Módulo 8: Testing Estructural

Diseño y Desarrollo de Software (*1er. Cuat. 2019*)

Profesora titular de la cátedra:
Marcela Capobianco

Profesores interinos:
Sebastian Gottifredi y
Gerardo I. Simari

Licenciatura en Ciencias de la Computación – UNS

Licencia

Pruebas de SW

- **Elemento crítico** para la garantía de **calidad** del software.
- Para **evitar errores**, debe hacerse una definición de **pruebas minuciosas y bien planificadas**.
- Se estima que estas actividades **llevan un 40%** del esfuerzo total del proyecto.
- Este porcentaje aumenta en sistemas críticos.
- ***Objetivo*:** Diseñar una serie de **casos de prueba** con **alta probabilidad** de **encontrar errores**.

Conceptos básicos IEEE 829-83

- ***Error*** (“error” en inglés): Las personas cometen errores cuando codifican.
- ***Defecto*** (“fault” o “defect” en inglés): Es el resultado de un error. Es la representación de un error (código, narrativas, etc.), se los conoce como *bugs*.
- ***Falla*** (“failure” en inglés): Ocurre cuando se ejecuta un defecto. El término *bug* también suele usarse de esta manera.

Conceptos básicos IEEE 829-83

- ***Incidente*** (“incident” en inglés): Síntoma asociado con una falla.
- ***Testeo***: Acto de probar el software con casos de test.
- ***Caso de test***:
  - Elementos que poseen una identidad y están asociados con un *comportamiento específico* del programa.
  - Tienen también una lista de entradas y salidas.

Diferentes tipos de defectos

Table 1.2 Logic Faults

Missing case(s)
Duplicate case(s)
Extreme condition neglected
Misinterpretation
Missing condition
Extraneous condition(s)
Test of wrong variable
Incorrect loop iteration
Wrong operator (e.g., < instead of ≤)

Diferentes tipos de defectos

Table 1.3 Computation Faults

Incorrect algorithm
Missing computation
Incorrect operand
Incorrect operation
Parenthesis error
Insufficient precision (round-off, truncation)
Wrong built-in function

Diferentes tipos de defectos

Table 1.4 Interface Faults

Incorrect interrupt handling
I/O timing
Call to wrong procedure
Call to nonexistent procedure
Parameter mismatch (type, number)
Incompatible types
Superfluous inclusion

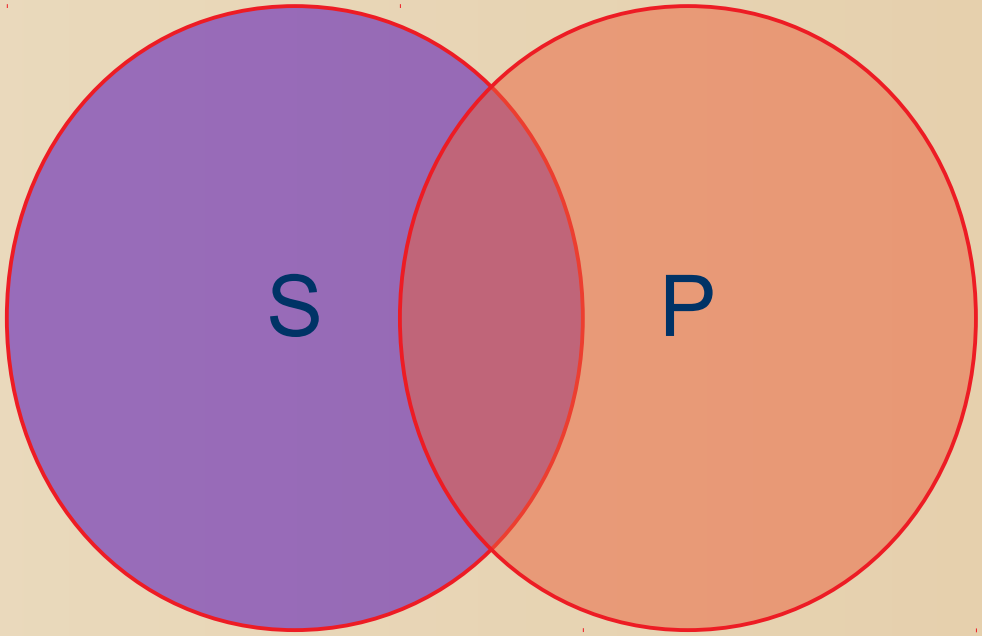
Caso de test

- ID del caso de test
- Propósito
- Pre-condiciones
- Entradas
- Salidas esperadas
- Post-condiciones
- Historia de ejecución:

Fecha *Resultado* *Versión* *Ejecutado por*

Comportamiento: *Especificado* vs. *Implementado*

Universo de comportamientos



S = Especificado (significado *pretentido*)
P = Programado (significado *efectivo*)

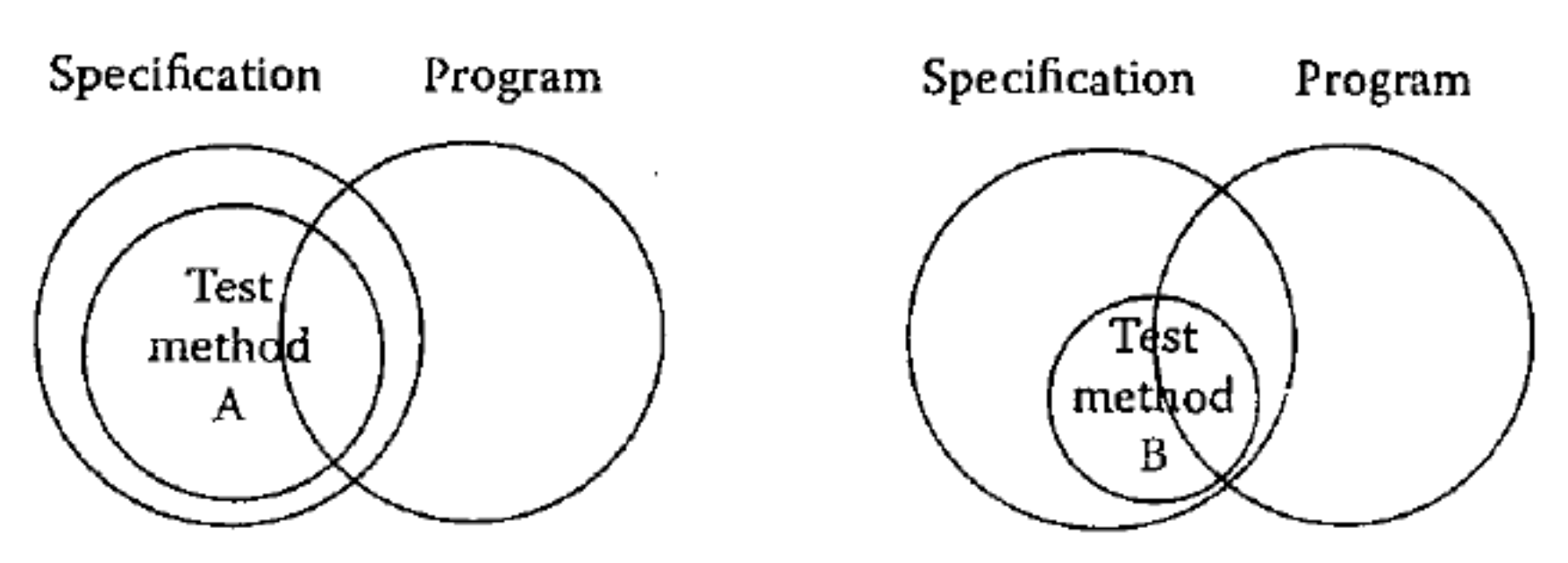
Diferentes regiones

- La **intersección** entre las regiones S y P es la parte "***correcta***": comportamiento programado tal cual fue especificado.
- "***Correctitud***" sólo tiene sentido con respecto a la relación entre especificación e implementación.
- Las otras dos regiones corresponden a comportamientos:
  - implementados pero no especificados, y
  - especificados pero no implementados.

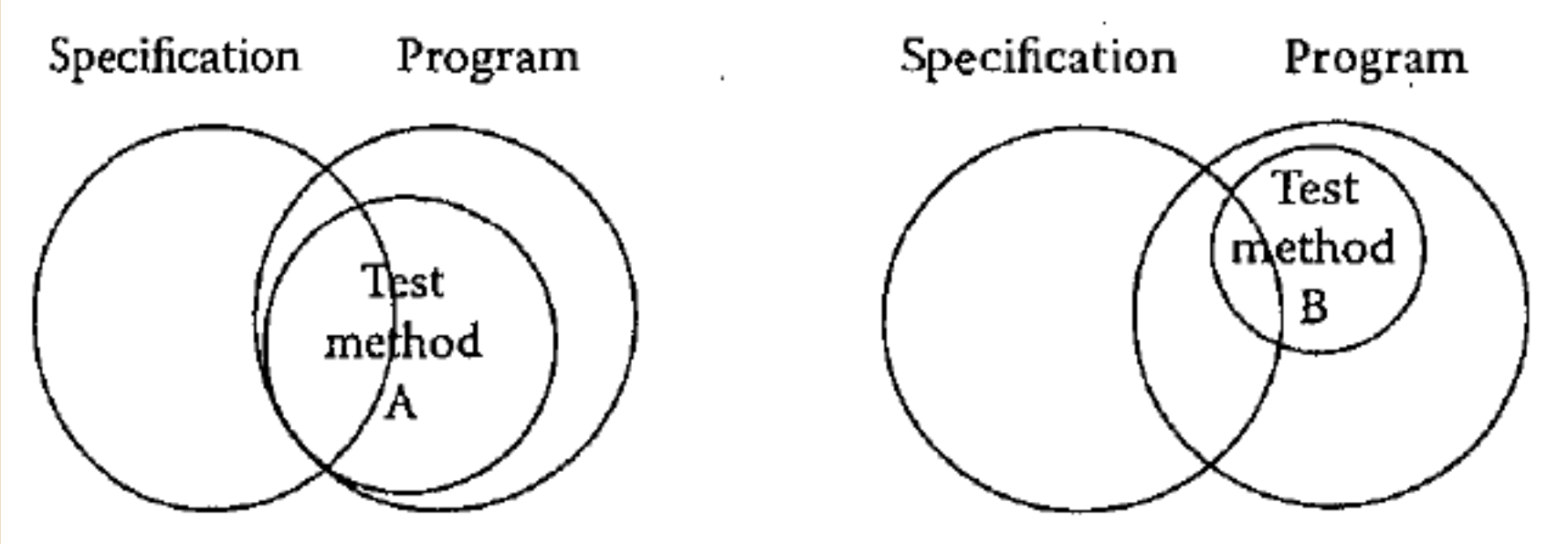
Testing

- Técnicas que intentan establecer una **guía sistemática** para **desarrollar casos de test** que
  - comprueben la **lógica de los componentes**
  - verifiquen la **correspondencia** especificada entre las **entradas** y **salidas** del programa.
- El SW se prueba desde dos perspectivas:
  - Pruebas de ***caja blanca*** (lógica interna)
  - Pruebas de ***caja negra*** (requerimientos del software)

Testing de caja negra



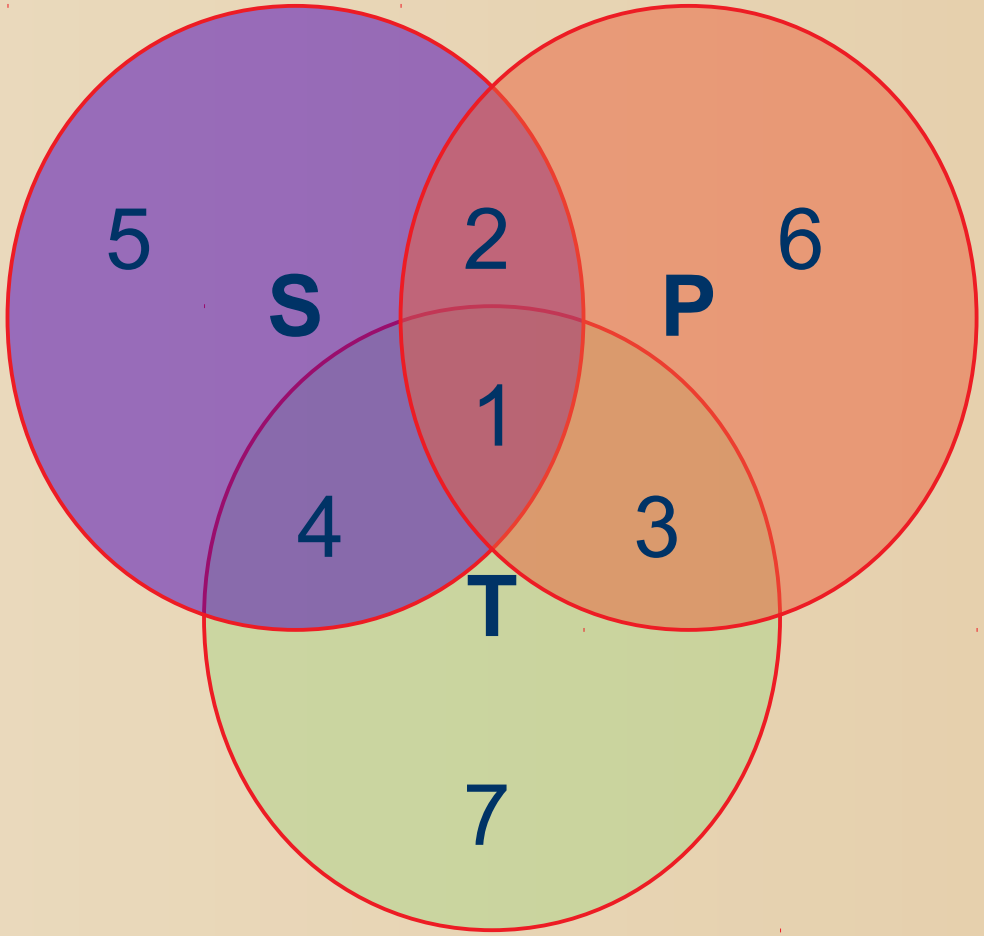
Testing de caja blanca



Diferentes regiones *bis*

Qué pasa si agregamos un nuevo conjunto a la figura, correspondiente a los comportamientos *testeados*?

Comportamiento:
Especificado vs. *Implementado* vs. *Testeado*



Diferentes regiones *bis*

Surgen varias regiones más con comportamientos:

- Especificados pero no testeados (2 y 5)
- Especificados sí testeados (1 y 4)
- Testeados pero no especificados (3 y 7)
- Implementados pero no testeados (2 y 6)
- Implementados sí testeados (1 y 3)
- Testeados no implementados (4 y 7)

Diferentes regiones *bis*

- En general, se intenta que la **región 1 sea lo más grande posible**.
- Todas las regiones son importantes.
- En particular, si hay casos de test que corresponden a comportamientos no especificados:
  - El caso de test tal vez es innecesario
  - La especificación puede ser deficiente
  - El tester puede asegurarse de que no sucede nada inaceptable.

Fundamentos de las pruebas

- Una prueba es un paso “destructivo”: tiene **es útil** si se **encuentra un error** no descubierto.
- **Demuestra** además cómo **funciona el software** de acuerdo con los requerimientos.
- *Atención*: ¡Las pruebas **no** pueden **asegurar** la **ausencia de defectos**!

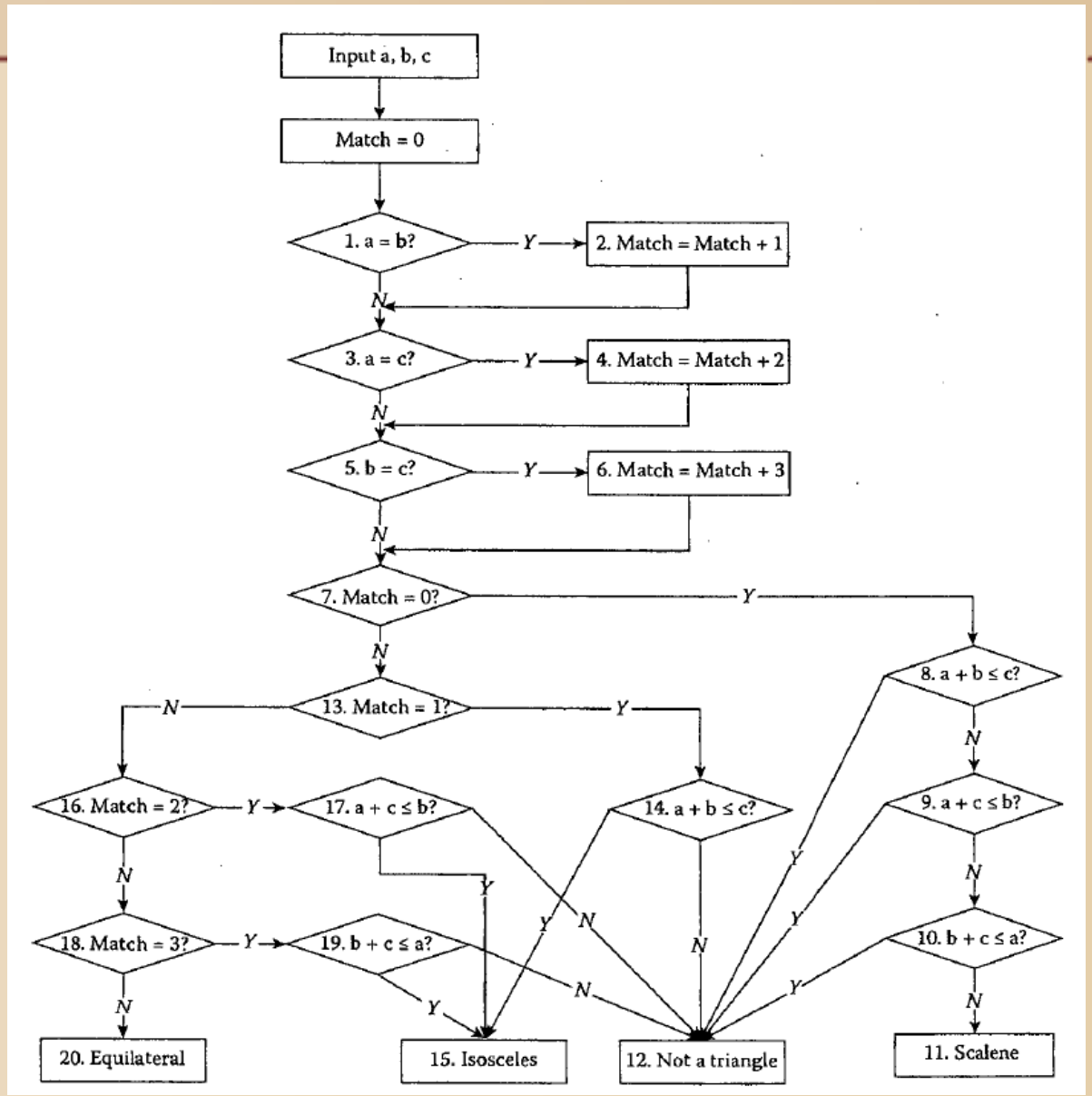
Principios de las pruebas

- Las **pruebas** deben poder vincularse con los **requerimientos del cliente**.
  - Es buena práctica **planificarlas** antes de que empiecen.
- El 80% de los errores corresponde a un 20% de los módulos.
- Deben ir de lo pequeño hacia lo grande.
- **No** son **posibles/viables** las pruebas **exhaustivas**.

El Problema del Triángulo

- Dados tres números enteros *a*, *b* y *c* interpretados como lados de un triángulo, determinar si éstos corresponden a un triángulo de tipo:
  - equilátero,
  - isósceles,
  - escaleno, o bien a
  - valores *imposibles* para un triángulo (por ejemplo, lados de longitud 0, o puntos pertenecientes a una recta).
- Es un problema muy simple, pero con lógica de implementación lo suficientemente compleja como para ilustrar el problema del testing.

El Problema del Triángulo



Función *NextDate* (Próxima fecha)

- La complejidad del problema del triángulo se debe a las relaciones entre las *entradas* y posibles *salidas*.
- Para ilustrar otro tipo de complejidad, que surge de relaciones *entre entradas*, usaremos *NextDate*:
  - Tres entradas: día, mes, año, con rangos adecuados.
  - Determina o bien la fecha siguiente, o señala que la entrada *no es válida* (31 de junio de cualquier año).
- Un caso interesante es además la necesidad de detectar *años bisiestos*.

Diferencias entre testing y SQA

- SQA: *Software Quality Assurance*
- SQA apunta a mejorar el producto a través de una mejora en el proceso.
- Testing se centra más en el producto: Se pretende descubrir fallas en él.

Pruebas de Caja Blanca

Pruebas de caja blanca

- Las pruebas de **caja blanca** realizan un **examen** minucioso de los **detalles procedurales**.
- Se testean los **caminos lógicos** del software.
- *Problema*: Es **imposible** realizar un **testeo exhaustivo**.
- *Importancia*: Permite testear los caminos lógicos más importantes, como así también las estructuras de datos.

Pruebas de caja blanca

- Se usa la **estructura de control** del diseño procedural para **derivar** los **casos de prueba**. **Objetivos:**
  - Garantizar que al menos todos los "**caminos independientes**" dentro de un módulo se han seguido.
  - Probar todas las **decisiones lógicas** en sus caminos *verdadero* y *falso.*
    - Ejecutar todos los **bucles** en sus **límites**.

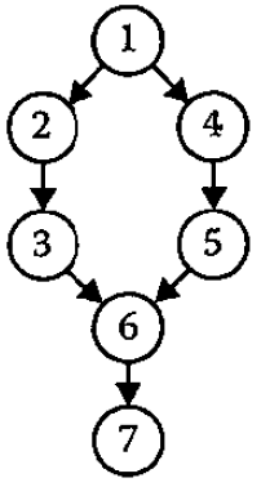
Grafo de programa

- Dado un programa escrito en un lenguaje imperativo, su *grafo de programa* es un grafo dirigido en el cual los nodos son fragmentos de sentencias y los arcos representan flujo de control.
- Si *i* y *j* son nodos del grafo, existe un arco de *i* a *j* si y sólo si el fragmento que corresponde al nodo *j* puede ser ejecutado después del correspondiente al nodo *i*.

Grafo de programa

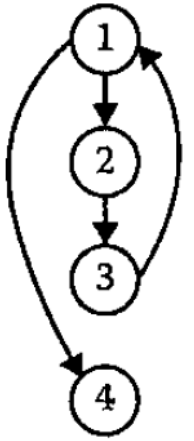
If–Then–Else

```
1  If <condition>
2     Then
3        <then statements>
4     Else
5        <else statements>
6  End If
7  <next statement>
```



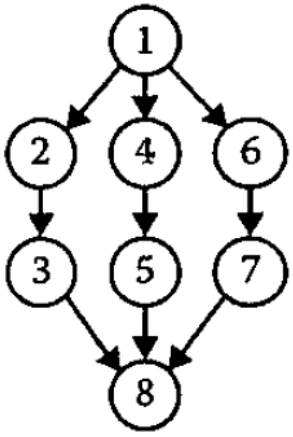
Pretest loop

```
1  While <condition>
2     <repeated body>
3  End While
4  <next statement>
```



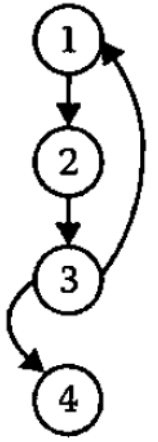
Case/Switch

```
1  Case n of 3
2     n=1:
3        <case 1  statements>
4     n=2:
5        <case 2  statements>
6     n=3:
7        <case 3  statements>
8  End Case
```



Posttest loop

```
1  Do
2     <repeated body>
3  Until <condition>
4  <next statement>
```

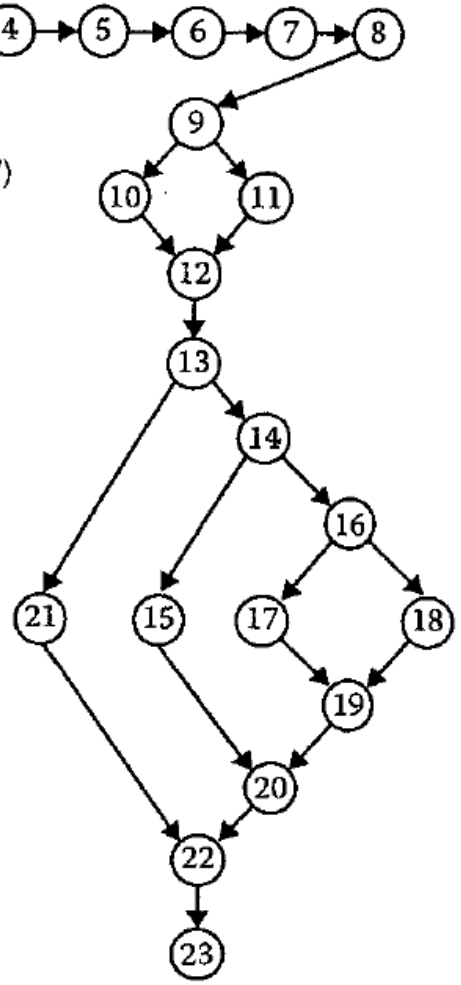


Grafo para el *Problema del Triángulo*

```
1  Program triangle2
2  Dim a,b,c As Integer
3  Dim IsATrinagle As Boolean
4  Output("Enter 3 integers which are sides of a triangle")
5  Input(a,b,c)
6  Output("Side A is", a)
7  Output("Side B is", b)
8  Output("Side C is", c)
9  If (a < b + c) AND (b < a + c) AND (c < a + b)
10   Then IsATriangle = True
11   Else IsATriangle = False
12 EndIf
13 If IsATriangle
14   Then  If (a = b) AND (b = c)
15           Then Output ("Equilateral")
16           Else  If (a≠b) AND (a≠c) AND (b≠c)
17                    Then Output ("Scalene")
18                    Else Output ("Isosceles")
19                EndIf
20           EndIf
21   Else  Output("Nota a Triangle")
22 EndIf
23 End triangle2
```

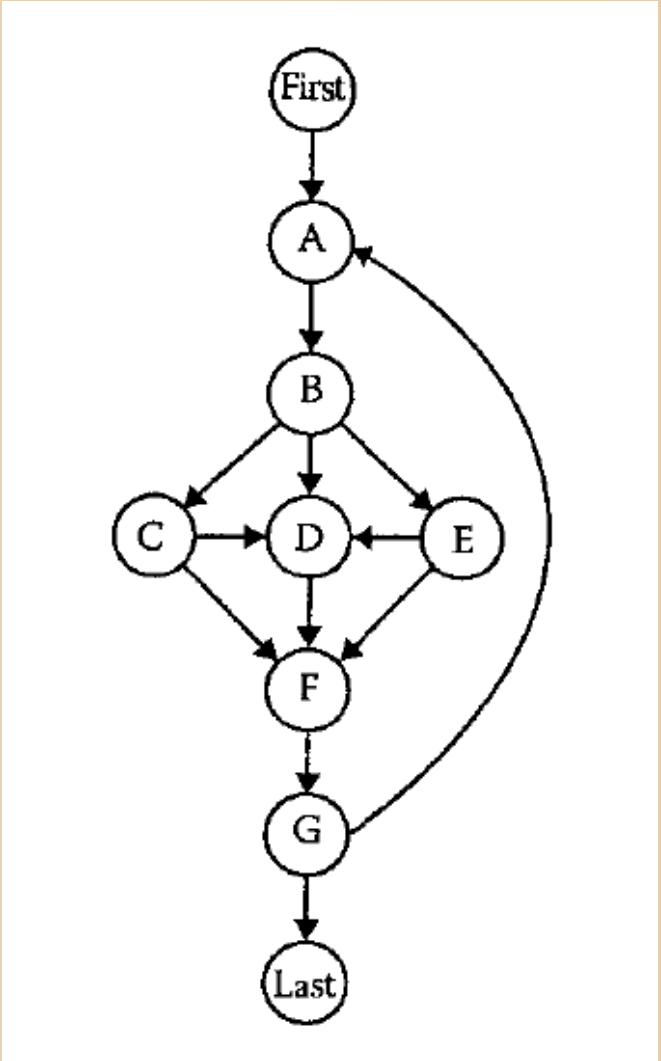
Grafo para el *Problema del Triángulo*

```
1  Program triangle2
2  Dim a,b,c As Integer
3  Dim IsATrinagle As Boolean
4  Output("Enter 3 integers which are sides of a triangle")
5  Input(a,b,c)
6  Output("Side A is", a)
7  Output("Side B is", b)
8  Output("Side C is", c)
9  If (a < b + c) AND (b < a + c) AND (c < a + b)
10    Then IsATriangle = True
11    Else IsATriangle = False
12 EndIf
13 If IsATriangle
14    Then  If (a = b) AND (b = c)
15             Then Output ("Equilateral")
16             Else  If (a≠b) AND (a≠c) AND (b≠c)
17                      Then Output ("Scalene")
18                      Else Output ("Isosceles")
19                   EndIf
20             EndIf
21    Else  Output("Nota a Triangle")
22 EndIf
23 End triangle2
```



Billones de caminos

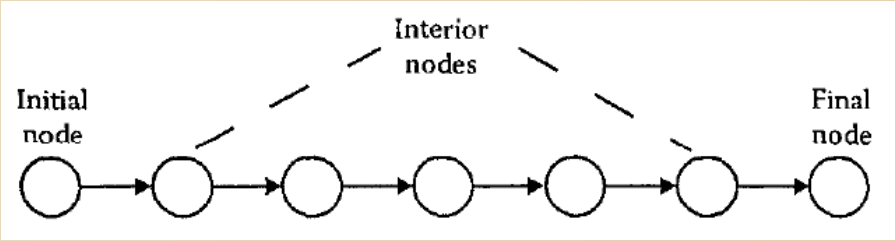
- Ya mencionamos que mirar todos los caminos es imposible.
- ¡Este simple grafo tiene 4.768.371.582.030 caminos!
- Este ejemplo en particular no surge de la programación estructurada.
- Esto no significa que no se pueda inspeccionar este espacio de manera inteligente.



DD-Paths

- *Decision to Decision Paths* [Miller, 1977]: Resumen las secuencias sin ramificaciones.
- Son una forma de llegar a un grafo condensado:

1) nodo solo con indeg = 0
2) nodo solo con outdeg = 0
3) nodo solo con indeg > 1 o outdeg > 1
4) nodo solo con indeg = 1 y outdeg = 1
5) cadena maximal con longitud mayor o igual a 1



DD-Paths para el *Problema del Triángulo*

```
1  Program triangle2
2  Dim a,b,c As Integer
3  Dim IsATrinagle As Boolean
4  Output("Enter 3 integers which are sides of a triangle")
5  Input(a,b,c)
6  Output("Side A is", a)
7  Output("Side B is", b)
8  Output("Side C is", c)
9  If (a < b + c) AND (b < a + c) AND (c < a + b)
10    Then IsATriangle = True
11    Else IsATriangle = False
12 EndIf
13 If IsATriangle
14   Then  If (a = b) AND (b = c)
15           Then Output ("Equilateral")
16           Else  If (a≠b) AND (a≠c) AND (b≠c)
17                    Then Output ("Scalene")
18                    Else Output ("Isosceles")
19                EndIf
20           EndIf
21   Else  Output("Nota a Triangle")
22 EndIf
23 End triangle2
```

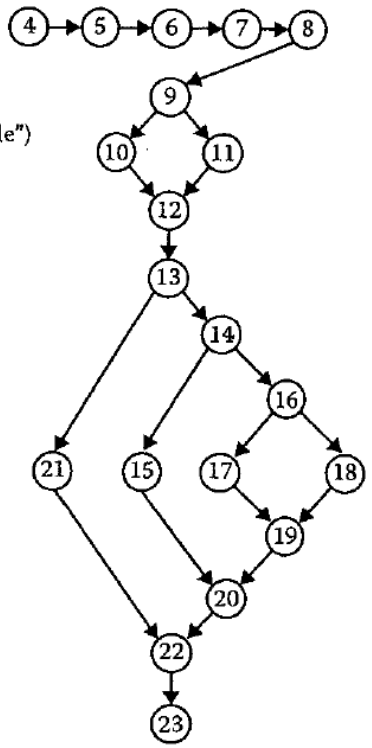
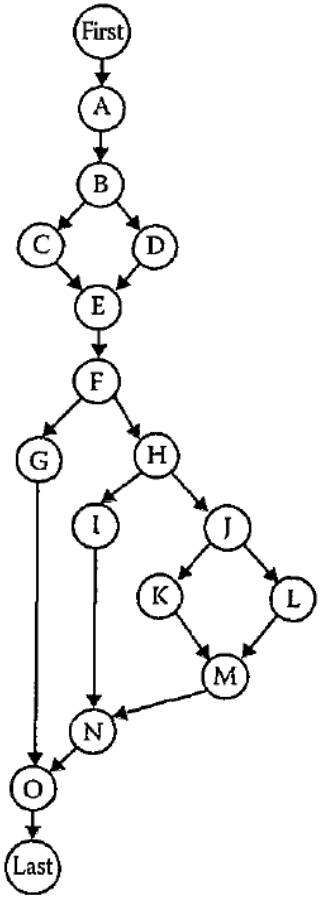


Figure 8.2 Nodes	DD-Path	Case of definition
4	First	1
5-8	A	5
9	B	3
10	C	4
11	D	4
12	E	3
13	F	3
14	H	3
15	I	4
16	J	3
17	K	4
18	L	4
19	M	3
20	N	3
21	G	4
22	O	3
23	Last	2



Cubrimientos

- Dado un **programa**, pueden existir varios grafos asociados, pero todos convergen a **un único conjunto de *DD-paths***.
- Existen buenas herramientas automáticas para computar los grafos y *DD-paths*.
- La razón de ser de los *DD-paths* es que permiten especificar en forma precisa el ***cubrimiento* de los casos de test**.
- Tener una visión del cubrimiento de las pruebas en un programa permite manejar mejor el proceso.

Algunos criterios de Cubrimiento

- G_{node}: Cada nodo del grafo del programa es cubierto por algún caso de test.

  Significa que todas las sentencias son ejecutadas.

- G_{edge}: Cada arco del grafo del programa es cubierto por algún caso de test.

  Significa que todas las decisiones posibles son ejercitadas.

- G_{path}: Cada camino del principio al fin *es cubierto por algún caso de test.*

Cubrimientos de Miller

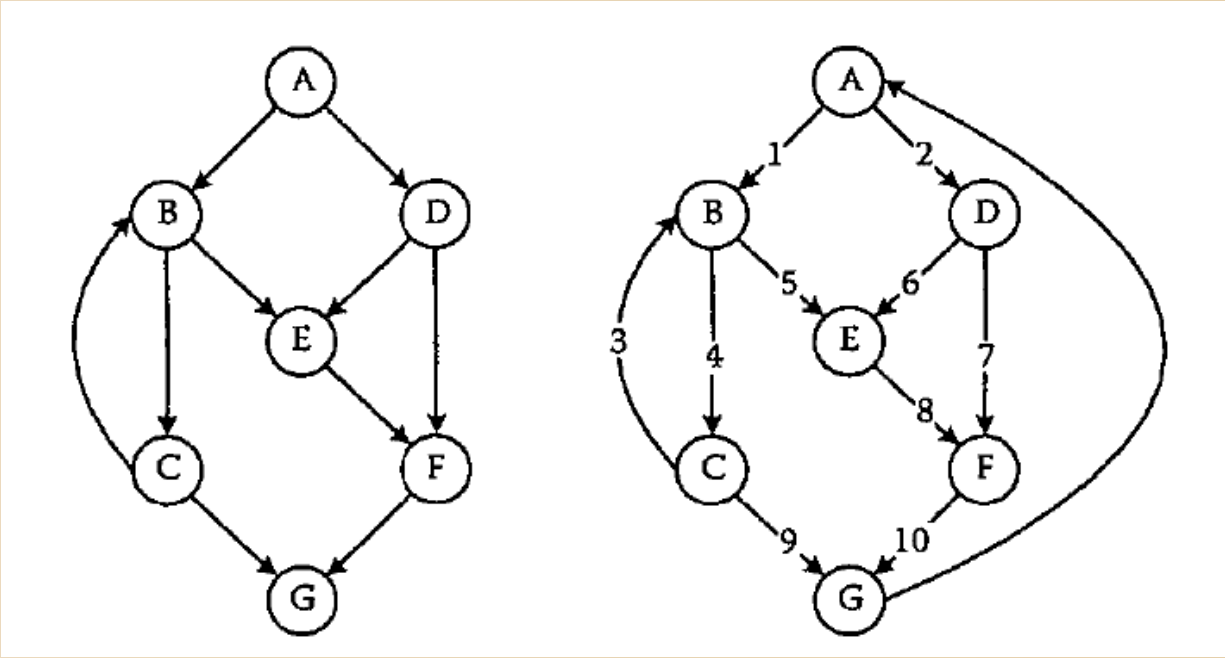
- C_0 = Cada statement del programa
- C_1 = Cada *DD-path*
- C_{1P} = Cada predicado con cada salida posible
- C_2 = C_1 + cobertura de bucles
- C_d = C_1 + caminos dependientes (*data flow testing*)
- C_{MCC} = Condiciones múltiples
- C_{ik} = Cada camino del programa que contiene hasta *k* repeticiones de un bucle (normalmente se toma *k* = 2)
- C_{stat} = Fracción “estadísticamente significativa” de los caminos posibles
- C_∞ = Todos los caminos de ejecución posibles

Testeo del camino base

- Una forma de obtener un cubrimiento C_{1P}
  - A partir de cubrir todos los nodos (G_{node}) y arcos (G_{edge})
- Buscamos que caracterizar cada **camino linealmente independiente** en el grafo
- Cada **camino** nos permitirá derivar un **caso de prueba**
- Esta idea fue propuesta y desarrollada por **McCabe** hace más de tres décadas.

Testeo del camino base

Consideremos el siguiente grafo de programa (note que no surge de programación estructurada):



Testeo del camino base

- **Complejidad ciclomática** de un grafo conexo G:

  $V(G) = e - n + 2$

  donde e es la cantidad de aristas y n la de nodos.
- Este **número** corresponde a la **cantidad de *caminos linealmente independientes*** en el grafo.
- Para el grafo anterior, tenemos:

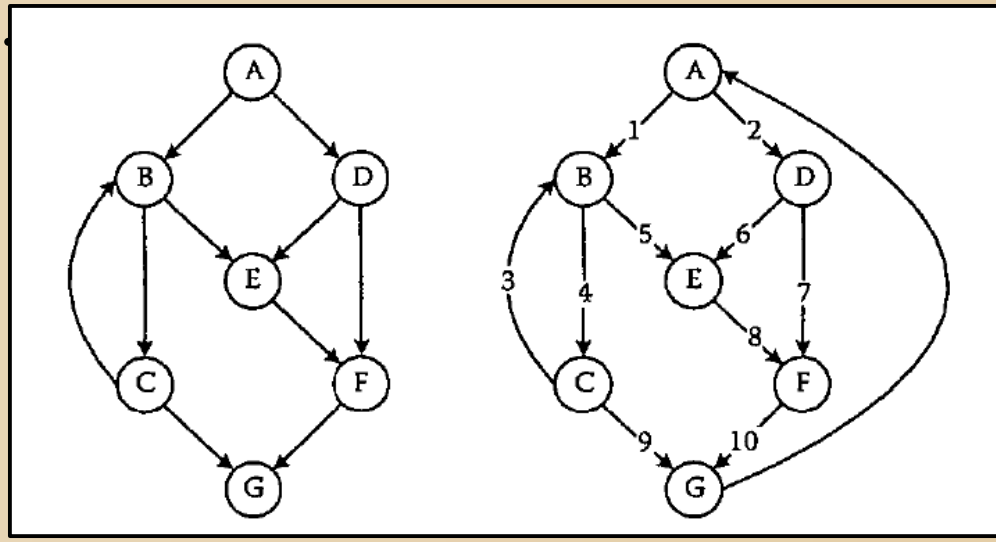
  $V(G) = 10 - 7 + 2 = 5$

Testeo del camino base

- *Camino linealmente independiente*:

  Cualquier camino del programa que **introduce** un nuevo conjunto de sentencias (**nuevo nodo**) o una nueva condición (**nuevo arco**).

  - p1: A, B, C, G



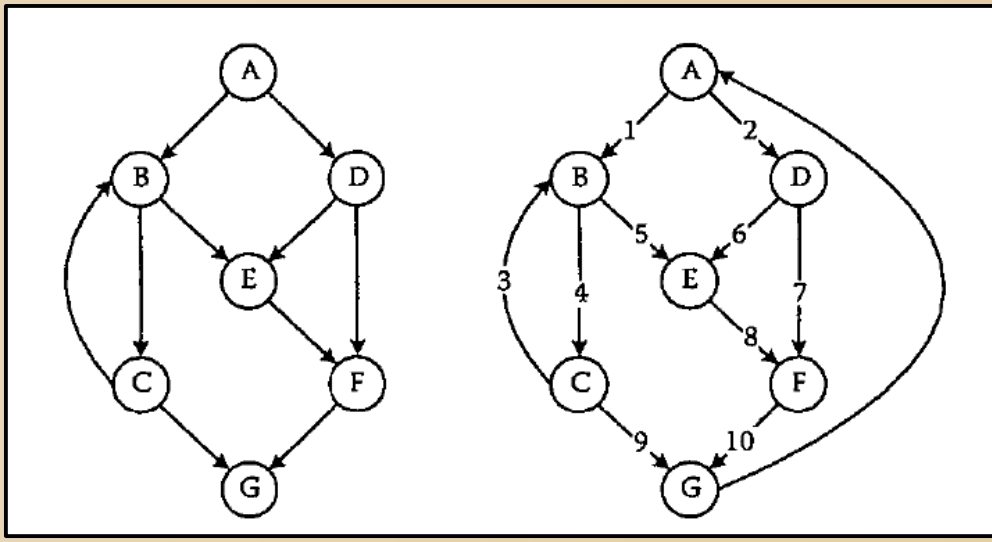
A	B	C	D	E	F	G	1	2	3	4	5	6	7	8	9	10
p1	p1	p1				p1	p1			p1					p1	

Testeo del camino base

- *Camino linealmente independiente*:

  Cualquier camino del programa que introduce un nuevo conjunto de sentencias (nuevo nodo) o una nueva condición (nuevo arco).

  - p1: A, B, C, G
  - p2: A, B, C, B, C, G



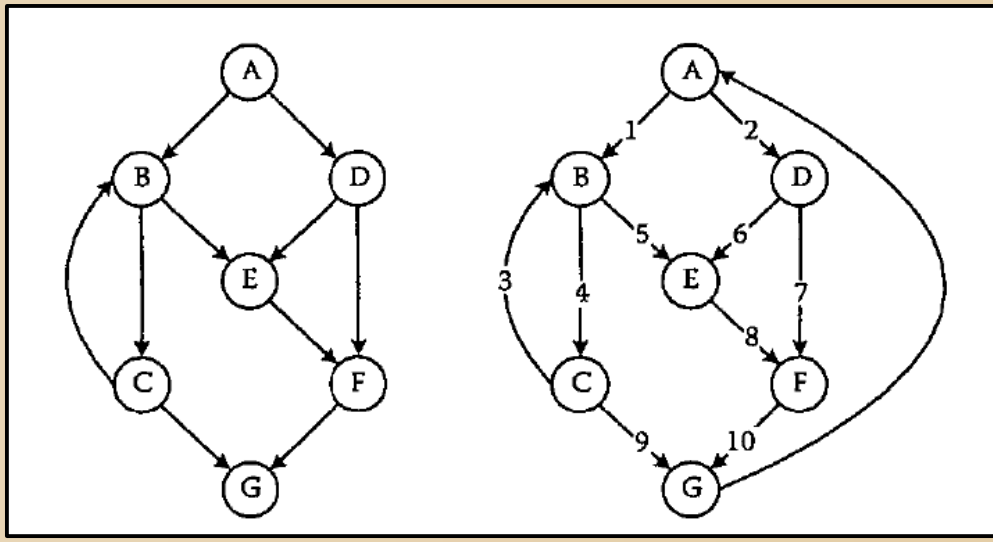
A	B	C	D	E	F	G	1	2	3	4	5	6	7	8	9	10
p1 p2	p1 p2	p1 p2				p1 p2	p1 p2		p2	p1 p2					p1 p2	

Testeo del camino base

- *Camino linealmente independiente*:

  Cualquier camino del programa que introduce un nuevo conjunto de sentencias (nuevo nodo) o una nueva condición (nuevo arco).

  - p1: A, B, C, G
  - p2: A, B, C, B, C, G
  - p3: A, B, E, F, G



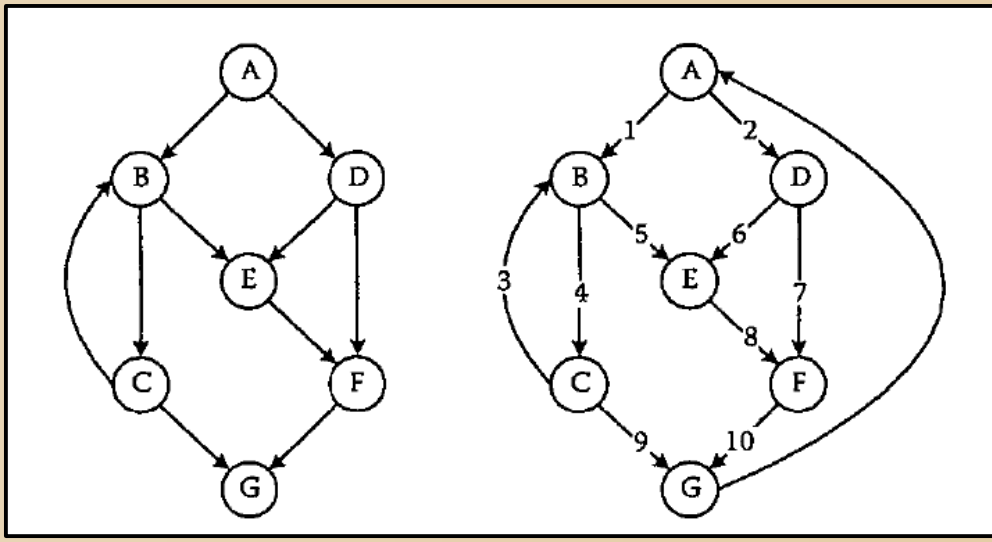
A	B	C	D	E	F	G	1	2	3	4	5	6	7	8	9	10
p1 p2 p3	p1 p2 p3	p1 p2		p3	p3	p1 p2 p3	p1 p2 p3		p2	p1 p2	p3			p3	p1 p2	p3

Testeo del camino base

- *Camino linealmente independiente*:

  Cualquier camino del programa que introduce un nuevo conjunto de sentencias (nuevo nodo) o una nueva condición (nuevo arco).

  - p1: A, B, C, G
  - p2: A, B, C, B, C, G
  - p3: A, B, E, F, G
  - p4: A, D, E, F, G



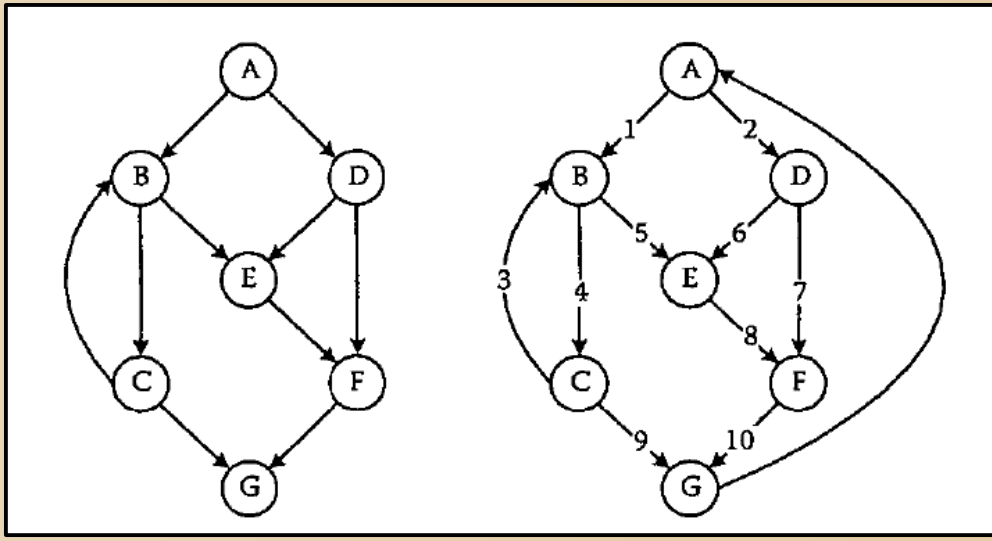
A	B	C	D	E	F	G	1	2	3	4	5	6	7	8	9	10
p1 p2 p3 p4	p1 p2 p3	p1 p2	p4	p3 p4	p3 p4	p1 p2 p3 p4	p1 p2 p3	p4	p2	p1 p2	p3	p4		p3 p4	p1 p2	p3 p4

Testeo del camino base

- *Camino linealmente independiente*:

  Cualquier camino del programa que introduce un nuevo conjunto de sentencias (nuevo nodo) o una nueva condición (nuevo arco).

  - p1: A, B, C, G
  - p2: A, B, C, B, C, G
  - p3: A, B, E, F, G
  - p4: A, D, E, F, G
  - p5: A, D, F, G



A	B	C	D	E	F	G	1	2	3	4	5	6	7	8	9	10
p1 p2 p3 p4 p5	p1 p2 p3	p1 p2	p4 p5	p3 p4	p3 p4 p5	p1 p2 p3 p4 p5	p1 p2 p3	p4 p5	p2	p1 p2	p3	p4	p5	p3 p4	p1 p2	p3 p4 p5

Testeo del camino base: Casos de prueba

- Partiendo del código, calculamos el grafo de DD-Paths.
- Determinamos la complejidad ciclomática del gráfo resultante.
- Determinamos el conjunto básico de caminos independientes.
- Preparamos los casos de prueba que forzarán la ejecución de cada camino del conjunto básico.
  - ¿Es posible que alguno de los casos de prueba desarrollados no sea ejecutable?

Prueba de la condición

- Ejercita las condiciones lógicas requeridas en el módulo de un programa.
- La intuición es buscar errores en las condiciones, tales como variables, operadores, etc.
- Existen distintas estrategias:
  - de ramificaciones (V o F)
  - del dominio (con n variables tenemos 2^n casos)

Condiciones múltiples

- Si sólo se prueban las condiciones para asegurarnos de que se ejercitan todos sus resultados, estamos dejando afuera posibilidades.
- Las condiciones complejas pueden tener *más de una manera* de cumplirse o no cumplirse.
- Una posibilidad es hacer una tabla de decisión.
- Otra es recodificar las condiciones complejas en lógica *if-then-else* anidada: surgen entonces más *DD-paths* para contemplar.

Prueba de bucles simples

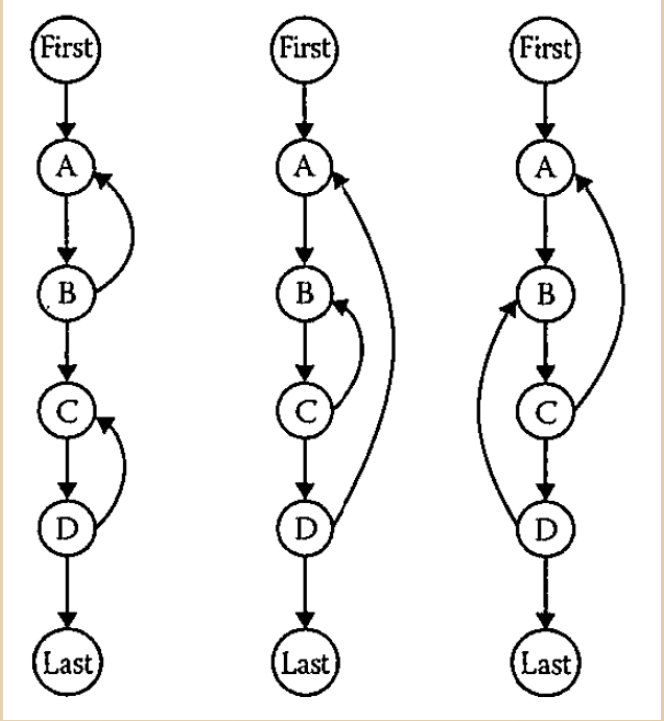
- La aproximación más simple al análisis de bucles es que cada vuelta involucra una decisión: atravesar el bucle o salir de él (o no entrar).

- También puede analizarse la *cantidad* de veces que se atraviesa, con posibilidad de una cota en dicho valor.

Prueba de bucles

Se centra exclusivamente en la validez de las construcciones de bucles:

- Bucles simples
- Bucles concatenados
- Bucles anidados
- Bucles no estructurados, o "*anudados*" (si bien no pueden darse orgánicamente en lenguajes estructurados, pueden surgir en Java con try/catch)



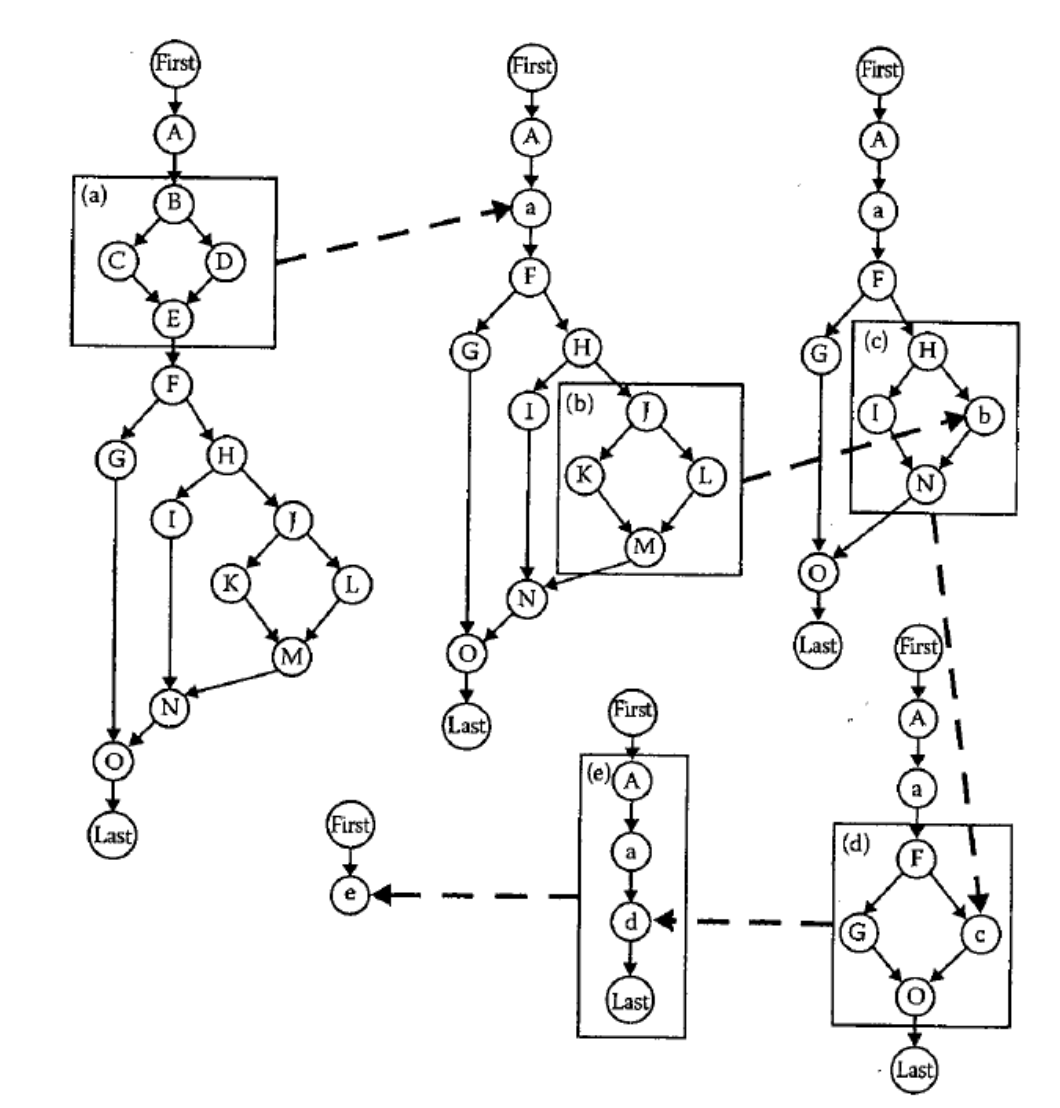
Prueba de bucles

- Los concatenados independientes son simplemente una secuencia de bucles.
- Para los anidados, una estrategia posible es:
  - Comenzar por el bucle de más adentro.
  - Hacer testeo de bucle simple, con el bucle de afuera en un valor mínimo, y los demás con valores típicos.
  - Trabajar hacia afuera, reemplazando cada bucle testeado por un nodo.

Complejidad esencial

- Podemos usar la noción de complejidad ciclomática para definir la *complejidad esencial*.
- Es la complejidad ciclomática del grafo que resulta de *condensar iterativamente* las construcciones estructuradas
- Un programa estructurado *siempre* puede condensarse en un único nodo, y por lo tanto tiene complejidad esencial 1.

Condensación de grafos



Algunas observaciones

- Como vamos a ver la clase que viene las pruebas de caja negra nos separan mucho del código.
- Sin embargo, el análisis de caminos en grafos tal vez nos lleve *demasiado cerca* del código.
- Los criterios y métricas discutidas nos da una manera de evaluar la *calidad del testing*, no una forma ideal de generar casos de test.
- Por ejemplo, pueden usarse como información adicional para los casos de test de caja nerga: si atraviesan los mismos caminos, no aportan nada nuevo.

Bibliografía

P. Jorgersen: "*Software Testing: A Craftsman's Approach*", 4th Ed. Auerbach Publications, 2013. Capítulos 8 y 9.