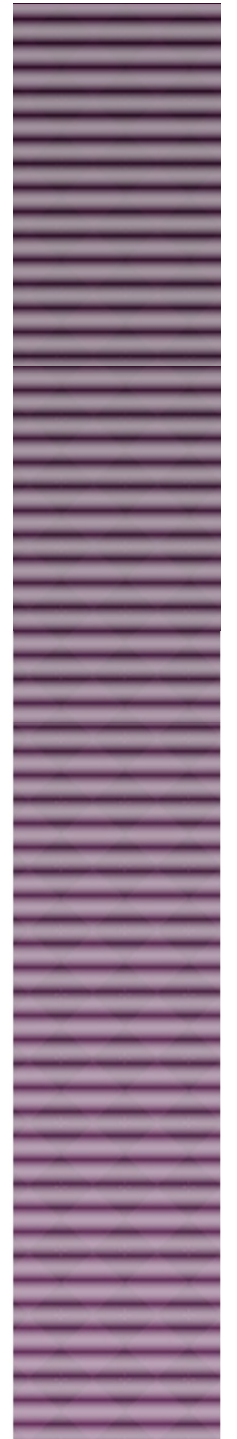


# UNIDAD 3

## Diseño y Realización de Pruebas

*"La prueba de programas puede utilizarse para mostrar la presencia de errores, pero nunca para demostrar su ausencia."*

**Edgar Dijkstra**



# 1. Introducción

Una **prueba de software** es todo proceso orientado a comprobar la calidad del software mediante la identificación de fallos en el mismo.

La prueba implica necesariamente la ejecución del software.

Uno de los objetivos fundamentales de las pruebas es que el porcentaje de fallos detectados por el usuario sea lo menor posible.

Probar un producto no es sino comparar su estado actual con el estado esperado de acuerdo a una especificación del mismo.

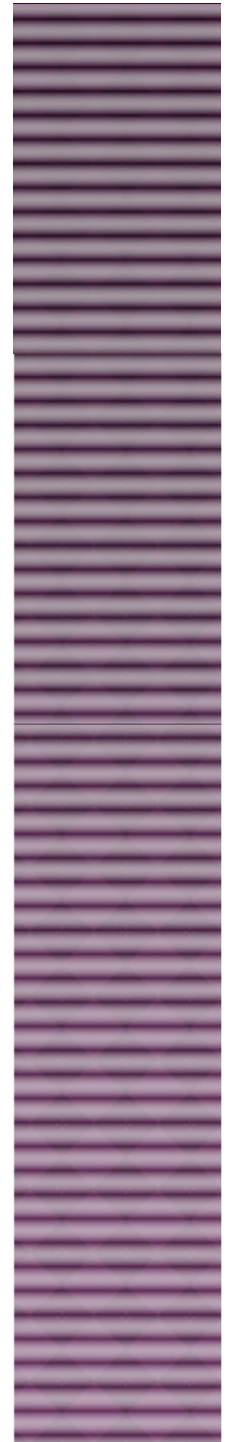


# 1. Introducción

Las pruebas son la última de las actividades del desarrollo cuyo único objetivo es detectar fallos en el software -> FALSO.

Se recomienda comenzar a diseñar las pruebas al mismo tiempo que se realiza la especificación de requisitos.

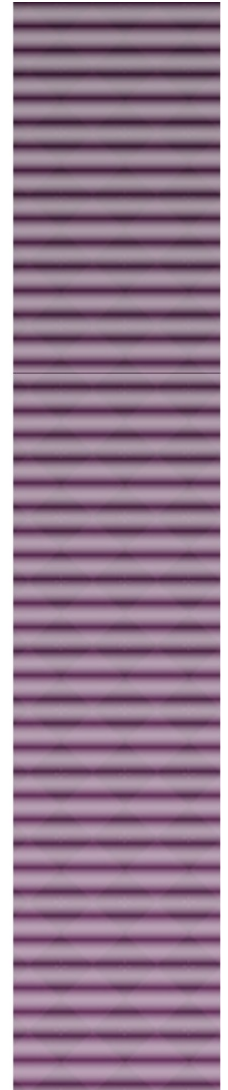
El proceso de prueba de un software es considerado en la actualidad una tarea compleja y constante a lo largo del proceso de desarrollo, cuyo objetivo último es la producción de software de calidad.



# 1. Introducción

La ejecución de pruebas de un sistema involucra una serie de etapas:

- Planificación de pruebas
- Diseño y construcción de los casos de prueba
- Definición de los procedimientos de prueba
- Ejecución de las pruebas
- Registro de resultados obtenidos
- Registro de errores encontrados
- Depuración de los errores
- Informe de resultados obtenidos



## 2. Conceptos Fundamentales

Un **error** o **defecto** es una imperfección en el software que provoca un funcionamiento incorrecto del mismo.

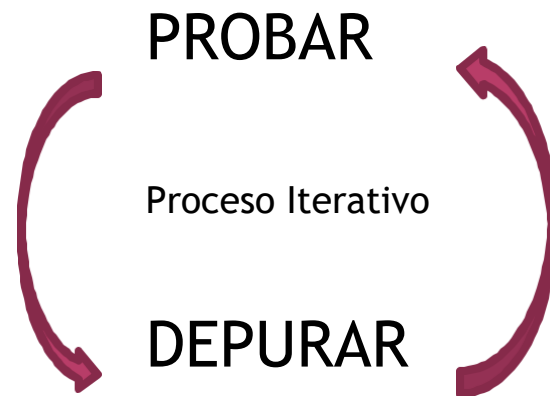
La **validación** es el proceso de evaluación de un sistema o de uno de sus componentes para determinar si el producto que se está construyendo corresponde a las necesidades que en su momento se detectaron, es decir, si es el producto adecuado

Las pruebas permiten descubrir fallos que han sido causados por defectos en el software, los cuales deben de subsanarse.

## 2. Conceptos Fundamentales

**Probar** un software es el proceso de mostrar la presencia de un error en el mismo.

**Depurar** un software consiste en descubrir en qué lugar exacto se encuentra un error y modificar el software para eliminar dicho error.

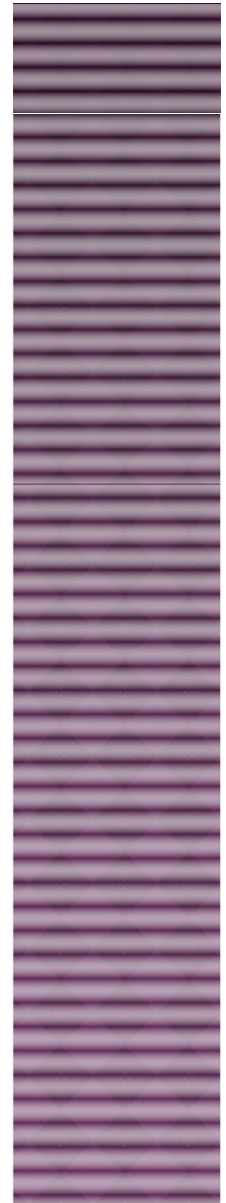




## 2. Conceptos Fundamentales

Un **oráculo** es cualquier agente (humano o no) capaz de decidir si un programa se comporta correctamente durante una prueba y, por tanto, capaz de dictaminar si la prueba se ha superado o no.

Una **prueba** es una 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.





# 3. Riesgos

Los principales riesgos a los que se enfrenta una organización al distribuir software insuficientemente probado son:

Deterioro de su imagen

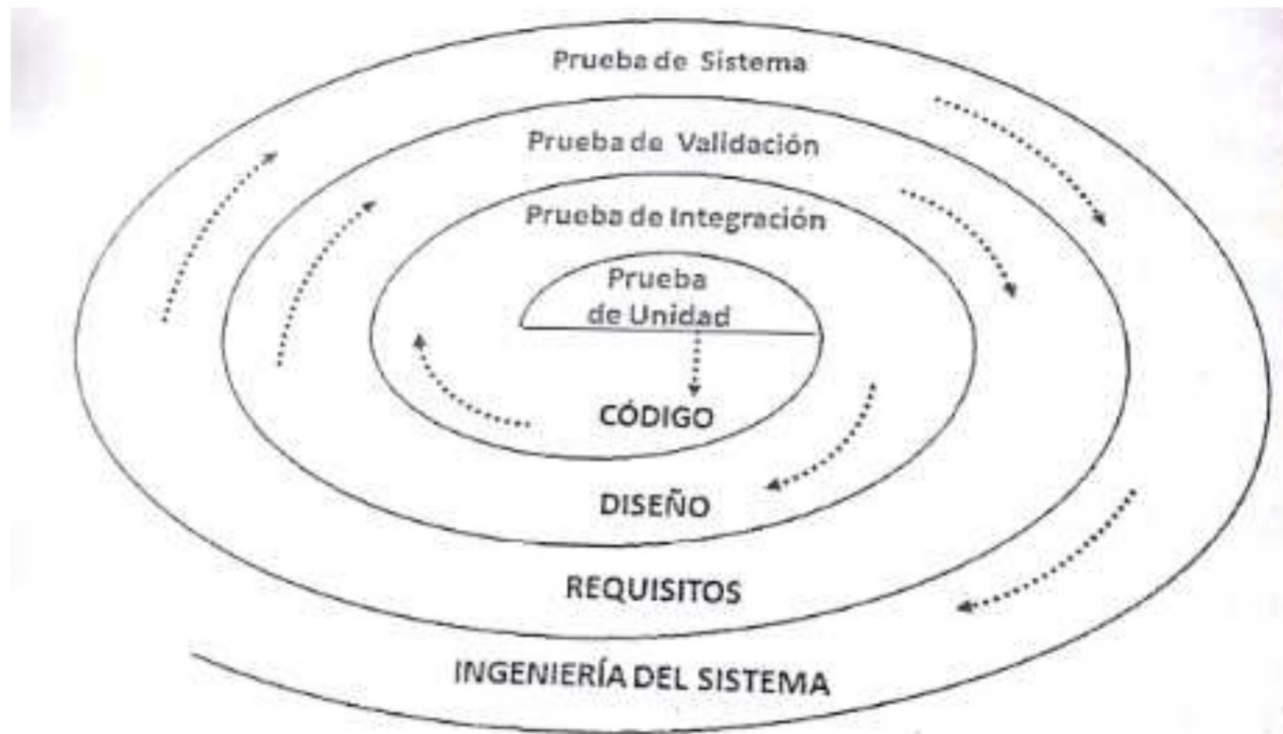
Pérdidas económicas

**Consecuencias legales:** en junio de 1990, millones de teléfonos dejaron de funcionar durante 9 horas en EEUU por culpa de una sentencia **break** situada en el lugar incorrecto del programa de encaminamiento de llamadas de la compañía AT&T. 70000 personas reclamaron haber sido afectadas, mientras se estima que 70 millones de llamadas no se completaron.




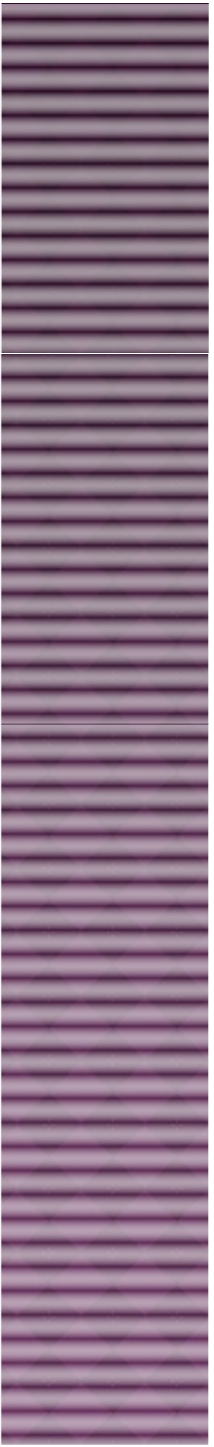
## 4. Estrategias prueba Software

La estrategia de prueba del software se puede ver en el contexto de una espiral.



## 4. Estrategias prueba Software

- En el vértice de la espiral comienza la **prueba de unidad**, la cual se centra en probar cada uno de los módulos de los que se compone el software.
- La prueba avanza para llegar a la **prueba de integración** donde se toman los módulos probados mediante la prueba de unidad y se construye una estructura de programa que esté de acuerdo con lo que dicta el diseño. El foco de atención es el diseño.
- La espiral avanza llegando a la **prueba de validación o de aceptación** donde se prueba el software en el entorno real de trabajo con intervención del usuario final. Se validan los requisitos establecidos como parte del análisis de requisitos del software comparándolos con el sistema que ha sido construido.

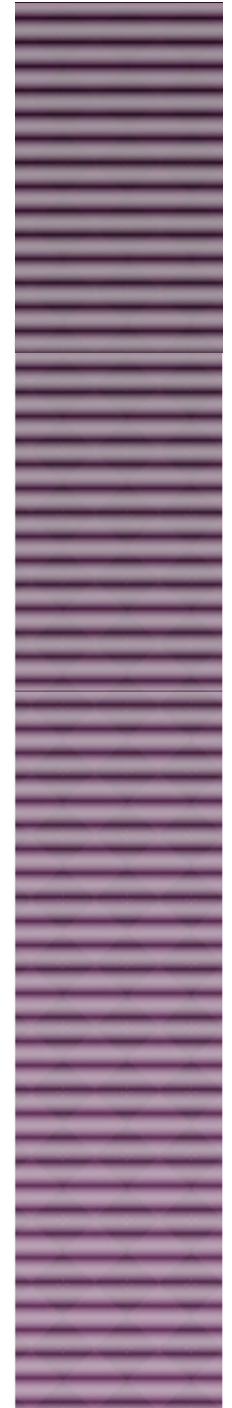
- 
- 
- Finalmente se llega a la **prueba del sistema** donde se verifica que cada elemento encaja de forma adecuada y se alcanza la funcionalidad y rendimiento total. Se prueba como un todo el software y otros elementos del sistema.

## 4.1 Pruebas de unidad

Una **prueba unitaria** es la prueba de un módulo concreto dentro de un software que incluye muchos otros módulos, con el objeto de verificar que el funcionamiento aislado de dicho módulo cumple la función para la que ha sido construido.

El objetivo final del conjunto de las pruebas unitarias que se llevan a cabo es comprobar que las partes individuales que conforman el software son correctas.

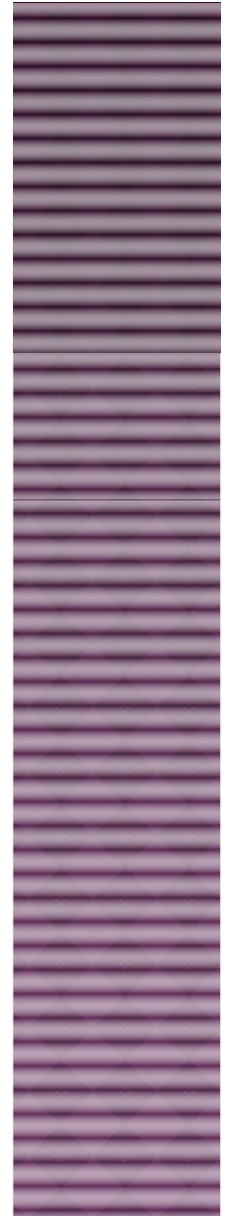
Una prueba de unidad es un componente software específicamente creado para verificar el funcionamiento de un componente del sistema en construcción.



## **4.2 Pruebas de integración**

Una prueba de integración es el proceso que permite verificar si un módulo funciona adecuadamente cuando trabaja conjuntamente con otros, por lo que es necesario comprobar si existen defectos en las interfaces entre dicho módulo y el resto.

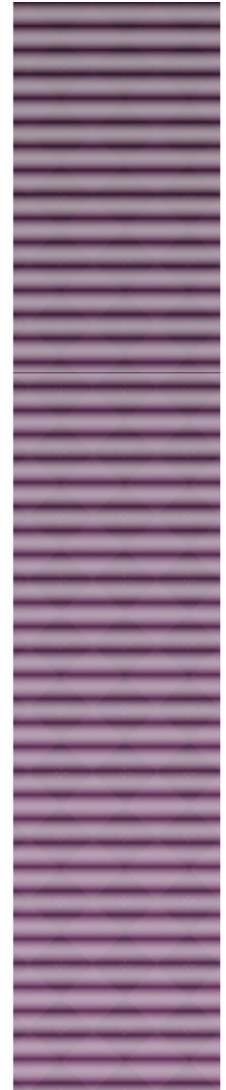
Se trata de un proceso incremental en el que se van paulatinamente involucrando, durante todo el tiempo que dura la construcción del sistema, un número creciente de módulos, para terminar probando el sistema como conjunto completo.



## **4.2 Pruebas de integración**

Existen dos enfoques fundamentales para llevar a cabo las pruebas:

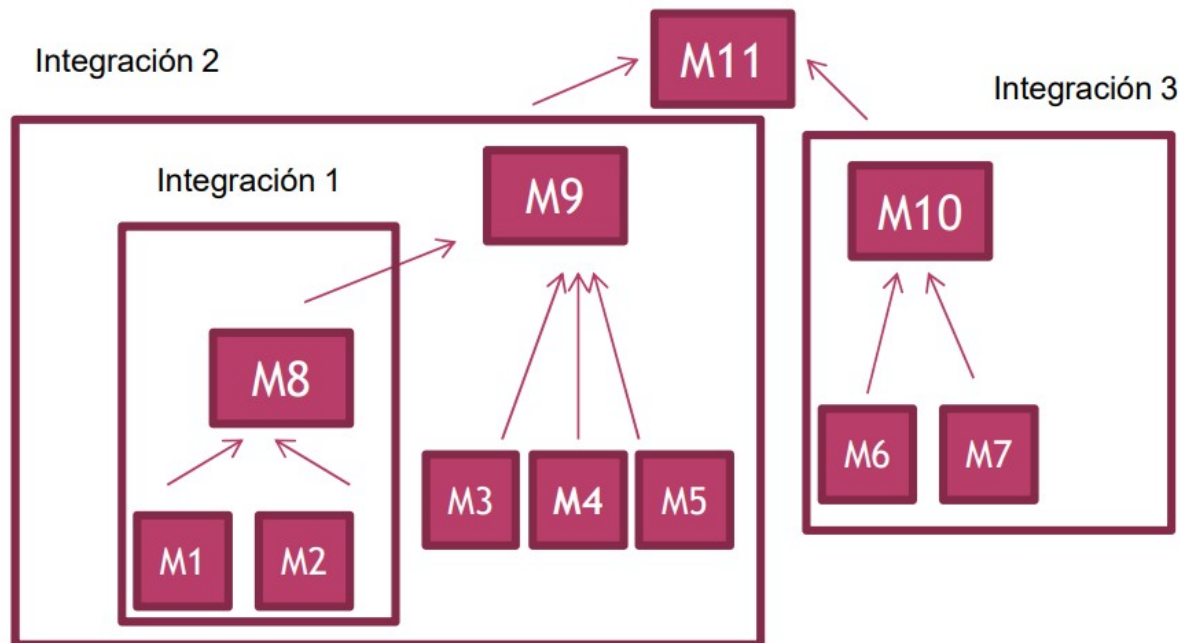
- Pruebas de integración ascendente
- Pruebas de integración descendente





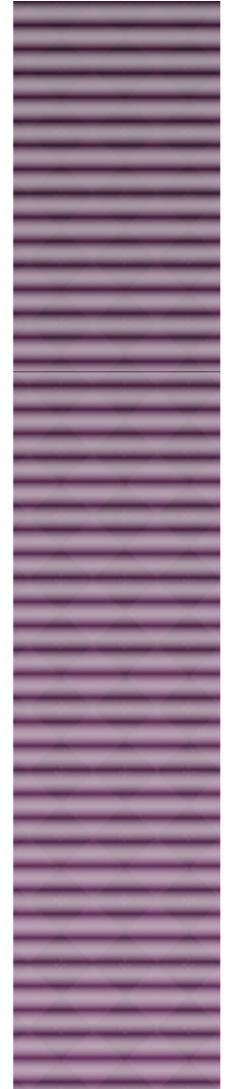
## 4.2.1 Pruebas de integración ascendente

- Se prueban primero los componentes de menor nivel, luego los que invocan o utilizan de algún modo los anteriores, y así sucesivamente.
- Este proceso se repite hasta alcanzar el nivel más alto: aquel en el cual se ven involucrados todos los componentes.

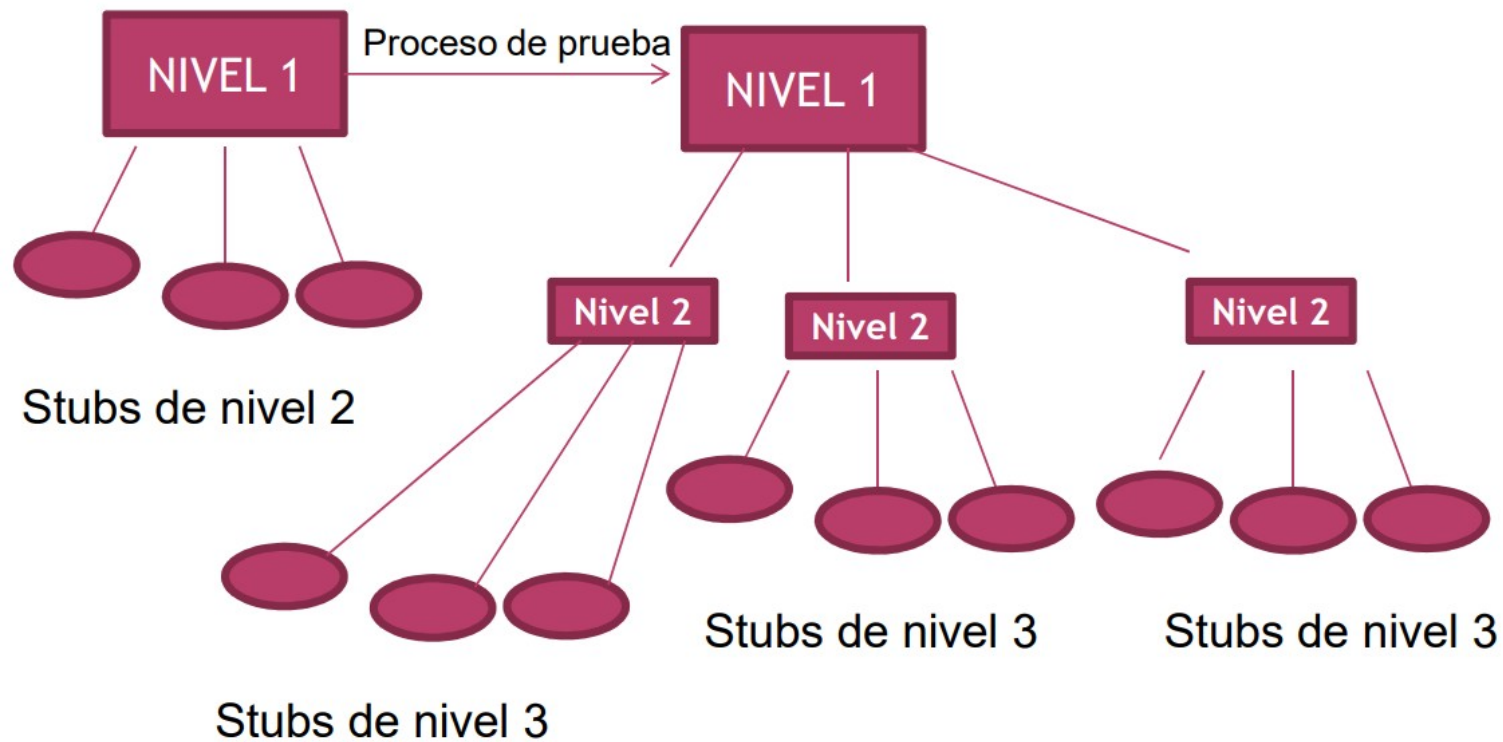


## 4.2.2 Pruebas de integración descendente

- El proceso consiste en probar primero el componente de más alto nivel, después aquellos que son utilizados por dicho componente, y así sucesivamente con el resto de componentes hasta llegar al nivel más bajo.
- Existe una particularidad: cuando un componente que se está probando invoca a otro que aún no ha sido probado, debe crearse lo que se denomina un **procedimiento tonto** (stub), una implementación temporal de una parte del programa con el único propósito de poder realizar la prueba.



## 4.2.2 Pruebas de integración descendente



## 4.2.3 Ventajas y desventajas

Integración Ascendente	Integración Descendente
(D) Los componentes más importantes son los últimos en ser probados, lo cual a menudo postpone la detección de fallos importantes	(V) Los casos de prueba se pueden definir en función de las funcionalidades del sistema
(D) El flujo de control es dependiente de los elementos de más alto nivel, que se prueban los últimos, lo cual dificulta las pruebas en sistemas muy dependientes de una estricta secuencia temporal.	(V) Las cuestiones importantes de funcionalidad se prueban al principio, lo cual permite la detección temprana de fallos.
(V) Son especialmente adecuadas en la programación orientada a objetos, pues en este modelo de programación, los objetos se prueban primero independientemente y luego en conjunción con otros.	(D) Puede ser necesario programar muchos stubs, lo que constituye un importante esfuerzo de desarrollo de código que luego no será utilizado.



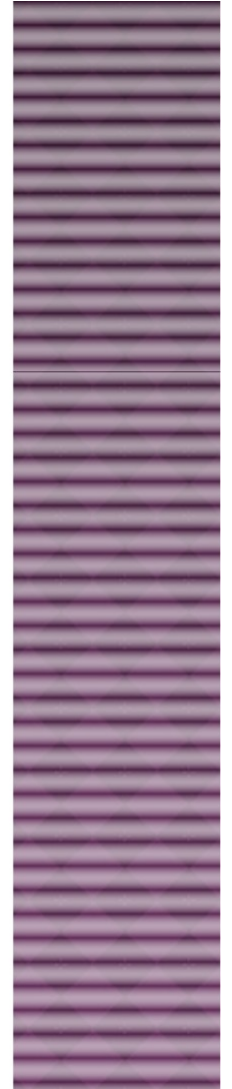
## 4.3 Pruebas de validación

- La validación se consigue cuando el software funciona de acuerdo con las expectativas razonables del cliente definidas en el **documento de especificación de requisitos del software**.
- Se llevan a cabo una serie de pruebas que demuestran la conformidad con los requisitos.
- Las técnicas a utilizar son:
  - **Prueba Alfa:** se lleva a cabo por el cliente o usuario en el lugar de desarrollo. El cliente utiliza el software de forma natural bajo la observación del desarrollador que irá registrando los errores y problemas de uso.



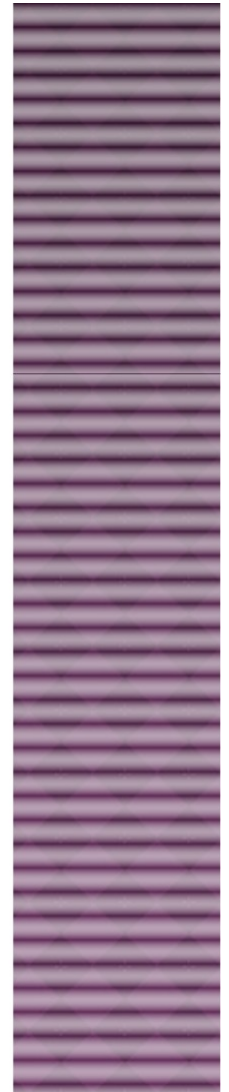
## 4.3 Pruebas de validación

- **Prueba beta:** se lleva a cabo por los usuarios finales del software en su lugar de trabajo. El desarrollador no está presente. El usuario registra todos los problemas que encuentra e informa al desarrollador en los intervalos definidos en el plan de prueba. Como resultado de los problemas informados, el desarrollador lleva a cabo las modificaciones y prepara una nueva versión del producto.



## 4.3 Pruebas del sistema

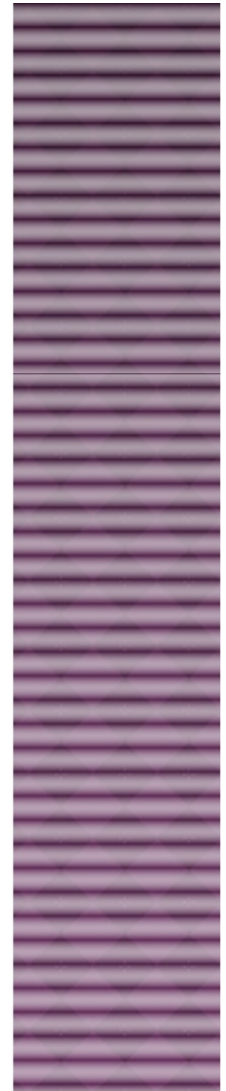
- La prueba del sistema está formada por un conjunto de pruebas cuya misión es ejercitar profundamente el software.
- Los principales puntos de comprobación durante la pruebas del sistema son la seguridad, la velocidad, la exactitud y la fiabilidad.
- Dentro de las pruebas del sistema se suelen realizar las siguientes:
  - **Pruebas de rendimiento:** verifican que el software alcanza los objetivos de rendimiento especificados por el cliente y expresados en el documento de requisitos. Estas pruebas son realizadas internamente por el equipo de desarrollo, si bien los resultados de las mismas frecuentemente se proporcionan al cliente para su comprobación.





## 4.3 Pruebas del sistema

- **Pruebas de instalación:** se realizan para asegurar tanto el funcionamiento correcto de las opciones y funcionalidades de la instalación como para asegurar que todos los componentes necesarios se instalaron correctamente. Es importante probar las diferentes opciones de instalación, así como el buen funcionamiento de las opciones de desinstalación.
- **Pruebas de desgaste:** llevan al sistema más allá de sus límites normales de operación con el objeto de comprobar que no sólo es capaz de soportar los límites de carga marcados, sino que también puede funcionar correctamente por encima de los mismos.



## 4.3 Pruebas del sistema

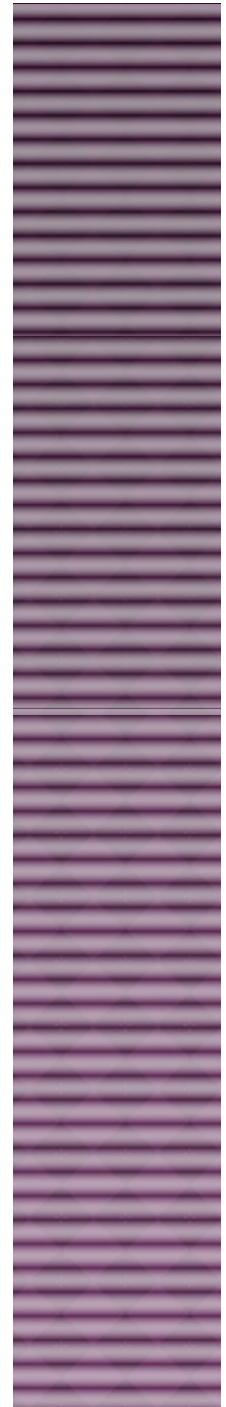
- **Pruebas de conformidad:** consisten en verificar si el comportamiento del software es conforme con ciertas especificaciones a las que debería ajustarse según lo establecido en los requisitos. Por ejemplo, aquellos programas que tratan información sobre datos de carácter personal registrados en soporte físico, deberán de cumplir con la normativa vigente sobre protección de datos. Frecuentemente estas pruebas las realiza una organización externa especialmente acreditada para esta función. Dicha organización emite un veredicto final sobre si el software es o no conforme con la normativa vigente.



## 4.3 Pruebas del sistema

- **Pruebas de regresión:** son aquellas que se llevan a cabo sobre un componente para verificar que los cambios realizados no han producido efectos no deseados. El fundamento de este tipo de pruebas es comprobar que un componente que ya había sido probado no se ha visto afectado por modificaciones en otros componentes.

Cuando el sistema en desarrollo reemplaza a un sistema previo en funcionamiento, consisten en verificar que el rendimiento del nuevo sistema es, al menos, tan bueno como el del sistema antiguo.



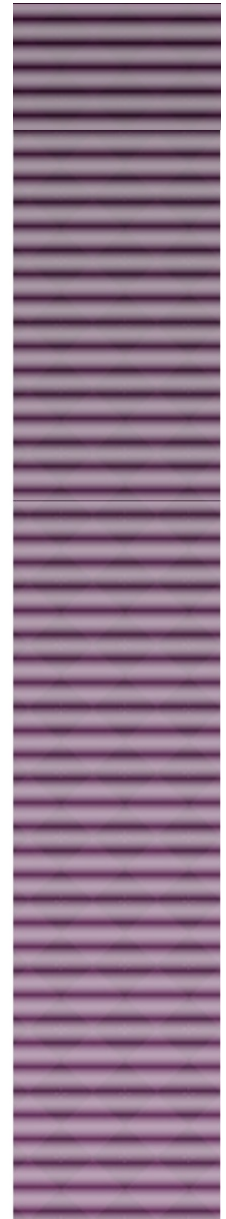
## **4.3 Pruebas del sistema**

- **Pruebas de recuperación:** cuando el software termina de manera incorrecta o incontrolada, cuando las operaciones en curso no pueden completarse debido a una terminación abrupta por causas indeterminadas, cuando se produce un fallo de hardware o, en general, cuando el sistema sufre algún desastre, es necesario poner en marcha mecanismos de recuperación. Las pruebas de recuperación verifican que dichos mecanismos funcionan correctamente.
- **Pruebas de configuración:** evalúan el software en los diferentes entornos que pueden configurarse según lo especificado en los requisitos.



## 5. Casos de prueba

- Un caso de prueba es un conjunto de entradas, condiciones de ejecución y resultados esperados, que han sido desarrollados para un objetivo particular, como por ejemplo, ejercitar un camino concreto de un programa o verificar el cumplimiento de un determinado requisito.
- Para llevar a cabo un caso de prueba, es necesario definir las precondiciones y post-condiciones, identificar unos valores de entrada y conocer el comportamiento que debería tener el sistema ante dichos valores.
- Tras realizar ese análisis e introducir dichos datos en el sistema, se observa si el comportamiento es el previsto o no y por qué.
- De esta forma se determina si el sistema ha pasado o no la prueba.



## 5. Casos de prueba

Ejemplo: supongamos que tras la programación de una calculadora sencilla, se desea probar la operación sumar. Si suponemos que la calculadora tiene 3 botones rotulados como “1”, “+” y “=”, y una caja de texto donde se muestran los resultados, las instrucciones para ejecutar la prueba serían:

Presione el botón etiquetado como 1.

Presione el botón etiquetado como +.

Presione el botón etiquetado como 1.

Presione el botón etiquetado como =.

Observe y anote el resultado.

El resultado de la prueba será considerado como correcto si el resultado obtenido es 2 e incorrecto en otro caso.

## **5. Casos de prueba**

- En términos generales, se calcula que entre el 30% y el 40% del esfuerzo total de un proyecto se dedica a actividades de prueba.
- Este porcentaje es mucho mayor cuando el software controla procesos críticos para la seguridad de las personas como puede ser el software médico, el de control de centrales nucleares, el control de vuelo en aeronaves, etc.

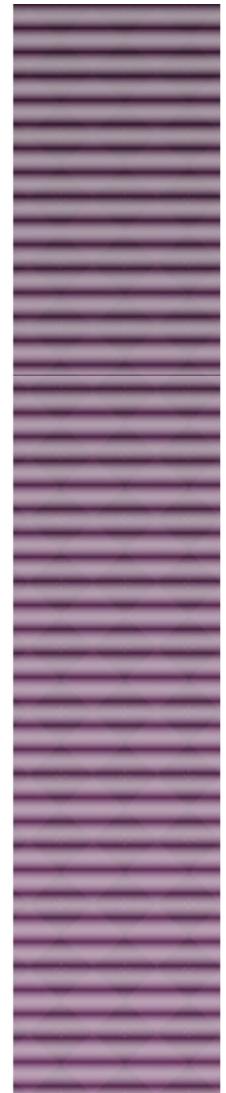


## 5. Casos de prueba

- Durante el proceso de prueba se han de preparar un número de casos de prueba lo suficientemente significativo como para cubrir todas las posibles causas de fallo -> prueba completa o prueba exhaustiva -> en la mayoría de ocasiones no es factible.
- Se denomina prueba **exhaustiva** o prueba **completa** a una prueba ideal que proporcionaría la seguridad de que se han comprobado todas y cada una de las posibles causas de fallo.

## 5.1 Limitaciones en casos de prueba

- **Imposibilidad de hacer pruebas exhaustivas:** no es posible asegurar al 100% que un software esté libre de errores. Sólo una prueba exhaustiva permitiría corroborar que no hay errores.
- **Selección de los casos de pruebas:** un problema importante es decidir cuáles son los casos de prueba más adecuados en cada situación. Los criterios de selección dependen también de decisiones personales basadas en la intuición y la experiencia, lo que aumenta la incertidumbre acerca de su validez.



## 5.1 Limitaciones en casos de prueba

- **Selección de los equipos de prueba:** las pruebas deberían ser llevadas a cabo, en el caso ideal, por personas independientes, completamente imparciales con respecto al software a probar y, a ser posible, que no hubieran participado en su realización directamente.
- **Finalización de las pruebas:** la capacidad para decidir si se puede terminar de hacer pruebas o no, y cuándo, es algo difícil y para lo que se requiere experiencia.

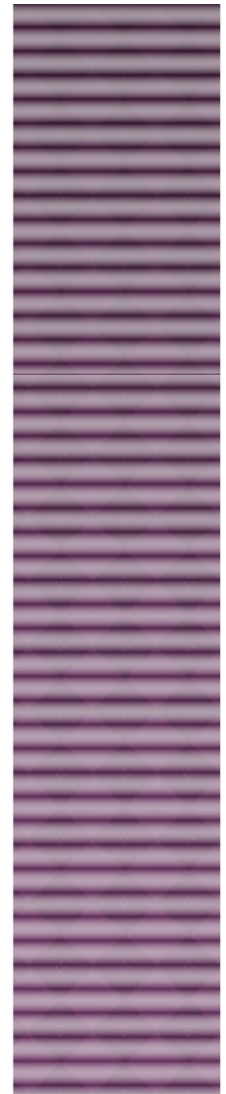
## 5.2 Técnicas casos de prueba

- Para llevar a cabo el diseño de casos de prueba se utilizan dos técnicas o enfoques:
  - **Pruebas de caja blanca:** se centran en validar la estructura interna del programa. Necesitan conocer los detalles procedimentales del código.
  - **Técnicas de caja negra:** se centran en validar los requisitos funcionales sin fijarse en el funcionamiento interno del programa. Necesitan saber la funcionalidad que el código ha de proporcionar.
- Estas pruebas no son excluyentes y se pueden combinar para descubrir diferentes tipos de errores.



## 5.2.1 Pruebas de caja blanca

- Son aquellas que están basadas en información acerca de cómo se ha diseñado o programado el software.
- En el contexto de la prueba de software, se habla de pruebas de caja blanca cuando la propia naturaleza de la prueba impone la observación del código del módulo a probar.
- Frecuentemente, el objetivo es probar todos los caminos posibles de ejecución en el código, lo cual se mide mediante la noción de **cobertura** de código.



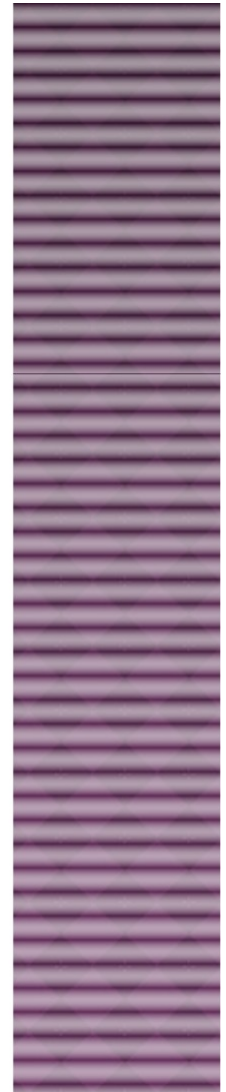
## 5.2.1 Pruebas de caja blanca

- La **cobertura** es una medida porcentual que indica la cantidad de código que ha sido probado.
- La medida de cobertura de un software siempre será mejor cuanto más cerca esté del 100% -> 95% suficiente.
- Las técnicas de cobertura de código (la cobertura de rutas, de sentencias o la cobertura de condiciones) son técnicas de caja blanca que permiten automatizar las pruebas de software.



## **5.2.1 Pruebas de caja blanca**

- **Cobertura de sentencias.** Se trata de generar los casos de prueba necesarios para que cada sentencia o instrucción del programa se ejecute al menos una vez.
- **Cobertura de decisiones.** Consiste en escribir casos suficientes para que cada decisión tenga, por lo menos una vez, un resultado verdadero y, al menos una vez, uno falso.
- **Cobertura de condiciones.** Se trata de diseñar tantos casos como sea necesario para que cada condición de cada decisión adopte el valor verdadero al menos una vez y el falso al menos una vez.





## 5.2.1.1 Prueba del camino básico

- Es una prueba de caja blanca cuyo objetivo es verificar que todas las instrucciones del programa se ejecutan al menos una vez y que se recorren todos los caminos independientes.
- 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 del programa.
- **Camino independiente** es aquel que introduce al menos una sentencia de procesamiento (o condición) que no estaba considerada en el conjunto de caminos independientes calculados hasta ese momento.
- Para obtener el conjunto de caminos independientes se construirá el **grafo de flujo asociado** y se calculará su **complejidad ciclomática**.



## 5.2.1.1 Prueba del camino básico

### Pasos:

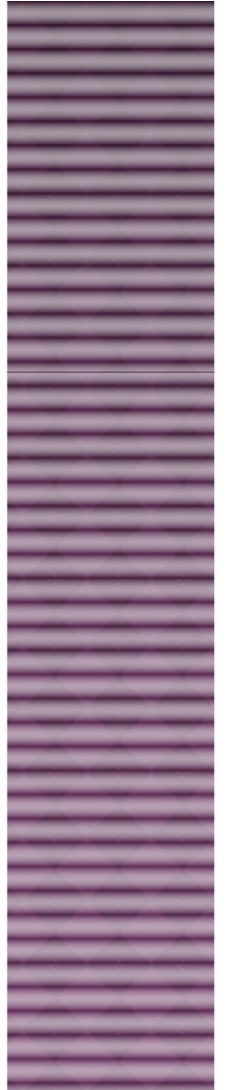
1. Partiendo de un código, de un pseudocódigo o de un diagrama de flujo hay que elaborar el **grafo de flujo** asociado al programa.
2. Calcular la complejidad ciclomática estableciendo el número de caminos independientes. Se denomina **independiente** a cualquier camino que introduce al menos una sentencia de procesamiento no considerada anteriormente.
3. Seleccionar un conjunto de caminos básicos.
4. Generar casos de prueba para cada uno de los caminos del paso anterior.



## 5.2.1.1 Camino básico (Paso 1)

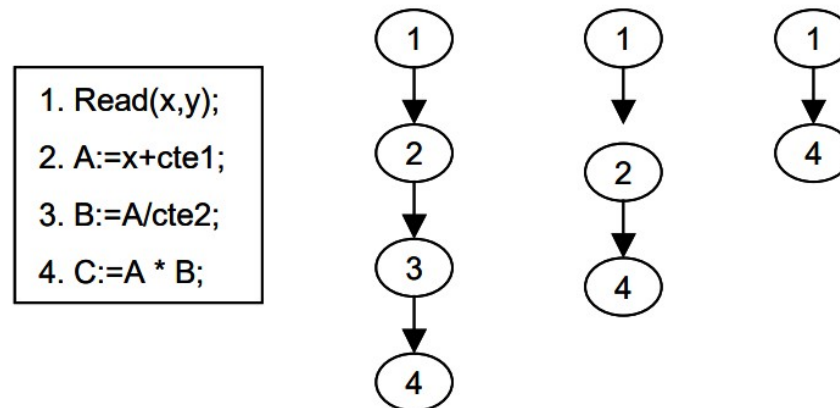
### Paso 1: Grafo de flujo

- Existe un solo nodo inicial y un solo nodo final.
- Un grafo de flujo representa un programa de modo que se observan con claridad los diferentes puntos donde existen cambios de dirección, permitiendo analizar los diversos caminos que llevan del inicio al fin del mismo.
- La ejecución de cualquier programa se realiza en forma secuencial, instrucción por instrucción, a menos que se realice un cambio de dirección, debida a una orden específica (bucles for o while) o a la evaluación de una condición.



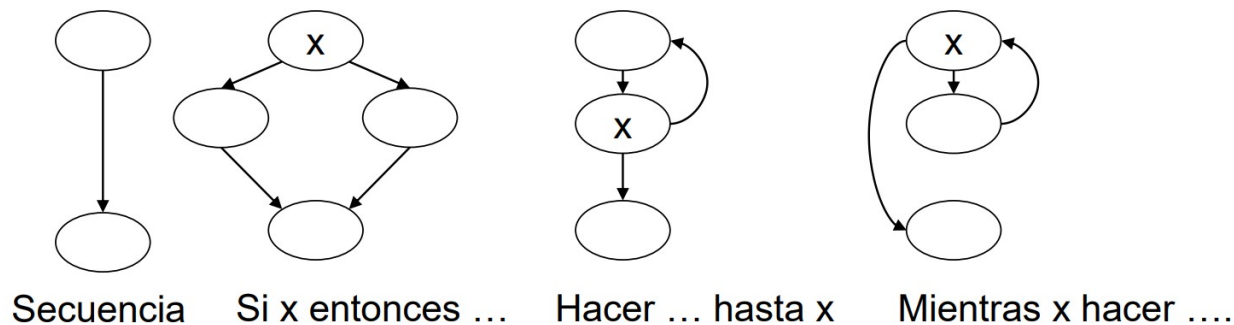
- El grafo contendrá dos tipos de elementos:
  - **Nodos(vértices):** que representan una o más instrucciones secuenciales o instrucciones condicionales.
  - **Arcos(aristas):** que representan cambios de dirección.
- Las instrucciones secuenciales (las que no son condicionales), pueden representarse una sola en cada nodo o varias de ellas en uno solo, ya que si se ejecuta la primera se ejecutarán las siguientes una tras otra.

Es conveniente agruparlas en un único nodo.



## Paso1: Obtención del grafo de flujo:

1. Señalar sobre el código la condición de cada decisión tanto en sentencias “si-entonces” como en los bucles “mientras” o “hasta”.
2. Agrupar el resto de las sentencias situadas entre cada dos condiciones según los esquemas de representación de las estructuras básicas.



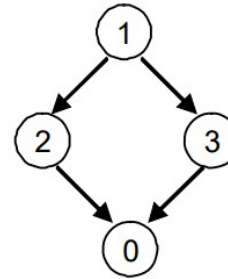
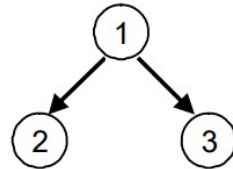
3. Numerar tanto las condiciones como los grupos de sentencias, de manera que se les asigne un identificador único. Es conveniente identificar los nodos que representan condiciones y señalar cuál es el resultado que provoca la ejecución de cada una de las aristas que surgen de ellos.

## Ejemplos diagramas flujo:

ej1:

```

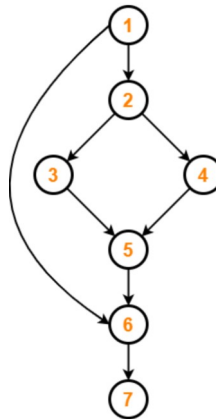
1  if (x<20)
2  then a=1;
3  else a=-1;
    
```



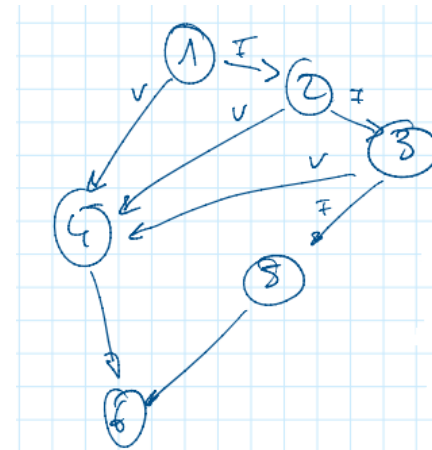
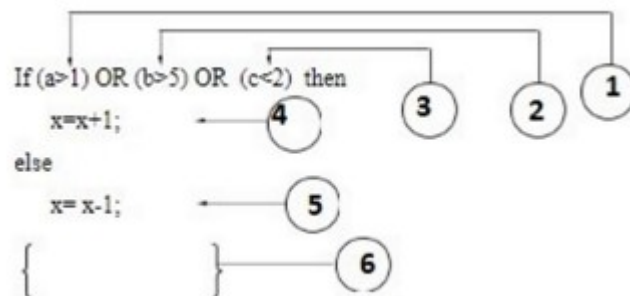
ej2:

```

1.  IF A = 354
2.  THEN IF B > C
3.  THEN A = B
4.  ELSE A = C
5.  END IF
6.  END IF
7.  PRINT A
    
```



ej3:



## ej4:

```
public class Cuenta {  
  
    public static void main(String[] args) {  
        int i=0;  
        int contPares=0;  
        int contImpares=0;  
        int num;  
  
        Scanner sc = new Scanner("System.in");  
  
        while (i<11){  
            num=sc.nextInt();  
            if (num%2==0)  
                contPares++;  
            else  
                contImpares++;  
            i++;  
        }  
  
        System.out.println("El número de pares es: " + contPares);  
        System.out.println("El número de impares es: " + contImpares);  
    }  
}
```

1

2

3

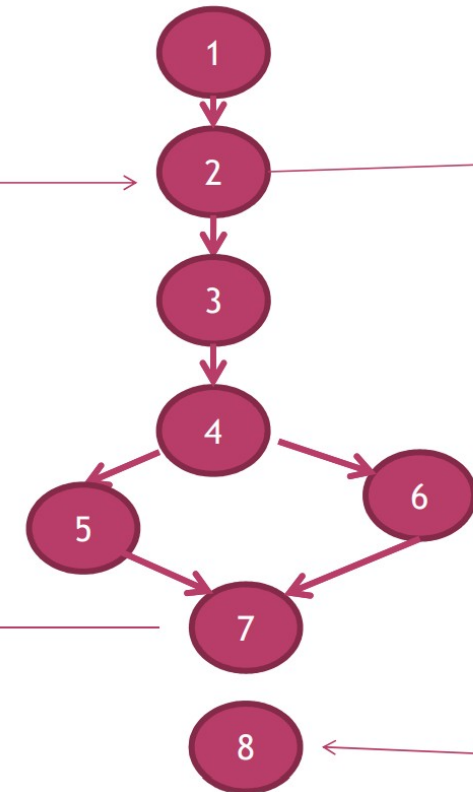
4

5

6

7

8





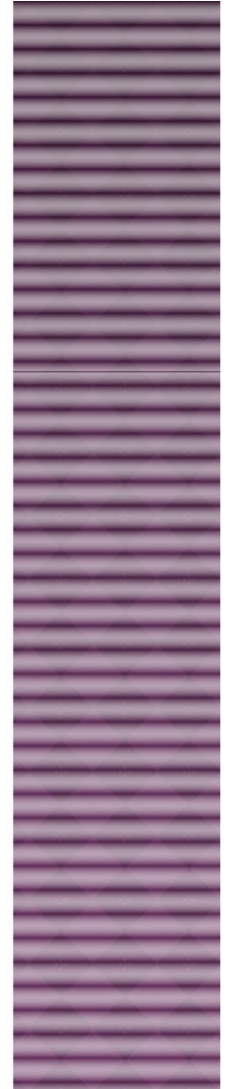
## 5.2.1.1 Camino básico (Paso 2)

### Paso 2: Calcular complejidad ciclomática

- Es una métrica que nos aporta una medida cuantitativa de la complejidad lógica de un programa.
- Determina el número de caminos independientes del conjunto básico del programa y nos da una cota superior para el número de pruebas que debemos realizar para asegurar que al menos cada sentencia del programa se va a ejecutar una vez.
- Si la complejidad ciclomática de una unidad de software es superior a 10 es muy recomendable refactorizar el código.



- La complejidad ciclomática  $V(G)$  de un grafo de control se calcula:
  - $V(G) = A - N + 2$ , 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 condición.
- La complejidad ciclomática para el ejemplo 2(ej2) es:
  - $V(G) = \text{Aristas} - \text{Nodos} + 2 = 8 - 7 + 2 = 3$
  - $V(G) = \text{Nodos predicado} + 1 = 2 + 1 = 3$
- La complejidad ciclomática para el ejemplo 4(ej4) es:
  - $V(G) = \text{Aristas} - \text{Nodos} + 2 = 9 - 8 + 2 = 3$
  - $V(G) = \text{Nodos predicado} + 1 = 2 + 1 = 3$



## **5.2.1.1 Camino básico (Paso 3)**

**Paso 3: Seleccionar conjunto de caminos independientes**

- El valor de  $V(G)$  nos da el número de caminos independientes del conjunto básico de un programa.
- Un camino independiente es cualquier camino del programa que introduce, por lo menos, un nuevo conjunto de sentencias de proceso o una condición.
- En términos del diagrama de flujo, un camino independiente está constituido por lo menos por una arista que no haya sido recorrida anteriormente a la definición del camino.

## 5.2.1.1 Camino básico (Paso 3)

Para el ejemplo 2, el conjunto de caminos independientes será:

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

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

Camino 3: 1 - 2 - 4 - 6

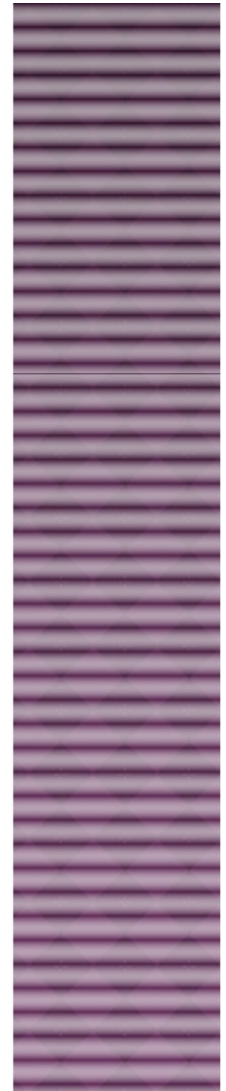
Camino 4: 1 - 4 - 6

Para el ejemplo 4, el conjunto de caminos independientes será:

Camino 1: 1 - 2 - 8

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

Camino 3: 1 - 2 - 3 - 4 - 6 - 7 - 2 - 8



## 5.2.1.1 Camino básico (Paso 4)

### Paso 4: Generar casos de prueba para cada camino independiente

- El último paso de la prueba del camino básico es construir los casos de prueba que fuerzan la ejecución de cada camino.
- Con el fin de comprobar cada camino, debemos escoger los casos de prueba de forma que las condiciones de los nodos predicado estén adecuadamente establecidas.

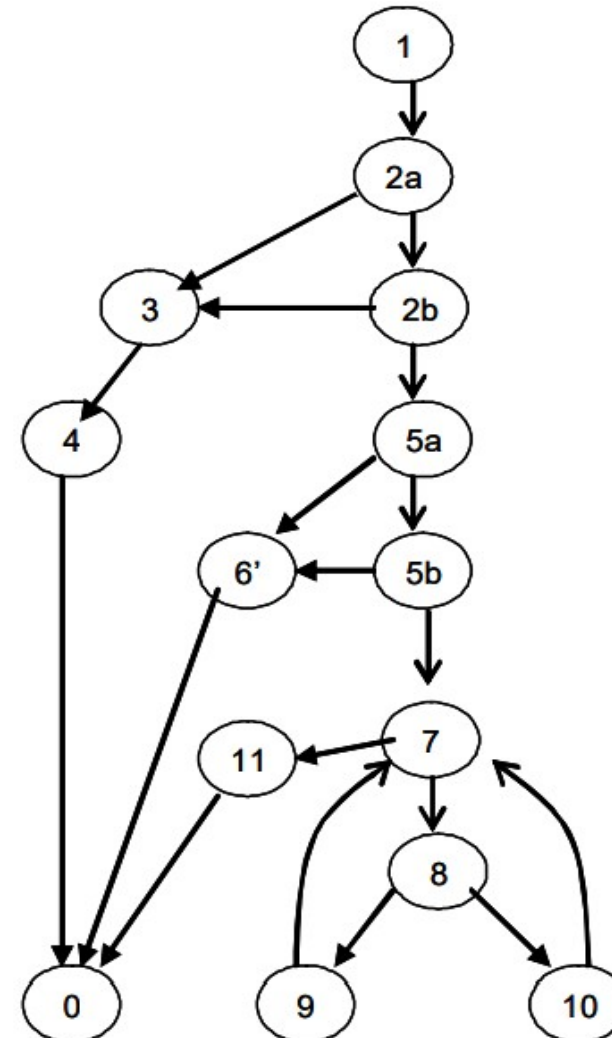
#### Casos prueba ejemplo 4:

Camino	Caso de prueba	Resultado Esperado
1	Escoger algún valor de $i$ tal que NO se cumpla la condición $i < 11$ . Por ejemplo, $i = 11$	Visualizar el número de pares y el de impares.
2	Escoger algún valor de $i$ tal que SÍ se cumpla la condición $i < 11$ y escoger un valor de $num$ tal que SÍ se cumpla $num \% 2 = 0$ . Por ejemplo, $i = 6$ y $num = 5$	Suma 1 al contador de pares
3	Escoger algún valor de $i$ tal que SÍ se cumpla la condición $i < 11$ y escoger un valor de $num$ tal que NO se cumpla $num \% 2 = 0$ . Por ejemplo, $i = 1$ y $num = 5$	Suma 1 al contador de impares

## 5.2.1.1 Camino básico (Ejemplo)

```
1 lee x, y
2 Si (x<=0) o (y<=0) entonces
3   Escribe "deben ser no negativos;
4   regresa -1
5 Si (x=1) o (y=1) entonces
6   regresa 1
7 Mientras (x <> y)
8   Si (x>y) entonces
9     x = x - y
10  de otro modo y = y - x
11 regresa x
```

(a) Código de máximo común divisor



(b) Grafo correspondiente

## 5.2.1.1 Camino básico (Ejemplo)

- $V(G) = \text{Aristas} - \text{Nodos} + 2 = 19 - 14 + 2 = 7$
- $V(G) = \text{Nodos predicado} + 1 = 6 + 1 = 7$

número	Camino
1	1-2a-2b-5a-5b-7-8-9-7-11-0
2	1-2a-2b-5a-5b-7-8-10-7-11-0
3	1-2a-2b-5a-5b-7-11-0
4	1-2a-2b-5a-5b-6-0
5	1-2a-2b-5a-6-0
6	1-2a-2b-3-4-0
7	1-2a-3-4-0





- Primer camino: 1-2a-2b-5a-5b-7-8-9-7-11-0: La primera condición (2a y 2b) indican que deberá cumplirse  $x > 0$  y  $y > 0$ ; la siguiente condición (5a y 5b) indican que  $x \neq 1$  y  $y \neq 1$ ; la condición 8 lleva a  $x > y$  y ya no debe volver al ciclo. Por lo tanto, dos posibles valores de entrada serán  $x=4$ ,  $y=2$ .
- Segundo camino: 1-2a-2b-5a-5b-7-8-10-7-11-0: Es similar al anterior, pero debe cumplirse  $x < y$ ; entonces puede tomarse  $x=3$ ,  $y=6$ .
- Tercer camino: 1-2a-2b-5a-5b-7-11-0: Es similar, pero  $x=y$ ; entonces se usará  $x=37$ ,  $y=37$
- Cuarto camino: 1-2a-2b-5a-5b-6-0: El inicio es igual, pero  $y=1$ ; se propone  $x=7$ ,  $y=1$
- Quinto camino: 1-2a-2b-5a-6-0: Similar al anterior,  $x=1$  y  $y > 1$ ; se propone  $x=1$ ,  $y=13$
- Sexto camino: 1-2a-2b-3-4-0: En este camino  $x > 0$ ,  $y \leq 0$ ; se propone  $x=18$ ,  $y=0$
- Séptimo camino: 1-2a-3-4-0: Aquí  $x \leq 0$ ,  $y > 0$ ; se propone  $x=-1$ ,  $y=3$

Resumiendo los casos de prueba se tiene:

Entrada	Salida
$x = 4, y = 2$	2
$x = 3, y = 6$	3
$x = 37, y = 37$	37
$x = 7, y = 1$	1
$x = 1, y = 13$	1
$x = 18, y = 0$	Mensaje "deben ser no negativos"
$x = -1, y = 3$	Mensaje "deben ser no negativos"

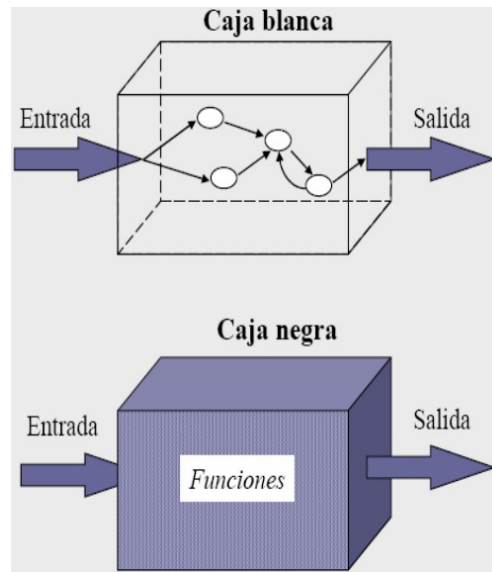
## **5.2.1 Pruebas de caja negra**

- Son aquellas donde los casos de prueba se basan solamente en el comportamiento de la entrada y salida de datos.
- No hace falta conocer la estructura interna del programa.
- Se pretende obtener casos de prueba que demuestren que las funciones del software son operativas, es decir, que las salidas que devuelve la aplicación son las esperadas en función de las entradas que se proporcionen.
- El sistema se considera como una caja negra cuyo comportamiento sólo se puede determinar estudiando las entradas y las salidas que devuelve en función de las entradas suministradas.



## 5.2.1 Pruebas de caja negra

- Se refieren al hecho de pensar en un sistema como si las entradas de datos se conocieran, pero su funcionamiento interno se ignorase (automóvil, televisor, lavadora, etc).
- Se parte de la especificación de requisitos del módulo con el objetivo de comprobar si lleva correctamente a cabo aquellas funciones que se esperan de él.



## 5.2.1 Pruebas de caja negra

- **Ventajas:** sencillas de ejecutar ya que el encargado de la prueba se limita a introducir un conjunto de datos de entrada y estudiar la salida. -> Fácil automatización.
- **Desventajas:** alto coste de mantenimiento de las pruebas automatizadas (cambios en la aplicación suelen influir en la validez de las pruebas de caja negra), y el riesgo de dejar parte del código sin probar si no se diseñan correctamente.



## **5.2.1 Pruebas de caja negra**

- Existen diferentes técnicas para confeccionar los casos de prueba de caja negra: clases de equivalencia, análisis de valores límite, métodos basados en grafos, pruebas de comparación, etc.
- Un problema importante en las pruebas de caja negra es la incertidumbre sobre la cobertura de las mismas. Se emplean técnicas heurísticas.
- La técnica de las clases de equivalencia permite reducir la casuística de manera considerable.



## **5.2.1.1 Clases de equivalencia**

- Es un método de prueba de Caja Negra que divide todo el conjunto de posibles datos de entrada de un programa en clases de datos (particiones) a partir de las cuales se pueden derivar casos de prueba.
- Este método intenta dividir el dominio de entrada de un programa en un número finito de clases de equivalencia. De tal modo que se pueda asumir razonablemente que una prueba realizada con un valor representativo de cada clase es equivalente a una prueba realizada con cualquier otro valor de dicha clase.





## **5.2.1.1 Clases de equivalencia**

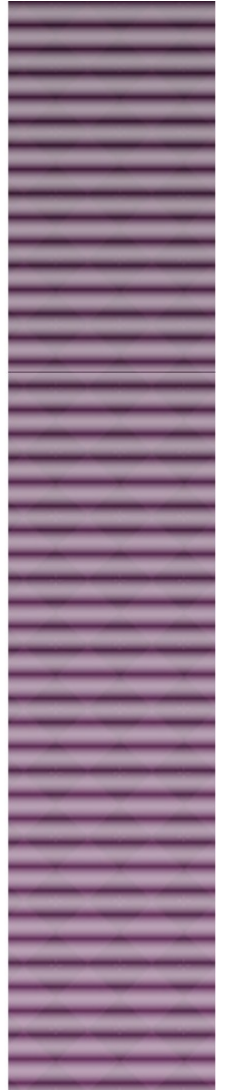
- Esto quiere decir que si el caso de prueba correspondiente a una clase de equivalencia detecta un error, el resto de los casos de prueba de dicha clase de equivalencia deben detectar el mismo error.
- Y viceversa, si un caso de prueba no ha detectado ningún error, es de esperar que ninguno de los casos de prueba correspondientes a la misma clase de equivalencia encuentre ningún error.
- Las clases de equivalencia se obtienen de las condiciones de entrada descritas en las especificaciones del software a desarrollar.





## **5.2.1.1 Clases de equivalencia**

- El diseño de casos de prueba según esta técnica consta de dos pasos:
  1. Identificar las clases de equivalencia
    - Clases de equivalencia válidas.
    - Clases de equivalencia invalidas
  2. Identificar los casos de prueba a partir de las clases de equivalencia.



## 5.2.1.1 Clases de equivalencia

- Las clases de equivalencia se obtienen de las condiciones de entrada descritas en las especificaciones del software a desarrollar (normalmente una frase en la especificación) y dividiéndola en dos o más grupos.
- Se definen dos tipos de clases de equivalencia:
  - **Clases de equivalencia válidas**, que representan entradas válidas al programa.
  - **Clases de equivalencia no válidas**, que representan valores de entrada erróneos
- Este proceso es heurístico, existen un conjunto de criterios que ayudan a su identificación.



## 5.2.1.2 Definición clases de equivalencia

Condición entrada	Ejemplo	Clases de equivalencia válidas	Clases de equivalencia no válidas
Un valor específico	"..Introducir <b>cinco</b> valores.."	1 clase que contemple dicho valor	2 clases que representen un valor por encima y otro por debajo
Un rango de valores	"...Valores entre <b>0 y 10</b> ..."	1 Clase que contemple los valores del rango	2 clases fuera del rango, una por encima y otra por debajo
Conjunto finito valores	"..Valores <b>2, 4, 6 o 10</b> ..."	Tantas clases como valores admitidos	2 clases fuera del conjunto de valores, una por encima y otra por debajo. Tantas clases como valores intermedios
Conjunto de opciones admitidas	"... <b>lunes, martes o jueves</b> ..."	Tantas clases como valores admitidos	1 clase para un valor no admitido
Condición booleana	"Introducir un número <b>par</b> "	1 clase que cumpla la condición	1 clase que no cumpla
Clases menores o distintos rangos tratados de manera diferente por el programa	" <b>Las personas menores de 25 años tendrán una bonificación del 10% y las personas entre 25 y 50 una bonificación del 15%</b> "	Se tratan de manera individual igual que los rangos de valores. <b>0&lt;=edad&lt;25;</b> <b>25&lt;=edad&lt;=50;</b>	Se tratan de manera individual igual que las clases para rangos de valores.

## **5.2.1.3 Identificar casos de prueba**

- Para crear los casos de prueba a partir de las clases de equivalencia se han de seguir los siguientes pasos:
  1. Asignar a cada clase de equivalencia un número único.
  2. Generar el menor número de casos de prueba que solapen todas las clases de equivalencia válidas.  
Teniendo en cuenta que se tiene que generar un caso de prueba por cada salida diferente que provoquen los datos de entrada.
  3. Generar un caso de prueba individual por cada una de las clases de equivalencia inválida

## **5.2.1.3 Identificar casos de prueba**

- La razón de cubrir con casos individuales las clases no válidas es que ciertos controles de entrada pueden enmascarar o invalidar otros controles similares.
- Por ejemplo, si tenemos dos clases válidas: “introducir cantidad entre 1 y 99” y “seguir con letra entre A y Z”, el caso [105, 1] (dos errores) puede dar como resultado 105 fuera de rango de cantidad, y no examinar el resto de la entrada no comprobando así la respuesta del sistema ante una posible entrada no válida

## **5.2.1.4 EJEMPLO 1**

- Construcción de una batería de pruebas para detectar posibles errores en la construcción de los identificadores de un hipotético lenguaje de programación.
- Las reglas que determinan sus construcción sintáctica son:
  - No debe tener más de 15 ni menos de 5 caracteres.
  - El juego de caracteres utilizables es:
    - Letras (Mayúsculas y minúsculas)
    - Dígitos (0,9)
    - Guión (-)
- El guión no puede estar ni al principio ni al final, pero puede haber varios consecutivos.
- Debe contener al menos un carácter alfabético.
- No puede ser una de las palabras reservadas del lenguaje.



## 5.2.1.4 Tabla clases equivalencia

Condición entrada	Clases de equivalencia válidas	Clases de equivalencia no válidas
Entre 5 y 15 caracteres	$5 \leq n^{\circ} \text{ caracteres Ident.} \leq 15$ (1)	$n^{\circ} \text{ caracteres Id} < 5$ (2) $15 < n^{\circ} \text{ caracteres}$ (3)
El identificador debe estar formado por letras, dígitos y guión	Todos los caracteres del Ident. $\in \{\text{letras, dígitos, guión}\}$ (4)	Alguno de los caracteres del Ident. $\notin \{\text{letras, dígitos, guión}\}$ (5)
El guión no puede estar al principio, ni al final Puede haber varios seguidos en el medio	Identificador sin guiones en los extremos y con varios consecutivos en el medio (6)	Identificador con guión al principio (7) Identificador con guión al final (8)
Debe contener al menos un carácter alfabético	Al menos un carácter del Ident. $\in \{\text{letras}\}$ (9)	Ningún carácter del Ident. $\in \{\text{letras}\}$ (10)
No puede usar palabras reservadas	El Identificador $\notin \{\text{palabras reservadas}\}$ (11)	un caso por cada palabra reservada (12,13,14....)

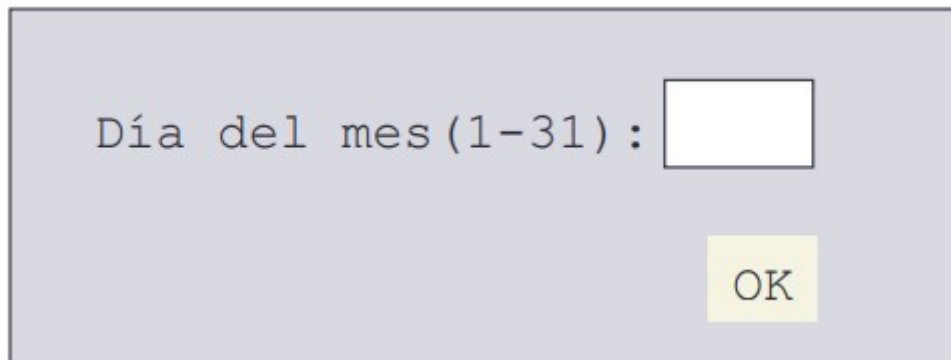


## 5.2.1.4 Derivación casos de prueba

Identificador	Clases de equivalencia cubiertas	Resultado
Num-1---d3	1,4,6,9, 11 (todas las válidas)	El sistema acepta el identificador
Nd3	2	Mensaje error
Num-1-letr3--d32	3	Mensaje error
Nu%m-1---d3	5	Mensaje error
-um-1---d3	7	Mensaje error
num-1---d3-	8	Mensaje error
456-1---23	10	Mensaje error
Integer	12	Mensaje error
El resto de palabras reservadas	13,14..	Mensaje error

## 5.2.1.5 EJEMPLO 2

- El pago retrasado de ciertas facturas mensuales conlleva los siguientes recargos:
  - Si se paga entre el día 1 y 10 no tiene ningún recargo.
  - Si se paga entre el día 11 y 20 tiene un recargo del 2%.
  - Si se paga el día 21 o posterior tiene un recargo del 4%.
- Se ha realizado un programa que a partir de un número entero que representa a un día, visualiza el recargo a aplicar:



Día del mes (1-31):

OK

## 5.2.1.5 Tabla clases equivalencia

Condición entrada	Clases de equivalencia válidas	Clases de equivalencia no válidas
Nº de parámetros	{n=1} (1)	{n<1 (nulo)} (2)
Tipo de parámetros	{dia ∈ N} (entero) (3)	{dia ∈ Numérico no natural} (4.1) {dia ∈ Caracteres} (4.2) (alfabéticos y especiales)
Sin recargo	{dia>0, dia≤10} (5)	{dia<0} (negativo) (6)
Recargo 2%	{dia≥11, dia≤20} (7)	
Recargo 4%	{dia≥21, dia≤30} (8)	{dia>30} (9)

## 5.2.1.5 Derivación casos de prueba

Entrada	Clases de equivalencia cubiertas	Resultado
4	1, 3, 5	SR
18	1, 3, 7	2%
26	1, 3, 8	4%
	2	Mensaje error
3.8	4.1	Mensaje error
'ab'	4.2	Mensaje error
-1	6	Mensaje error
32	9	Mensaje error