9. Testing

Introducción

El testing es una parte fundamental del proceso de desarrollo de software que se realiza para garantizar la calidad del producto final.

Su principal objetivo es crear **un producto con alta calidad y productividad** a través de la mínima presencia de defectos en el mismo.

> 💡 **Relación calidad-testing**
> Recordemos, que tener un producto con alta calidad significa respetar características como confiabilidad, mantenibilidad, etc, y mientras más testing tengamos, menos defectos en nuestro producto, por lo que habría menos chances de que el mismo falle y la confiabilidad aumentará.

El testing intentará encontrar los **defectos** que las revisiones humanas no detectan.

> **Defecto**: bug, causa un desperfecto.

> **Desperfecto:** ocurre cuando el comportamiento de un SW es distinto al esperado o especificado. Éste implica la ocurrencia de un defecto.

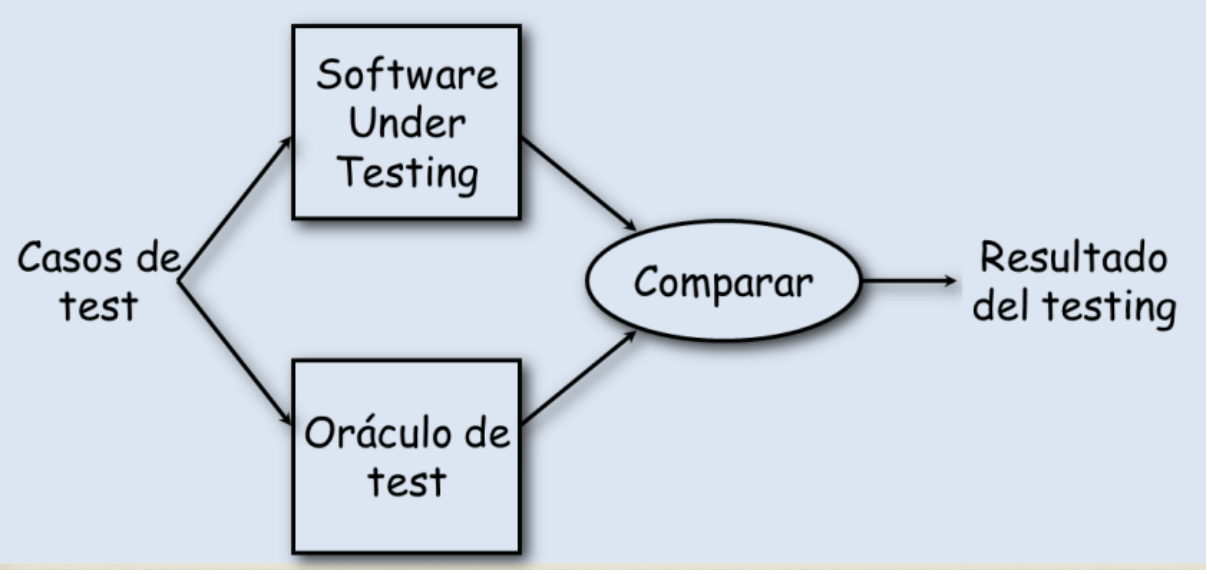
Para esto, se ejecutan los programas siguiendo un conjunto de casos de test. Si hay desperfectos en la ejecución de un caso de test, entonces, con debugging, tendremos mejores posibilidades de encontrar el defecto real.

¿Cómo sabemos si durante la ejecución de un programa, obtenemos un desperfecto? Con el oráculo de test ⬇️

Oráculo de test ⭐

A la hora de verificar la ocurrencia de un desperfecto en la ejecución de un caso de test, necesitamos saber cómo es el comportamiento correcto del SW en ese caso. Para ésto, usamos el oráculo de test.

En otras palabras, el oráculo de test nos dirá cuál debería ser el comportamiento correcto del SW en un caso de ejecución para poder detectar los desperfectos a la hora de testeo.



Siguiendo los casos de test, los tests se ejecutan usando el SW y, una vez obtenido el resultado, comparamos con lo que el oráculo de test nos dice que deberíamos haber obtenido bajo un correcto funcionamiento del SW.

- El oráculo de test usa la especificación para decidir el “comportamiento correcto”.
- Las mismas especificaciones pueden tener errores.

Casos de test y criterios de selección ⭐

Los casos de tests a usar deben ser exhaustivos, ya que queremos evidenciar la mayor cantidad de fallas. El problema es que esto produce un alto costo, ya que mientras mas testing tengamos, mas $ hay que poner.

Para esto, se tiene un **criterio de selección** de tests que **especifica las condiciones que el conjunto de casos de test debe satisfacer con respecto al programa y a la**

especificación. En otras palabras, ayudan a elegir qué casos de prueba incluir en el conjunto.

Para elegir éstos criterios, deben seguirse dos propiedades:

- Confiabilidad: un criterio es confiable si un conjunto de casos de prueba cumple con el criterio, todos los casos en ese conjunto identifican los mismos errores en el software.

  👉 Si diferentes casos de prueba cumplen con el mismo criterio pero detectan diferentes errores, no se puede confiar en los resultados de las pruebas.

- Validez: un criterio es válido si para cualquier error en el programa, hay un conjunto de casos de test que cumple tal criterio y detecta el error.

> 💡 Es casi imposible tener un criterio confiable y válido al mismo tiempo, y que además sea satisfecho por una cantidad manejable de casos de test.

Bien, teniendo ahora una idea de qué son los casos de tests, para qué se usan, veamos cómo podemos saber si estamos construyendo casos de tests útiles.

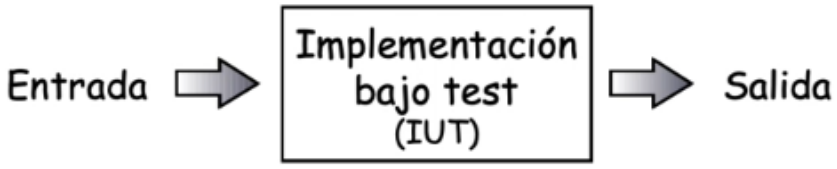
Testing de caja negra ⭐

Es un enfoque de construcción de casos de test. Aquí, el SW se trata como una caja negra; uno supone la implementación del SW (no la tenemos).

Lo que se busca con el testing de caja negra, es hacer casos de test que verifiquen que el comportamiento esperado del SW, está especificado.

Tenemos:

- Entrada: valores que ingresan, cambiaremos estos valores para testear
- Salida: analizaremos las salidas correspondiente a cada entrada.



Los casos de test se seleccionan sólo a partir de la especificación. Como dijimos antes, no se usa la estructura interna del código.

> 💡 Como nos basamos en la especificación:
> - Si testeamos el sistema: nuestra SRS define el comportamiento esperado.
> - Si testeamos módulos: la especificación producida en el diseño define el comportamiento esperado.

Desventajas y mejores

Como es muy costoso testear de forma exhaustiva usando este enfoque (o sea, tener casos de tests con todos los elementos del espacio de entrada especificados), tenemos formas de mejorar el enfoque.

1️⃣ Dividir el espacio de entrada en clases de equivalencias

Si el SW funciona (o no) para un caso de test en una clase, muy probablemente funcione de la misma manera para todos los elementos de la misma clase. Ésta división que se hace es aproximada, no hay un método para hacerla perfectamente.

A tener en cuenta:

- Ésto funciona sólo porque la especificación requiere el mismo comportamiento en todos los elementos de una misma clase.
- Cada condición especificada como entrada, es una clase de equivalencia. Aquellas entradas inválidas también deben formar parte de una clase. ⬇️

Ej.: se especifica el rango $0 \leq x \leq MAX$. Luego:

- el rango [0..MAX] forma una clase,
- $x < 0$ define una clase inválida,
- $x > MAX$ define otra clase inválida.

Ejemplo de división por clase de equivalencia:

Ejemplo:

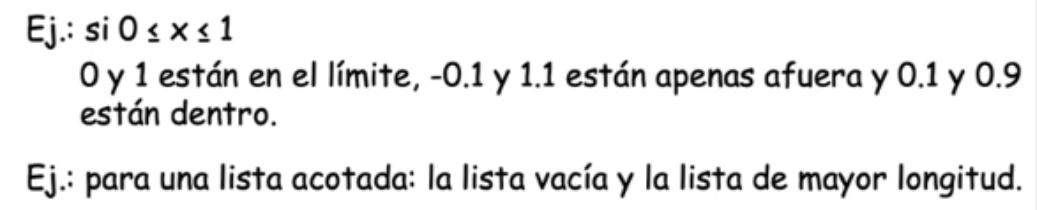
- Considerar un programa que toma dos entradas: un string **s** de longitud **N** y un entero **n**.
- El programa determina los **n** caracteres más frecuentes.
- Quien realiza el test cree que el programador puede haber tratado separadamente a los distintos tipos de caracteres.

Entrada	Clase de eq. válidas	Clase de eq. inválidas
s	1. Contiene números 2. Contiene letras mayúsculas 3. Contiene letras minúsculas 4. Contiene caract. especiales 5. El string es de longitud ≤ N	1. Caracteres no ASCII 2. Longitud del string ≥ N
n	6. Está en el rango válido	3. Está fuera del rango válido

Hay dos formas de encarar estos ejercicios para saber qué casos de tests tenemos:

- Seleccionar cada caso de test cubriendo tantas clases como sea posible.

    - Siguiendo esta forma, tendríamos un test que toma un string que cubre todas las clases de equivalencias (o sea, string s que tiene números, letras minúsculas, mayúsculas, caracteres especiales y de longitud ≤ N, con n fijo), y tenemos un test para cada clase inválida (porque son disjuntos). Daría un total de 4 tests.
- O dar un caso de test que cubra a lo sumo una clase válida por cada entrada.
    - Siguiendo esta forma, tendríamos un string s_1 que cumple la clase 1 que es que contiene números, otro string s_2 que cumple la clase 2 que contiene mayúsculas, etc … y un test para cada clase de equivalencia inválida. Tendríamos 6 + 3 tests totales.

> 💡 En esta mejora, es importante tener en cuenta los valores límites de las clases de equivalencia. Para eso, para cada clase de equivalencia, elegir valores en los límites de la clase, y valores justo fuera y dentro de los límites (+1 valor normal). Ésto sería un total de 6 valores de límite + 1 valor normal.
>
> 
>
>
> También considerar las salidas y producir casos de test que generen salidas sobre los límites.

2️⃣ Grafo de causa-efecto

Este método sirve para identificar y visualizar las relaciones que hay entre distintas condiciones de entrada (**causas**) y sus **efectos** correspondientes en el sistema. Lo que se hace es:

1. Se identifican causas que afecten al sistema y los efectos en el mismo resultantes.
2. Se crea un grafo donde las causas y los efectos son nodos. Las aristas se usan para conectar las causas a los efectos y mostrar las relaciones de dependencia. Ej: si una causa influye en varios efectos, se dibujan aristas desde la causa a cada uno de los efectos correspondientes.
    a. Existen nodos “and” y “or” para combinar causalidad.

> Un nodo "**and**" implica que todas las condiciones de entrada relacionadas deben ser verdaderas para que ocurra el efecto, mientras que un nodo "**or**" significa que al menos una de las condiciones debe ser verdadera para que se produzca el efecto.

3. Se consideran dependencias: las aristas en el grafo pueden tener dos dependencias distintas, **positivas** o **negativas**.
    a. Positiva: la causa contribuye de manera directa o positiva al efecto
    b. Negativa: la causa puede prevenir o inhibir el efecto.
4. Conformo un cuadro que indica la cantidad de tests, y qué efecto testea y qué causas necesita para ese efecto.

Ejemplo de esto:

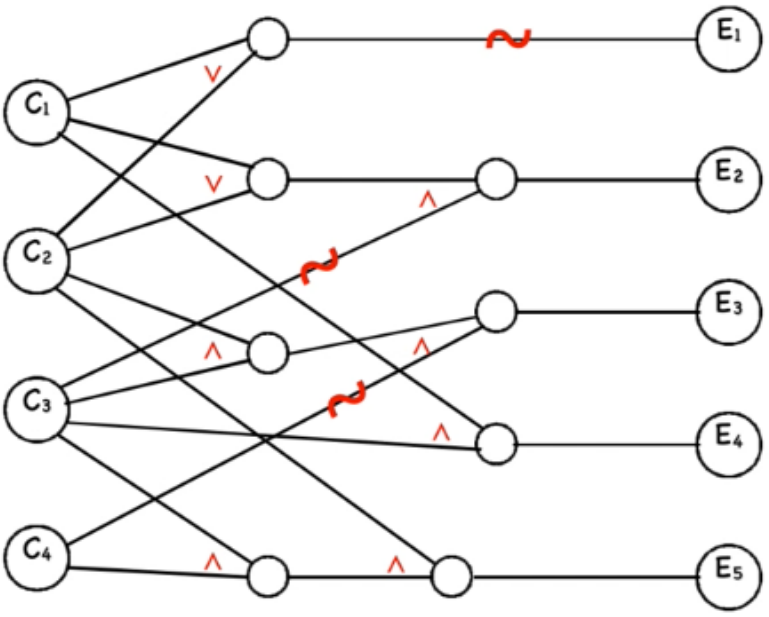
- Una base de datos bancaria que permite dos comandos:
  - Acreditar una cantidad en una cuenta.
  - Debitar una cantidad de una cuenta.
- Requerimientos:
  - Si se pide acreditar y el número de cuenta es válido => acreditar.
  - Si se pide debitar, el número de cuenta es válido, y la cantidad es menor al balance => debitar.
  - Si el comando es invalido => mensaje apropiado.

1. Cómo serán las causas y efectos en este ejemplo?

Causas
C1: el comando es “acreditar”
C2: el comando es “debitar”
C3: número de cuenta es válido
C4: la cantidad es válida

Efectos
E1: Imprimir “Comando inválido”
E2: Imprimir “número de cuenta inválida”
E3: Imprimir “cantidad débito inválida”
E4: Acreditar en la cuenta
E5: Debitar de la cuenta

2. Cómo será el grafo?



- los símbolos ∧ (AND) y ∨ (OR) simbolizan las causas necesarias para conectarse a un nodo.
- los piruletes ~ demuestran la negatividad. O sea, si está sobre una arista, significa que es la causa opuesta a la causa original.

Notemos que para llegar al efecto E3, por ejemplo, debemos haber tenido las causas C2, C3 y C4 al mismo tiempo, y además, C4 tiene su arista negativa. Recordemos que C4 es “la cantidad es válida”, pero como tiene la arista negativa, termina siendo “la cantidad es inválida”.

4. Como sería el cuadro?

#	1	2	3	4	5	6
C1	0	1	X	X	X	1
C2	0	X	1	1	1	X
C3	X	0	0	1	1	1
C4	X	X	X	0	1	X
E1	1					
E2		1	1			
E3				1		
E4					1	
E5						1

La primer fila indica los test que voy a hacer, teniendo como efecto de ese test el efecto que se marque con un 1 en su misma fila. Ejemplo, test 1 quiero ver E1. Luego, marco las causas que necesito para que se genere E1. Las causas que no afectan al efecto elegido, se marcan con una X.

En el test 1, para generar E1, necesito tener si o si la C1 (”comando acreditar”) y C2 (”comando debitar”) como negativas, ya que E1 es “comando inválido”.

En el test 2, quiero obtener E2, necesito tener si o si C1 como positiva y C3 como negativa.

En el test 3, quiero obtener E2 de nuevo ya que se puede obtener E2 con un orden de variables distinto, en este caso, C2 positivo y C3 negativo.

…

así hasta tener todos los tests sobre todos los efectos teniendo todas las combinaciones entre variables posibles.

3️⃣ Testing de a pares

Acá vamos a ver la cantidad de tests a realizar cuando tenemos que los casos a testear se hacen sobre múltiples parámetros que pueden tomar valores distintos o rangos distintos de valores. En este caso, tenemos defectos de dos tipos:

- Modo Simple: están relacionados con una sola condición o parámetro (ej: software no puede imprimir en un cierto tipo dado de impresora). Calcular la cantidad de tests para un defecto de modo simple es fácil: si hay **n** parámetros con **m** tipos de valores cada uno, tendremos **m** casos de tests por parámetro.
- Modo Doble: defectos que se manifiestan sólo cuando se combinan varias condiciones/parámetros, pero basta con analizar dos para justificar el defecto.

Ej.: el software de la cuenta del teléfono calcula mal las llamadas nocturnas (un parámetro) a un país particular (otro parámetro).

este no es un ejemplo muy ilustrativo, je.

Como sólo se analizan dos, a esto lo llamamos testear de a pares.

Usaremos una fórmula para conocer cuántos casos de test necesitaremos:

$$m^2 * n * (n-1)/2$$

donde m es la cantidad de valores que un parámetro puede tomar, y n es la cantidad de parámetros totales que hay.

Ejemplo:

Supongamos un producto de software multiplataforma que usa browsers como interfaz y debe trabajar sobre distintos sistemas operativos:

Tenemos tres parámetros:

- SO (parámetro A): Linux, MacOSX, Windows vista.
- Tamaño memoria (B): 512MB, 1GB, 2GB
- Browser (C): Firefox, Opera, Safari/Konqueror

En este caso, tenemos 3 parámetros, por lo tanto $n = 3$. Además, cada parámetro puede tomar 3 valores en total, por lo tanto $m = 3$. Entonces:

$$3^2 * 3 * (3-1)/2 = 9 * 3 * 1 = 27$$

Tenemos 27 test que realizar. Ahora, como tengo 3 parámetros/condiciones, tengo que hacer una tabla con los casos de a pares que voy a analizar (que en total son 27, como vimos recién en el cálculo)

A	B	C	Pairs
a1	b1	c1	(a1,b1) (a1,c1) (b1,c1)
a1	b2	c2	(a1,b2) (a1,c2) (b2,c2)
a1	b3	c3	(a1,b3) (a1,c3) (b3,c3)
a2	b1	c2	(a2,b1) (a2,c2) (b1,c2)
a2	b2	c3	(a2,b2) (a2,c3) (b2,c3)
a2	b3	c1	(a2,b3) (a2,c1) (b3,c1)
a3	b1	c3	(a3,b1) (a3,c3) (b1,c3)
a3	b2	c1	(a3,b2) (a3,c1) (b2,c1)
a3	b3	c2	(a3,b3) (a3,c2) (b3,c2)

4️⃣ Casos especiales 🚫

5️⃣ Testing basado en estados

Es una técnica que se centra en modelar un sistema como una máquina de estados para evaluar su comportamiento en diferentes estados lógicos. Cada estado en la máquina representará un estado lógico del sistema en un momento dado, enfocándonos cómo el SW responde a eventos de entrada en diferentes estados del sistema.

Tendremos CUATRO componentes:

- **Conjunto de estados lógicos**: representan el impacto acumulativo del sistema
- **Conjunto de transiciones**: representan el cambio de estado en respuesta a algún evento de entrada
- **Conjunto de eventos**: son las entradas al sistema
- **Conjunto de acciones**: son las salidas producidas en respuesta a los eventos de acuerdo al estado actual

Buscaremos generar un modelo de estado a partir de la especificación o del diseño. En la generación del modelo, se deben identificar los estados, las transiciones y los eventos del mismo.

Luego de la creación del modelo, diseñaremos los casos de prueba siguiendo los siguientes criterios:

- Cobertura de transiciones: el conjunto T de casos de test debe asegurar que toda transición sea ejecutada. Esto implica que se debe ejecutar al menos un caso de prueba para cada transición posible en el modelo de estado.
- Cobertura de parte de transiciones: T debe ejecutar todo par de transiciones adyacentes que entran y salen de un estado.
- Cobertura de árbol de transiciones: T debe ejecutar todos los caminos simples (del estado inicial al final, o a uno visitado)

Luego de diseñar los casos de test, se ejecutan los casos en el sistema, observando cómo el SW responde a los eventos de entrada en función de su estado actual. Se registran las salidas y se comparan con las salidas esperadas, así luego se evalúa si el SW se comporta de acuerdo con las especificaciones y los requisitos en todos los estados probados.

Testing de caja blanca ⭐

Este tipo de testing, se enfoca en la estructura interna del código fuente de un programa o SW. El objetivo es el mismo que en caja negra, con la única diferencia de que ahora sí tenemos el código disponible.

Debido a que tenemos el código, los casos de test se derivan a partir del mismo. Existen varios criterios para seleccionar el conjunto de casos de tests. Entre ellos:

1️⃣ Criterio basado en el flujo de control

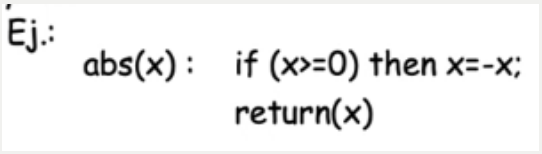
Aquí el programa será un **grafo** de flujo de control donde los **nodos** son bloques de código y las **aristas (i, j)** representan una posible transferencia de control del nodo i al j.

Supondremos la existencia de un nodo inicial y uno final, y se busca identificar caminos que conecten el nodo inicial con el nodo final. Estos caminos son secuencias de nodos que representan posibles rutas de ejecución del programa.

> **Explicación más clara (ta buena) — Chat GPT**
> En términos más simples, cuando el programa se está ejecutando, en algún momento se encuentra en un bloque de código particular, y la transferencia de control se refiere a la decisión de qué bloque de código se ejecutará a continuación. Las aristas en el grafo de flujo de control representan las rutas posibles que puede seguir la ejecución del programa, lo que significa que el programa podría saltar de un bloque de código (nodo i) a otro bloque de código (nodo j) si se cumplen ciertas condiciones o si se ejecuta una instrucción específica.

Para construir los casos de tests aquí, se deben tener en cuenta los siguientes criterios:

💡 Antes de ver los criterios, es importante entender la siguiente notación. Tenemos este programa



tiene un error apropósito porque se usa en un ejemplo de algo específico, pero lo uso para explicar notación nomás, asique obviar.

Aquí, si pensamos en un conjunto de tests para este código, representamos los tests por un **par**:
{ {x = 0, 0} } donde:

- Entrada que mi sistema recibe
- Resultado que mi sistema debería darme para la entrada puesta

Es eso nomas, que los casos de test los representamos por pares.

▼ **A: Criterio de cobertura de sentencia.**

Cada sentencia en el código fuente se debe ejercitar al menos una vez durante las pruebas. Esto significa que el conjunto de casos de prueba debe ser diseñado de manera

que todas las sentencias del código se activen en al menos una ocasión durante la ejecución de las pruebas

A tener en cuenta!

1. En el caso de condicionales, una sentencia es el `if (condición)`, no es que una sentencia sería el if (condición) + caso para cada caso. Es decir, que una sentencia de un condicional se ejercite, no significa que se tienen que contemplar todos los casos de todos los condicionales en el conjunto de pruebas (eso se hace en cobertura de ramificaciones que se ve dsp). En cambio, se tienen que hacer casos de test que prueben la sentencias nomás, independientemente del caso que se ingrese.

   Ejemplo: se tiene el siguiente código que calcula un descuento según la cantidad de productos comprados y si el cliente tiene membresía o no.

   ```
   def calcularDescuento(cantidad, tieneMembresia):
       if cantidad >= 10:
           if tieneMembresia:
               descuento = 0.20  # 20% de descuento para miembros con 10 o más productos
           else:
               descuento = 0.10  # 10% de descuento para no miembros con 10 o más productos
       else:
           descuento = 0.05  # 5% de descuento para compras de menos de 10 productos
       return descuento
   ```

   En este caso, según la cobertura de sentencia, tendríamos :

   - Al menos un caso de test que ejercite la sentencia del primer if, sea para ejercitarlo como verdadero o falso:
     { {cantidad = 5 y tieneMembresia = True, descuento = 0.05} }
   - Y como tenemos otra sentencia if dentro del primer if, necesitamos otro caso de test para ejecutar esa sentencia también. Si este if anidado no estuviera, con el caso test de arriba bastaría. El otro caso de test quedaría:
     { {cantidad = 10 y tieneMembresia = False, descuento = 0.10} }

   Notar que el valor tieneMembresía es indiferente, porque vuelvo a repetir, basta entrar a la sentencia del if.

2. Como vimos en 1, el criterio de cobertura de sentencia no garantiza detección de todos los errores, ya que no se testean todos los casos de la sentencias de los condicionales, sólo al menos una. Como los demás casos no se ejecutan, no podemos asegurar que esa parte del código no tenga errores. Entonces, aunque una sentencia se ejecuta al 100%, no significa que no haya errores.

   Ejemplo:

```
def abs(x):
  if (x>=0):
    x = -x
  return x
```

En este programa hay un error, que es que si ingreso x = 5, me devolverá -5, que está mal, porque -5 no es el absoluto de 5. Por lo tanto, si tengo un caso de test de sentencia de la siguiente forma:
{ {x=0, 0} }, el test pasará y será suficiente, porque la sentencia se ejecuta al menos una vez. Lo malo es que este test no refleja los errores que sí están presentes.

3. Puede haber nodos inalcanzables: puede haber rutas de ejecución que no se activan nunca en la práctica, o partes del código que están condicionalmente excluidas en función de ciertas condiciones. Éstos fenómenos pueden impedir lograr una cobertura completa.

▼ **B: Criterio de cobertura de ramificaciones.**

Cada arista debe ejecutarse al menos una vez en el testing, cada decisión debe ejercitarse como verdadera o como falsa. Si este criterio se cumple, entonces se cumple el de sentencias (A). O sea, cobertura de ramificaciones ⇒ cobertura de sentencias.

Ejemplo: tenemos el programa de ejemplo de cobertura de sentencias

```
def calcularDescuento(cantidad, tieneMembresia):
    if cantidad >= 10:
        if tieneMembresia:
            descuento = 0.20  # 20% de descuento para miembros con 10 o más productos
        else:
            descuento = 0.10  # 10% de descuento para no miembros con 10 o más productos
    else:
        descuento = 0.05  # 5% de descuento para compras de menos de 10 productos
    return descuento
```

En el código, tenemos una decisión o ramificación basada en la cantidad de productos comprados y si se tiene membresía o no. Entonces, el criterio de cobertura de ramificaciones nos dice que tenemos que tener casos de test que cubran todas las ramas de todos los ifs, o sea, todos las combinaciones de las decisiones/ifs!
Tendríamos entonces:

- Un caso de test para la rama donde la cantidad ≥ 10 y membresía es true:
  { {cantidad = 10 y tieneMembresía = True, descuento = 0.20} }
- Un caso de test para la rama donde la cantidad ≥ 10 y membresia es false:
  { {cantidad = 10 y tieneMembresía = False, descuento = 0.10} }
- Un caso de test para la rama donde la cantidad < 10 y tieneMembresía = True:
  { {cantidad = 5 y tieneMembresía = True, descuento = 0.05} }

> 💡 Notar, que como cubrimos todas las combinaciones, los casos del criterio de cobertura de sentencia que vimos antes se cumplen también. Por eso el de ramificaciones implica el de sentencias.

A tener en cuenta!

1. Puede pasar que haya casos de test que cubran todas las ramificaciones posibles pero que no encuentren el defecto, mientras que haya casos de test que cubran sentencias que si lo detectan! Ejemplo:

   ```
   def abs(x):
     if (x<0):
       x = x
     return x
   ```

   En este programa, tenemos un error, porque el absoluto de -1 es 1, no -1 como devolvería el programa. Si nuestro caso de tests para sentencias y ramificaciones fueran:

   - Cobertura de sentencias: { {x = -1, 1} }
   - Cobertura de ramificaciones: { {x=1, 1}; (x=0, 0) }

   veamos que si bien la cobertura de ramificaciones testea cuando el condicional es true y false, no detecta el error que hay! En cambio, la cobertura de sentencias sí. Porque cuando se ingrese -1 se va a esperar un 1, pero el programa devolverá -1, lo cual mostrará el defecto.

▼ **C: Criterio de cobertura de caminos.**

Todos los posibles caminos del estado inicial al final deben ser ejercitados. Éste criterio sería el más exhaustivo de los tres ya que garantiza que se prueben todas las combinaciones posibles de instrucciones y condiciones en el código. Si se cumple este criterio, se cumple el de cobertura de bifurcación (que significa que si la ejecución de un programa se bifurca en dos caminos distintos, esos dos caminos nuevos también se ejercitan).

A tener en cuenta!

1. Es difícil de lograr una cobertura de caminos en programas que tienen loops infinitos, porque los caminos serían infinitos.

> 💡 Breve resumen de los criterios:
> - **A**: para asegurar que las sentencias se ejerciten al menos una vez.
> - **B**: para asegurar que las ramificaciones en una decisión/if se ejerciten todas al menos una vez.
> - **C**: es como una unión de A y B, ya que se encarga de que todos los caminos posibles de ejecución del programa se ejerciten al menos una vez.

P.d.: Hay muchos otros criterios que se pueden usar, pero ninguno es suficientemente confiable como para decir que se detectarán todos los errores.

2️⃣ Criterio basado en el flujo de datos ⭐(¡¡¡¡¡¡VA AL PRÁCTICO SI O SI!!!!!)⭐

Este criterio se centra en la relación entre las definiciones y usos de variables en un programa. El objetivo principal es identificar posibles problemas relacionados con la correcta definición y uso de variables en el código.

Para esto, se utiliza un grafo de definición-uso que etiqueta y sigue las definiciones y usos de variables a través del programa.

Conceptos básicos

- Definición (def): representa la asignación de una variable en el código (aparece cuando a una variable le asignamos un valor, sea la primera vez que se le asigna un valor o no). Dentro del grafo definición-uso, usamos la etiqueta `def`.
- Uso-cómputo (uso-c): aparece cuando se usa una variable existente en un cálculo u operación. Se usa la etiqueta `uso-c` en el grafo.
- Uso-predicado (uso-p): aparece cuando se usa una variable en un predicado que afecta a la transferencia de control (como un condicional en un if). No se escribe uso-p en el grafo, sólo se representa la bifurcación a dos nodos con flechas, y sobre las flechas se escribe el nombre de la variable de uso-p.

Construcción del grafo

El grafo de def-uso se construye asociando variables a nodos y aristas del grafo de flujo de control (mencionado en criterio 1 de test de caja blanca). Se tienen tres puntos importantes para la construcción del grafo.

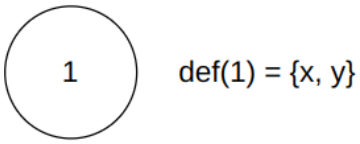
> 💡 Disclaimer: según profe, un nodo = línea de código = sentencia.

1. Por cada nodo i, def(i) es el conjunto de variables para el cual hay una asignación/definición en i. Si no hay asignaciones/definiciones en la línea, denotamos def(n° línea) = ∅.

   Ejemplo: tengo la siguiente línea de código

```
1. scanf(x, y)
```

En este caso, scanf (si recordamos de C) lee de stdin y escribe lo que se lee en las variables que tiene de parámetro (en este caso, x e y). Por lo tanto, hay asignaciones implícitas para x e y. Además (si nos adelantamos un toque, porque uso-c se ve en el punto de abajo), como no hay variables que se estén usando en una operación, marcamos al vacío como variable de uso-c. Entonces, al lado del nodo escribiremos:



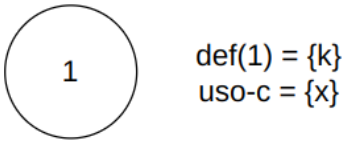
acá faltaría tener c-use = ∅, pero como se explica c-use en el punto siguiente no me quise adelantar.

2. Por cada nodo i, c-use(i) es el conjunto de variables para el cual hay un uso-c. Si no hay variables de c-use en la línea, denotamos def(n° línea) = ∅.

   Ejemplo: tengo la siguiente línea de código

   ```
   1. k = 1 + x
   ```

   En este caso, se está usando la x en una operación de suma, entonces x es de uso-c. Pero, además, se está asignando a k un valor, asi que k es parte del conjunto def(1). Entonces, al lado del nodo escribiremos:

3. Por cada arista (i, j), p-use(i,j) es el conjunto de variables para el cual hay un uso-p.

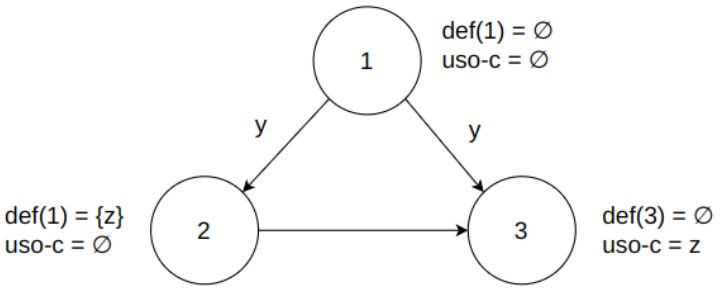
   Recordemos que las variables de uso-p no se escriben al lado del nodo como partes de un conjunto, se representa el uso-p a través de una bifurcación.

   Ejemplo: tenemos la siguiente línea de código

   ```
   1. if (y < 0)
   2.    z = 8
   3. printf(z);
   ```

   Aquí tenemos una bifurcación en el nodo 1, donde podemos ir, o al nodo 2 (si se cumple la condición del if) o (si no se cumple) al nodo 3 que sería la del print.

Tener en cuenta que, en el caso de que la condición del if se cumpla y se vaya al nodo 2, del nodo 2 se pasará al nodo 3, ya que la ejecución es secuencial. Entonces, teniendo en cuenta lo visto hasta ahora, tendríamos:



Tener en cuenta que del 2 se va al 3 porque el 3 no es un caso else del if, es una línea que se ejecuta si o si debido a la lógica secuencial. Si fuera un else, sería el mismo grafo pero sin la flecha del 2 al 3.

4. Un camino de i a j se dice libre de definiciones con respecto a una variable x si no hay definiciones de x en todos los nodos intermedios.

Ejemplo sumando todo:

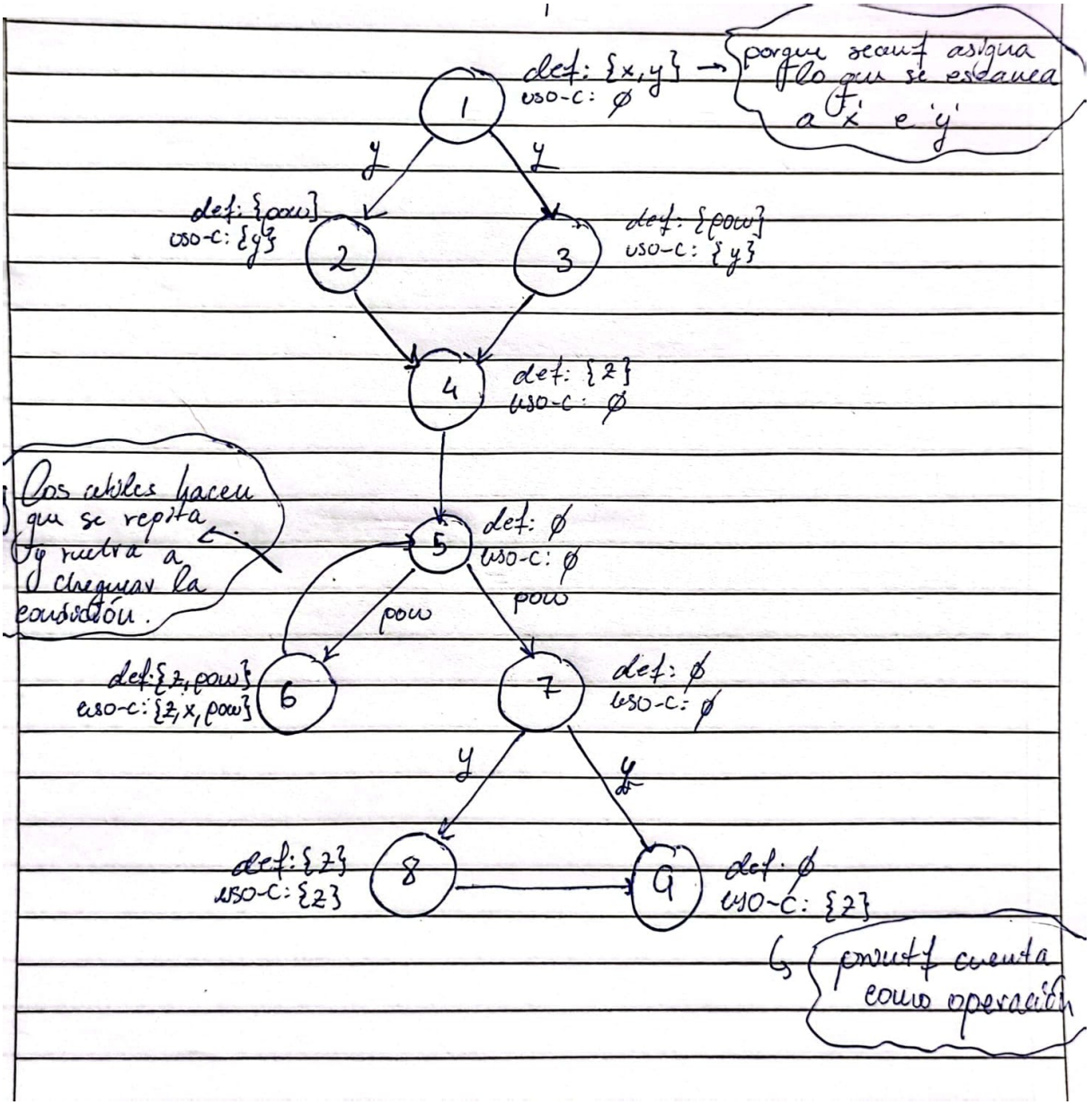
Tenemos el siguiente código:

```
1. scanf(x, y): if (y < 0)
2.      pow = 0 - y;
3. else pow = y;
4. z = 1.0;
5. while (pow != 0)
6.      { z = z * x; pow = pow - 1; }
7. if (y < 0)
8.      z = 1.0/z;
9. printf(z);
```

Usaremos los números de las lineas para representar los bloques de código. Cada línea será (recordemos) un nodo en el grafo.

1. Identificamos cada nodo. Usamos
2. En cada nodo debemos etiquetar los defs y los c-use. Si hay ifs con condiciones, hacemos las bifurcaciones necesarias.
3. Usamos flechas para dictar el siguiente nodo a ejecutar. Si se trata de un condicional, bifurcamos. Si se trata de un while, volvemos al nodo anterior (el de la línea del while).

Entonces tendríamos:



Soporte con herramientas 🚫

Testing Incremental

Es una estrategia de pruebas usadas para detectar todos los efectos posibles a un costo razonable. Se basa en agregar partes no testeadas incrementalmente a la parte ya testeada del software a medida que se desarrolla y se hace el test.

> 💡 En grandes sistemas, se usa siempre el testing incremental.

Niveles de testing 🚫

Plan de test

Se hace al inicio de todo y es un documento general que define el alcance y el enfoque del testing para el proyecto completo. Toma como entrada el plan del proyecto, la SRS, el diseño.

Entonces, en este documento tendremos:

- Especificación de la unidad de test: qué unidad se debe testear.
- Características a testear: funcionalidad, desempeño, usabilidad, nivel de testing a hacer, etc.
- Enfoque: criterio a usar, cuando detenerse, cómo evaluar, etc.
- Cosas entregables como lista de casos de test a usar, resultados de los mismos, reportes o información de cobertura…
- Cronograma y asignación de tareas

Especificación de los casos de test

Si bien el plan se enfoca en cómo proceder y qué testear, es la especificación de los casos de test en sí la que trata con los detalles de testeo de cada unidad.

Ésta se hace separadamente para cada unidad y se hacen de acuerdo con el plan de test que se tiene (enfoques, características, etc). Dentro de cada especificación tendremos:

- Entradas a usar.
- Condiciones que éste testeará.
- Resultado esperado.

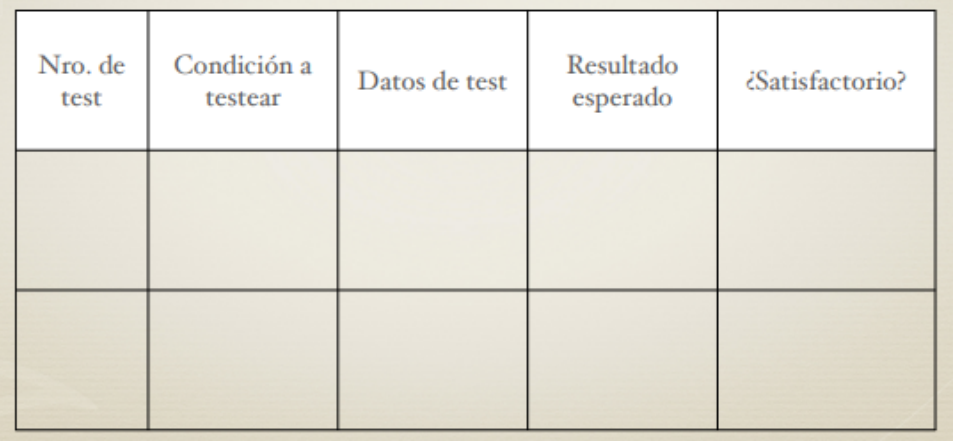
> 💡 La **efectividad** y **costo** del testing dependen del conjunto de casos de test seleccionados.

❓ Cómo podemos determinar si un conjunto de casos de test es bueno (o sea que detecta la mayor cantidad de defectos)?

No existe una forma para saber si un conjunto de casos de test es bueno, pero usualmente expertos suelen revisar estos casos siguiendo el proceso de inspección visto anteriormente para analizar la “bondad” de cada caso.

❓ Qué pinta tiene una especificación de un caso de test?

Las especificaciones se representan usando una tabla:



Nro. de test	Condición a testear	Datos de test	Resultado esperado	¿Satisfactorio?

Ejecución de casos de test y análisis

Algunos aspectos claves relacionados con la ejecución y análisis de tests son:

- En la ejecución se suelen usar “drivers” o “stubs”. Éstos elementos permiten probar una unidad o módulo en aislamiento, incluso si otras partes del sistema no están disponibles.
  - Drivers: módulos o programas temporales que se usan para simular o emular componentes del sistema que aún no se han desarrollado
  - Stubs: componentes que simulan las llamadas a funciones o módulos que aún no existen o están incompletos.
- Se preparan los entornos de prueba si es necesario: por ahí necesitamos configuraciones o módulos adicionales para poder testear. Éstos podrían ser ajustes de red, configuración de hardware, configuración de base de datos, entre otros.
- Automatización en la ejecución: hay tests que pueden automatizarse mediante herramientas especiales (suele usarse para tareas repetitivas). Otros tests necesitan ser realizados de forma manual.
- Se realizan informes de prueba despúes de la ejecución de tests: contienen los resultados de los mismos. Son esenciales para evaluar el estado del SW y su calidad.
- También se hace un seguimiento y control del esfuerzo invertido en el proceso de prueba para saber si se están cumpliendo plazos y asignación adecuada de recursos.
- El tiempo de computadora es también un indicador importante para monitorear el progreso del proceso de pruebas. Incluye cantidad de tiempo necesaria para ejecutar tests, complejidad de los mismos, etc.

Registro de defectos y seguimiento

Los grandes proyectos de SW pueden tener muchos defectos. Pueden ser distintas las personas que identifican a éstos defectos, por lo tanto, se lleva a cabo un seguimiento formal de los mismos para mantener la calidad de SW.

Los defectos generalmente se registran en sistemas de seguimiento de defectos o herramientas de seguimiento que permiten un registro estructurado.

Cada defecto tendrá un ciclo de vida que suele incluir etapas como:

- Identificación
- Asignación para corrección
- Corrección
- Verificación de corrección.

Además, podemos categorizar a los defectos según su tipo (como funcional, lógico, estándares, interfaz de usuario, rendimiento, entre otros) y severidad según el impacto en el SW (crítico, mayor, menor, cosmético)

> 💡 Es común que algunas empresas entreguen SW con errores comunes que no tienen un impacto grande en el SW. Ésto se hace ya que se tienen umbrales de errores y estándares para decidir qué errores se dejan y cuáles se solucionan antes de poner el “producto a la venta”.

> 💡 El registro de defectos se usa también para analizar la tendencia de cómo ocurren las apariciones de errores y sus correcciones.

fin