ANALISIS Y DISEÑO DE SISTEMAS (3303)

PRUEBAS DE SOFTWARE

Mg. Marcela Daniele

Departamento de Computación

Facultad de Ciencias Exactas, Físico-Químicas y Naturales

Universidad Nacional de Río Cuarto

¿ES POSIBLE CONSTRUIR SOFTWARE QUE NO FALLE?

- *Un programa es correcto si su comportamiento se adapta a su especificación, si verifica su especificación.*

Pruebas de Software – percepciones /definiciones

- *"Las pruebas del software permiten demostrar que no tiene errores".*
- *"El propósito de las pruebas es demostrar que un programa realiza las funciones indicadas correctamente".*
- *"Las pruebas son el proceso de establecer confianza en que un programa hace lo que se supone que debe hacer" .*

- *La prueba es el proceso de ejecución de un programa con la intención de encontrar errores.* Jhon Myers (The art of Software Testing, 1979)
- *Las pruebas de software pueden ser una manera muy eficaz de mostrar la presencia de errores, pero son totalmente inadecuadas para mostrar su ausencia".* Edsger W. Dijkstra

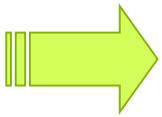
Pruebas de Software – Validación & Verificación

- Conceptos
- Consideraciones
- Objetivos de las pruebas de software
- Tipos de Prueba
- Prueba de Unidades
  - Pruebas de Caja Blanca o Estructural
  - Pruebas de Caja Negra o Funcional

Pruebas de Software – **Validación & Verificación**

Validación

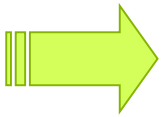
Consiste en comprobar que el sistema y sus componentes cumplen con los requisitos del cliente, definidos en las especificaciones de requerimientos.



¿Estamos resolviendo el problema correcto?

Verificación

Consiste en evaluar el sistema y sus componentes, y asegurar que cada función específica está correctamente implementada.



¿Estamos resolviendo correctamente el problema?

Pruebas de Software – Casos de Prueba

Las **Pruebas de Software** son un tipo de V&V, que consisten en descubrir cómo operan los componentes de un sistema en situaciones representativas y verificar si su comportamiento es el esperado.

- Lo más importante de las **pruebas de software** es **diseñar y crear Casos de Prueba efectivos**.
- Los **Casos de Prueba** son definidos a partir de las situaciones más representativas para las componentes del sistema.
- Existen técnicas y criterios para la definición de los **Casos de Prueba**.
- **Caso de Prueba:** define entradas, condiciones y resultados esperados para situaciones particulares o representativas.

Proceso de Pruebas de Software

El **Proceso de Pruebas de Software** está compuesto por tres procesos principales:

i. Desarrollo y Definición de los **Casos de Prueba.**

ii. Ejecución de los **Casos de Prueba** definidos.

iii. Análisis de resultados obtenidos luego de la ejecución de los **Casos de Prueba.**

Un **Proceso de Pruebas de Software** será **exitoso** cuando encuentre **errores**

Pruebas de Software – **Conceptos**

- **Error:** diferencia entre un valor calculado u observado y el valor correcto esperado. Generalmente provocado por la acción humana que conduce a un resultado incorrecto. Ejemplo: el desarrollador interpreta erróneamente un método del programa y provoca un error porque produce un resultado no esperado.

- **Defecto:** Podrían causar defectos: especificaciones mal definidas o incompletas, algún requerimiento que no puede implementarse, fallas en el diseño o en el código. Por ejemplo, un error de implementación causa un defecto, utilizar el operador "x + y > p" en vez de "x + y => p".

- **Falla:** al ejecutar el programa con algún defecto, no hará lo requerido o se comportará de una forma inadecuada.

  Ejemplo: Si se implementa "x + y > p" en lugar de "x + y => p", cuando se cumpla x+y=p, podría hacer fallar el sistema y en algunos casos con efectos catastróficos.

Pruebas de Software – Consideraciones importantes

- La Prueba es una actividad critica de la IS, se inicia en la primera etapa del ciclo de vida del software y debe realizarse tan sistemáticamente como fuera posible.
- Al generar **casos de prueba**, se deben incluir tanto datos de entrada válidos y esperados como no válidos e inesperados.
- Cada **caso de prueba** define el resultado de salida esperado.
- No será posible la prueba exhaustiva, infinitos escenarios.
- No resulta fácil encontrar los casos de prueba que muestren la evidencia de que el comportamiento deseado será visto en todos los casos o situaciones restantes.
- Las pruebas deben comprobar
  - si el software no hace lo que debe hacer, y
  - si el software hace lo que no debe hacer.

Pruebas de Software – Objetivos

El objetivo principal de las pruebas de software es aportar a la **calidad** del producto. El producto de software debe cumplir con los requerimientos del cliente, brindando una solución al problema planteado.

- "No se puede probar la **calidad**. Si la calidad no está presente antes de comenzar las pruebas, no estará cuando se termine de probar".
- La **calidad** se incorpora en el software a lo largo de todo el proceso de ingeniería del software.

En la actualidad, negocios, empresas, personas, sus maquinarias y dispositivos, sus decisiones, su gestión…….. dependen del funcionamiento del software y de la información que brinda, y una falla podría causas grandes pérdidas.

Pruebas de Software – **Objetivos**

- El proceso de pruebas del software debe garantizar que los **Casos de Prueba** resulten efectivos, que estén correctamente definidos para **encontrar errores**.

- Un caso de prueba será "efectivo" si posee una alta probabilidad de **descubrir al menos un nuevo error**.

- La Prueba debe servir para **Localizar errores** y no solo detectar su presencia.

- La Prueba debe ser **repetible**, aplicar más de una vez la misma entrada sobre el mismo elemento y verificar que produce el mismo resultado.

Pruebas de Software – Facilidad de Prueba

Existen métricas para probar la facilidad de prueba del software

Un software es más fácil de probar si cumple con características de calidad

- **Operatividad:** un buen funcionamiento.
- **Observabilidad:** salidas claras. Código accesible. Se detectan los errores fácilmente.
- **Controlabilidad:** consistente. Permite automatizar las pruebas.
- **Capacidad de Descomposición:** modularizado.
- **Simplicidad:** funcional, estructural y de código.
- **Estabilidad:** pocos cambios y controlados, recuperación correcta e inmediata ante fallos.
- **Facilidad de Comprensión:** diseño claro, comunicación adecuada, documentación accesible, detallada y organizada.

Tipos de Pruebas de Software

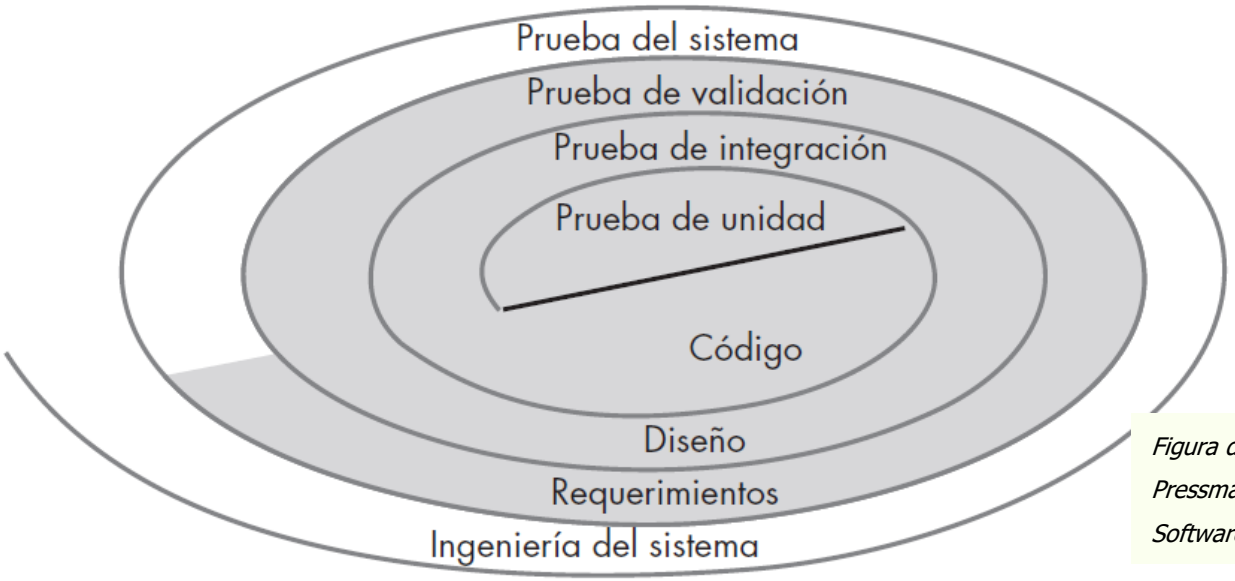


Figura del libro de R. Pressman. Ingeniería de Software. Un enfoque práctico

- **Prueba de Unidad:** las pruebas se enfocan en cada componente de manera individual, garantizando que funcionan adecuadamente como unidad.
- **Prueba de Integración:** los componentes deben ensamblarse o integrarse para formar el paquete de software completo. Pueden realizarse **pruebas de regresión** para asegurar que no se introdujeron nuevos errores.
- **Prueba de Validación:** los requerimientos establecidos en el modelado se validan confrontándose con el software que se construyó.
- **Prueba del Sistema:** se prueba el rendimiento del sistema como un todo.

Estrategia de Pruebas de Software

Una estrategia de software triunfará cuando quienes prueban el software:

- ***Especifican los requerimientos del producto en forma cuantificable***. Características de la calidad: portabilidad, mantenimiento, facilidad de uso, deben especificarse de forma medible.
- ***Establecen de manera explícita y medible los objetivos de las pruebas.*** Efectividad de las pruebas, tiempo medio antes de aparecer una falla, costo p/descubrir y corregir defectos y frecuencia de ocurrencia, hs de trabajo de prueba definidas en el plan de la prueba.
- ***Comprenden al usuario del SW y desarrollan un perfil para c/categoría***. Los CU describen escenarios para c/clase de usuario, enfocan las pruebas en el uso real del producto y reducen el esfuerzo de prueba global.
- ***Desarrollan un plan de prueba que enfatice "pruebas de ciclo rápido".*** Probar en ciclos rápidos cliente-utilidad, genera retroalimentación a pruebas y controla niveles de calidad.
- ***Construyen software "robusto", diseñado para probarse a sí mismo.*** El diseño debe usar técnicas antierrores, el SW diagnostica ciertas clases de errores.
- ***Usan revisiones técnicas efectivas como filtro previo a las pruebas.*** Las revisiones técnicas reducen el esfuerzo de pruebas para producir software de alta calidad.
- ***Hacen revisiones técnicas p/valorar estrategia de prueba y casos de prueba***. Descubrir inconsistencias, omisiones y errores, ahorra tiempo y mejora la calidad.
- ***Desarrollan un enfoque de mejora continuo para el proceso de prueba.*** La estrategia de pruebas debe medirse.

Herramientas para Pruebas de Software

Las **herramientas** son un soporte necesario para la automatización de las pruebas de software

- ✓ Elevada cantidad de casos de pruebas
- ✓ Repeticiones de las pruebas

- Herramientas para pruebas estáticas, ayudan a encontrar defectos en etapas tempranas, en el código, en la funcionalidad, en los modelos de software.
- Herramientas para planificación y gestión, documentar y evaluar los casos de pruebas, brindan una visión general de los casos de prueba del proyecto y la gestión de resultados.
- Herramientas para pruebas de automatización, se crean scripts en diferentes lenguajes y permiten automatizar las pruebas funcionales.
- Herramientas para seguridad y monitorización, permiten probar los protocolos de seguridad y monitorear.

Testing in the Small (Prueba de Unidades)

El diseño de los casos de pruebas debe considerar la mayor probabilidad de encontrar el mayor número de errores con la mínima cantidad de esfuerzo y tiempo posible.

- ✓ Aunque no será posible la prueba exhaustiva, la estrategia de prueba debe contemplar y definir estratégicamente los casos de prueba, de manera que sean eficientes, efectivos y económicos.

- ✓ La **pruebas de unidades** consisten en probar módulos individuales

Testing in the Small (Prueba de Unidades)

Técnicas de Prueba de Unidad

- **Prueba de Caja Blanca o Estructural:** permiten verificar la estructura interna de un programa, de donde derivan los casos de prueba.
- **Prueba de Caja Negra o Funcional:** los casos de prueba derivan de la especificación. Interesa la funcionalidad.

Testing in the Small (Prueba de Unidades)

- **Prueba de Caja Blanca o Estructural**

  - Criterio de Cobertura de Sentencia

  - Criterio de Cobertura de Arco

  - Criterio de Cobertura de Condición

  - Criterio de Cobertura de Camino

Caja Blanca - Criterio de Cobertura de Sentencia

Ejecuta el programa y se asegura de que todas sus sentencias son ejecutadas al menos un vez.

Asegura que se puede descubrir si las partes del programa contienen algún error, ejecutando cada sentencia al menos una vez.

Definición

Seleccionar un conjunto de casos de prueba T tal que, ejecutando el programa P por cada d en T, cada sentencia elemental de P es ejecutada al menos una vez.

Debilidades

- Ejecutar cada sentencia y ver como se comporta no garantiza que el programa no contenga errores.
- No es clara la definición de sentencia, cual es la sentencia más elemental.

Caja Blanca - Criterio de Cobertura de Sentencia

Ejemplo

```
read (x)
read (y)
If x > 0 then
      write ('x es positivo')
      x:=-2
else
      write ('x es negativo')
endif
If y > 0 then
      write ('y es positivo')
else
      write ('y es negativo')
endif
```

Definimos un conjunto mínimo de Casos de Prueba que cumpla con el criterio (todas las sentencias son ejecutadas al menos una vez!)

Ejemplo 1

T1={<x=-5, y=6>, <x=5, y=-6>}

Ejemplo 2

T2={<x=5, y=6>, <x=-5, y=-6>}

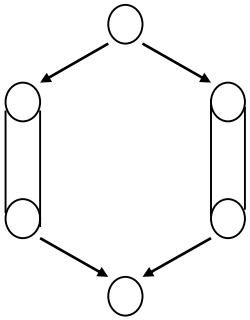
T1 y T2 son dos posibles conjuntos mínimos de CP ninguno encuentra error!

Caja Blanca - Criterio de Cobertura de Arco

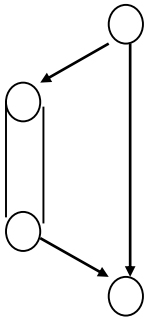
Describe el programa con un grafo del flujo de control, y asegura que se recorren todos los arcos en dicho grafo.

Notación para la construcción del **grafo del flujo de control**

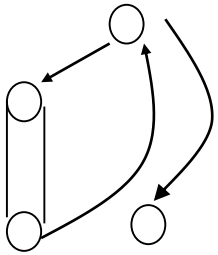
I/O, call, asignación



If-then-else



If-then



while loop

sentencias secuenciales

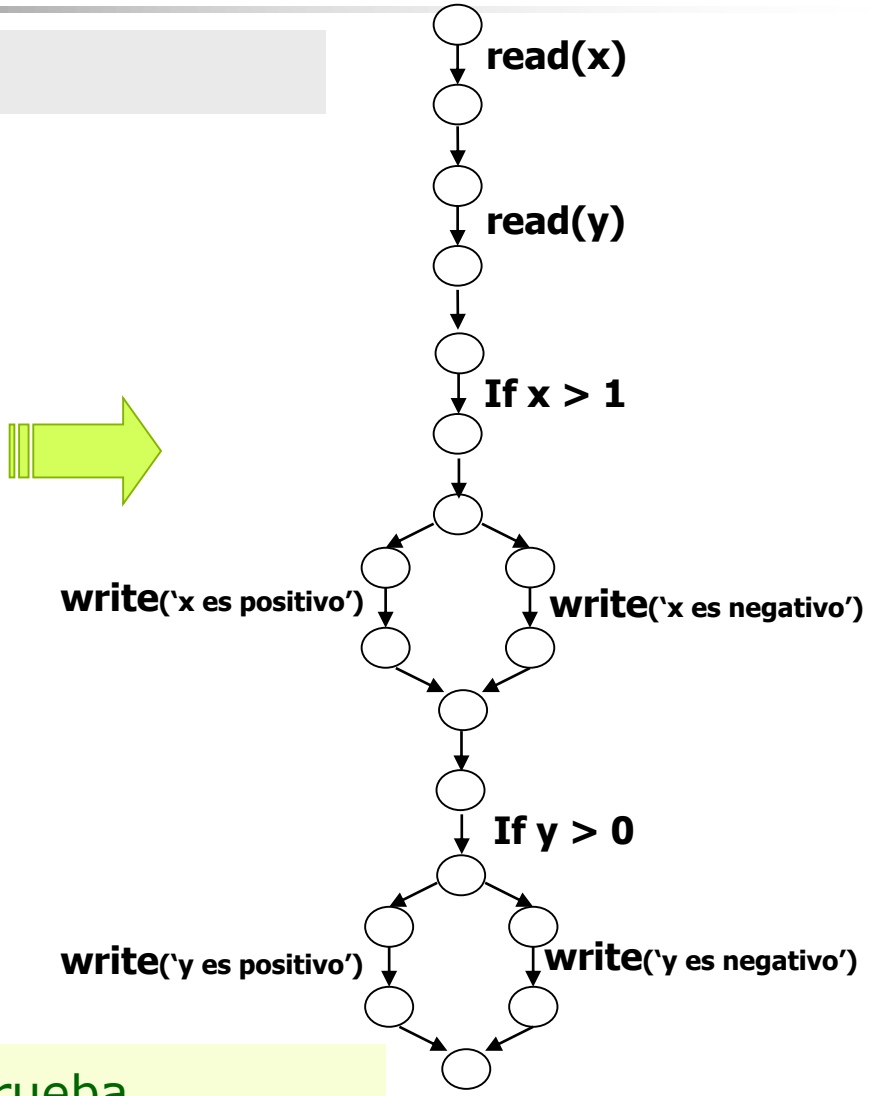
Definición - Criterio de Cobertura de Arco

Se selecciona un conjunto de prueba T tal que, ejecutando el programa P por cada d en T, cada arco del flujo de control de P es atravesado al menos una vez.

Caja Blanca - Criterio de Cobertura de Arco

Ejemplo 1

```
read (x)
read (y)
If x >= 1 then
        write ('x es positivo')
else
        write ('x es negativo')
endif
If y > 0 then
        write ('y es positivo')
else
        write ('y es negativo')
endif
```



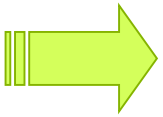
Conjunto mínimo de Casos de Prueba

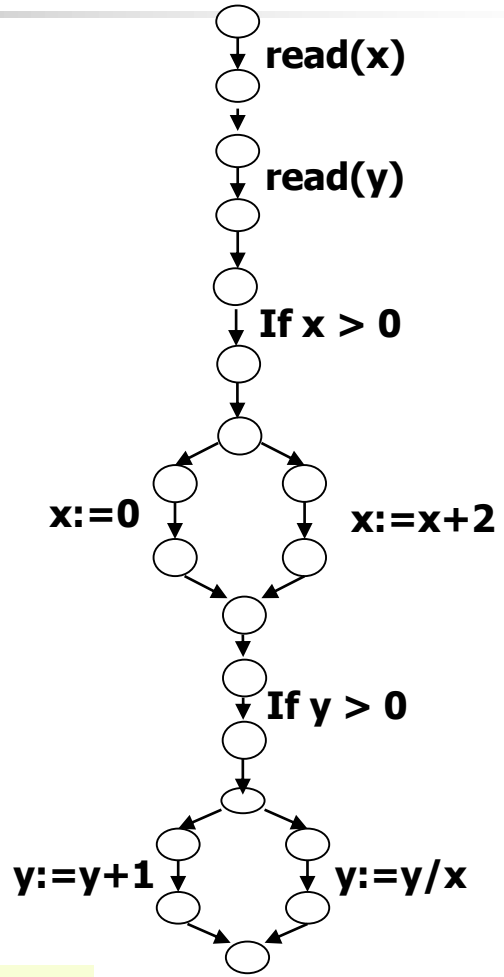
T = {<x = 5, y = 10>, <x = -5, y = -10>}

Caja Blanca - Criterio de Cobertura de Arco

Ejemplo 2

```
read (x)
read (y)
If x > 0 then
        x:=0
else
        x:=x+2
endif
If y > 0 then
        y:=y+1
else
        y:=y/x
endif
```





T1 = {<x = 5, y = 10>, <x = -5, y = -10>}

T1 no encuentra error

T2 = {<x = 5, y = -10>, <x = -5, y = 10>}

T2 encuentra el error!

Caja Blanca - Criterio de Cobertura de Condición

Divide las expresiones booleanas complejas en sus componentes e intenta cubrir todos los valores posibles de cada uno de ellos.

Definición

Se selecciona un conjunto de prueba T tal que, ejecutando el programa P por cada d en T, cada arco del flujo de control de P es atravesado al menos una vez y todos los posibles valores de los constituyentes de las condiciones compuestas son probadas al menos una vez.

Ejemplo: x y z

< x = V , z = V >

< x = F , z = ? >

< x = V , z = F >

Caja Blanca - Criterio de Cobertura de Camino

Asegura que todos los caminos del grafo de flujo de control del programa son atravesados al menos una vez.

Definición

Se selecciona un conjunto de prueba T tal que, ejecutando el programa P por cada d en T, todos los caminos principales o independientes desde el nodo inicial al final del flujo de control de P son atravesados.

Necesitamos saber cómo se definen los caminos principales o caminos independientes (Complejidad Ciclomática)

Complejidad Ciclomática

- Es una métrica que proporciona una medida cuantitativa de la complejidad lógica de un programa.
- Define el número de caminos independientes de un programa.
- Define un **límite superior** de Casos de Prueba a considerar con **Cobertura de Arcos** y un **límite inferior** de los posibles caminos independientes a considerar con **Cobertura de Caminos.**

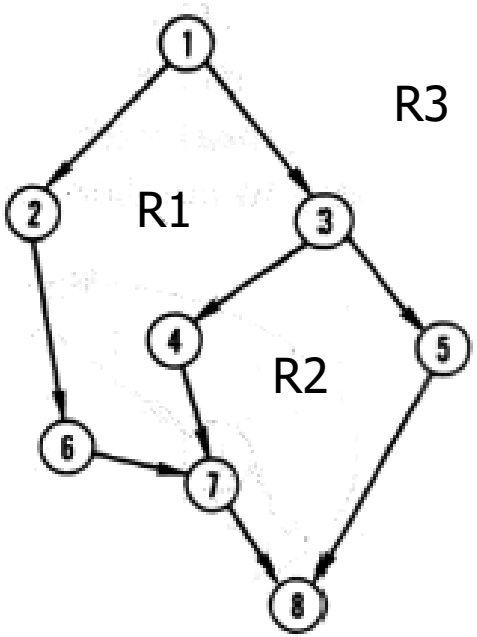
Formas posibles de calcular la **Complejidad Ciclomática**:

1. Contando el número de regiones del grafo de flujo.
2. A través de aristas y nodos
3. Contando los nodos predicado

Caja Blanca - Criterio de Cobertura de Camino

Formas posibles de calcular la **Complejidad Ciclomática**:

1. Contando el número de regiones del grafo de flujo de control.



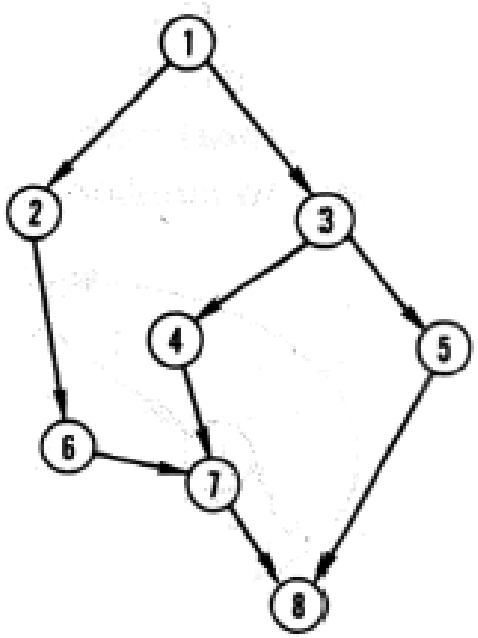
V (G) = 3

Caja Blanca - Criterio de Cobertura de Camino

2. Aplicando la siguiente fórmula, en base a las aristas y nodos:

$$V(G) = A - N + 2$$

donde A es el número de aristas del grafo de flujo y N es el número de nodos del mismo.



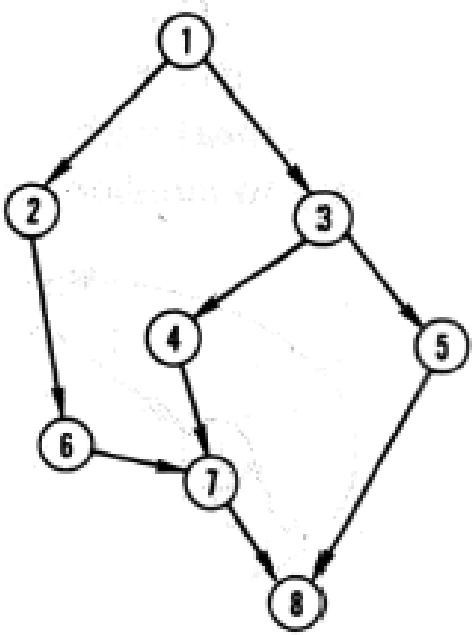
$$V(G) = A - N + 2$$
$$V(G) = 9 - 8 + 2$$
$$V(G) = 3$$

Caja Blanca - Criterio de Cobertura de Camino

3. Contando los nodos predicado:

$$V(G)=P+1$$

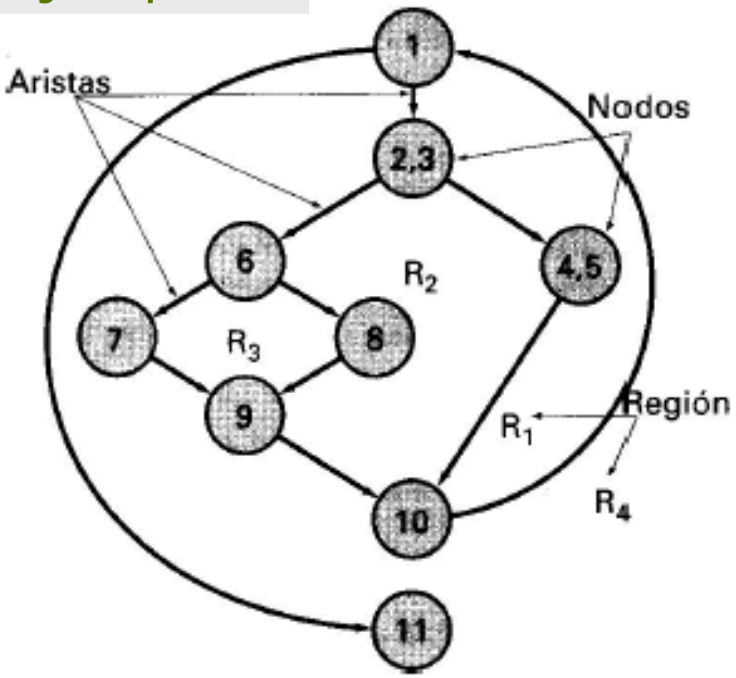
donde P es el número de nodos predicado contenidos en el grafo de flujo G.



$$V(G) = P + 1$$
$$V(G) = 2 + 1$$
$$V(G) = 3$$

Caja Blanca - Criterio de Cobertura de Camino

Ejemplo



Un camino independiente es cualquier camino del programa que introduce al menos un nuevo conjunto de sentencias o una nueva arista.

Grafo de Flujo

Nodos (N): una o más sentencias.

Aristas (A): flujos de control.

Regiones: áreas delimitadas por aristas y nodos.

$$V(G) = A - N + 2$$

$$11 - 9 + 2 = 4$$

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

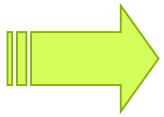
Ej. dos sentencias if-then-else secuenciales genera un conjunto mínimo de 4 CP.

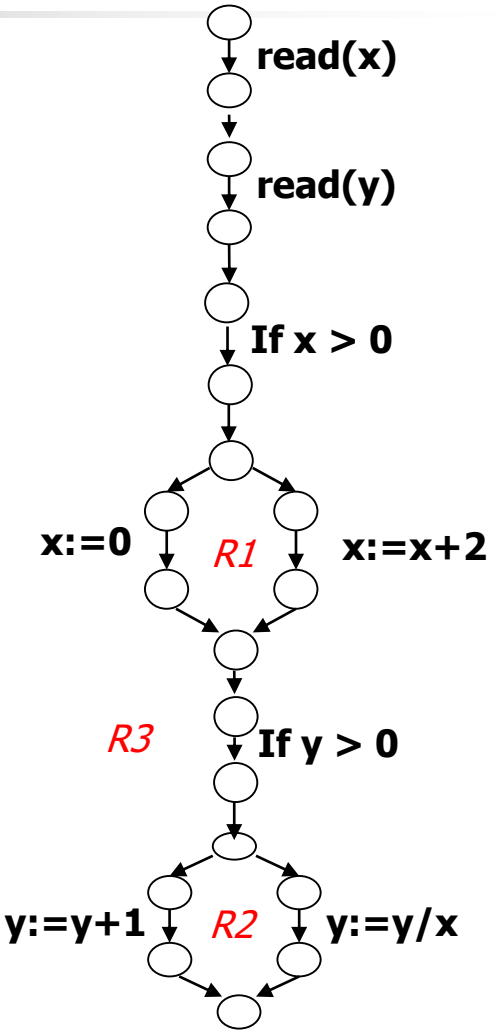
Ej. Una sentencia while genera 3 CP: no entra al loop, un nro. máximo de veces, y un nro. promedio de veces.

Caja Blanca - Criterio de Cobertura de Camino

Ejemplo

```
read (x)
read (y)
If x > 0 then
        x:=0
else
        x:=x+2
endif
If y > 0 then
        y:=y+1
else
        y:=y/x
endif
```





Calculamos la complejidad ciclomática = 3

T = {<x = 2, y = 2>, <x = -2, y = -2>, <x = 2, y = -2>, <x = -2, y = 2>}

T encuentra error!

Testing in the Small (Prueba de Unidades)

- **Prueba de Caja Negra o Funcional**

  - Análisis del Valor Límite

  - Prueba de Clases de Equivalencia

  - Pruebas basadas en Tablas de Decisión

Caja Negra - Análisis del Valor Límite

Define un rango de valores posibles para cada variable
y focaliza la prueba
en los valores de los extremos del rango.

- La técnica Análisis de Valor Límite es adecuada para usar cuando las variables son independientes, y refieren a cantidades físicas. Ej: temperatura, velocidad, presión, ángulo, ……

- **Identifica los Casos de Prueba** focalizando en los límites del espacio de entrada; asegura que los errores tienden a ocurrir cerca de los valores extremos de una variable de entrada.

Caja Negra - Análisis del Valor Límite

Dada una función F (programa) y las entradas **X1**, **X2**, …., **Xn**

se definen los valores límites (rango) para cada una de las variables:

$$\min X1 \leq \mathbf{X1} \leq \max X1$$

$$\min X2 \leq \mathbf{X2} \leq \max X2$$

……..

$$\min Xn \leq \mathbf{Xn} \leq \max Xn$$

- Análisis de Valor Límite **define los casos de prueba** con el min, min+1, nom, max, max-1 para cada variable.

- Los **casos de prueba** se obtienen fijando cada variable en su valor nominal, y luego moviendo por todos los valores (min, min+1, nom, max, max-1) de cada una de las demás variables.

Caja Negra - Análisis del Valor Límite

Ejemplo: dados los lados de un triángulo, definir el tipo de triángulo.

Definimos un rango entre 1 y 200 para las variables **a, b y c**

min=1 min+1=2 nom=100 max=200 max-1=199

(3*5=15 casos de prueba)

Caso	a	b	c	Salida Esperada
1	100	100	1	Isósceles
2	100	100	2	Isósceles
3	100	100	100	Equilátero
4	100	100	199	Isósceles
5	100	100	200	No es un Triáng.
6	100	1	100	Isósceles
7	100	2	100	Isósceles
8	100	100	100	Equilátero
9	100	199	100	Isósceles
10	100	200	100	No es un Triáng.
11	1	100	100	Isósceles
12	2	100	100	Isósceles
13	100	100	100	Equilátero
14	199	100	100	Isósceles
15	200	100	100	No es un Triáng.

Caja Negra - Análisis del Valor Límite

- **Prueba de Robustez**: es una extensión de la técnica de Análisis del Valor Límite. Agrega los valores min-1 y el máx+1 para cada variable. Observa el comportamiento del componente o sistema, cuando ocurre un valor que excede el rango definido.

Ejemplo: Triángulo

Para las variables **a**, **b** y **c**, con rango entre 1 y 200

min-1= 0 min=1 min+1=2

nom=100

max=200 max-1=199 **max+1=201**

Caso	a	b	c	Salida Esperada
1	100	100	0	No es un triángulo
2	100	100	1	Isósceles
3	100	100	2	Isósceles
4	100	100	100	Equilátero
5	100	100	200	No es un Triáng.
6	100	100	199	Isósceles
7	100	100	201	No es un triángulo
…	…	…	…	…

Caja Negra - Análisis del Valor Límite

- **Prueba del Peor Caso:** generalización de la técnica de análisis de valor límite. Los casos de prueba se definen realizando el producto cartesiano con los x valores posibles para cada una de las n variables (x^n). Esta técnica requiere de mucho más esfuerzo, es más costosa.

Ejemplo: Triángulo

Variables **a**, **b** y **c**, rango entre 1 y 200. Cada variable asume 5 valores: min=1 min+1=2 nom=100 max-1=199 max=200

Los CP son 5^3 = 125.

Caso	a	b	c	Salida Esperada
1	100	100	1	Isósceles
2	100	100	2	Isósceles
3	100	100	100	Equilátero
4	100	100	199	Isósceles
5	100	100	200	No es un Triáng.
….	….	….	….	….
….	….	….	….	….
….	….	….	….	….
125	200	200	200	Equilátero

Caja Negra - Prueba de Clases de Equivalencia

Propone la partición de un conjunto, en una colección de subconjuntos mutuamente disjuntos o ajenos

- Se definen las **Clases de Equivalencia** como las particiones en un conjunto.
- La disyunción entre subconjuntos o clases garantiza la no redundancia de elementos.
- Los subconjuntos o clases son determinados por una relación de equivalencia.
- La unión de clases forma el conjunto y asegura la completitud.
- **Métodos de Prueba con Clases de Equivalencia: débil y fuerte**

Caja Negra - Prueba de Clases de Equivalencia

Prueba con Clases de Equivalencia Débil

Un valor para cada clase de equivalencia define un caso de prueba.

Supongamos:

$A = A1 \cup A2 \cup A3$

$B = B1 \cup B2 \cup B3 \cup B4$

$C = C1 \cup C2$

caso de prueba (CP)	a	b	c
1	a1	b1	c1
2	a2	b2	c2
3	a3	b3	c1
4	a1	b4	c2

Caja Negra - Prueba de Clases de Equivalencia

Prueba con Clase de Equivalencia Fuerte

Define el producto cartesiano entre las clases de equivalencia definidas. Ej:

A x B x C = 3 x 4 x 2 = 24 casos de prueba.

Se obtiene un caso de prueba de cada posible combinación de entradas.

CP	a	b	c
1	a1	b1	c1
2	a1	b1	c2
3	a1	b2	c1
4	a1	b2	c2
5	a1	b3	c1
6	a1	b3	c2
.....	.....	.....	.....
.....	.....	.....	.....
22	a3	b3	c2
23	a3	b4	c1
24	a3	b4	c2

Caja Negra - Pruebas basadas en Tablas de Decisión

Se utilizan para representar y analizar la complejidad de las relaciones lógicas. Describen situaciones en las cuales un número de combinaciones de acciones son realizadas bajo la variación de un conjunto de condiciones.

- La propiedad de completitud de las tablas de decisión garantiza una prueba completa.
- Las tablas de decisión con todas entradas binarias se denominan Tablas de Decisión de Entrada Limitada.
- Si las condiciones permiten tener valores distintos, las tablas resultantes se denominan Tablas de Decisión de Entrada Extendida.
- No hay un orden particular para las condiciones, ni para la ocurrencia de las acciones seleccionadas.

Caja Negra - Pruebas basadas en Tablas de Decisión

Para identificar los casos de prueba con Tablas de Decisión se identifican condiciones como entrada y acciones como salidas.

Las reglas representan los **casos de prueba.**

Ejemplo: El problema del Triángulo

	reglas								
c1: a,b,c son parte de un triángulo?	N					Y			
c2: a = b ?	-			Y			N		
c3: a = c ?	-	Y			N			Y			N
c4: b = c ?	-	Y	N	Y	N	Y	N	Y	N
a1: no es un triángulo	X								
a2: Escaleno									X
a3: Isósceles					X		X	X	
a4: Equilátero		X							
a5: Imposible			X	X		X			

condiciones

acciones

Bibliografía

- Software Testing. A Craftsman´s Approach. Paul Jorgensen. Ed 4, 2014

  Capítulo 5: Boundary Value Testing

  Capítulo 6: Equivalence Class Testing

  Capítulo 7: Decision Table Based Testing

- Software Engineering. A Practitioner´s Approach, Eighth Edition. Roger S. Pressman. 2015.

  Capítulo 22: Software Testing Strategies

  Capítulo del 23 al 26: Métodos y técnicas de testing que implementan las estrategias

- Fundamentals of Software Engineering. C. Ghezzi, M. Jazayeri, D. Mandrioli. 1991.

  Capítulo 6: Verification. 6.1 6.2. 6.3.