:
  - Defectos en los datos: variables usadas antes de inicializarse, variables declaradas y nunca usadas, variables que no se usan entre dos asignaciones de algún valor, posibles violaciones de los límites de vector, etc.
  - Defectos en el control: fragmentos de código que nunca se alcanza.
  - Defectos en la entrada/salida: se devuelve una variable de salida sin que se modifique su valor.
  - Defectos en la interfaz: tipos o número de parámetros incorrectos, no utilización del resultado de una función, funciones no llamadas.
  - Defectos en el manejo de punteros.