FACULTAD DE INGENIERIA

Ingeniería de Software

Verificación y Validación

Bibliografía

- Software Engineering 9ed, cap. 8.
  Addison Wesley - Ian Somerville
- Software Engineering 9ed, cap. 15 (parcialmente).

- Generating Test Cases From Use Cases

Errores, Faltas y Fallas

un error humano

puede generar

una falta

(interna)

que puede generar

?!

una falla

(externa)

Fallas del Software

- ¿El software falló?
  - No hace lo requerido (o hace algo que no debería)
- Razones:
  - Las especificaciones no estipulan exactamente lo que el cliente precisa o quiere (reqs. faltantes o incorrectos)
  - Requerimiento no se puede implementar
  - Faltas en el diseño
  - Faltas en el código
- Objetivo: detectar y corregir estas faltas antes de liberar el producto

Objetivos de V&V

- Descubrir defectos
  - Provocar fallas (una forma de detectar defectos)
  - Revisar los productos (otra forma de detectar defectos)
- Evaluar la calidad de los productos
  - El probar o revisar el software da una idea de la calidad del mismo

Identificación y Corrección de Defectos

- Identificación de defectos
  - Determinar qué defecto o defectos causaron la falla

- Corrección de defectos
  - Cambiar el sistema para remover los defectos

Definición de V&V

- Sommerville
  - Verificación
    - Busca comprobar que el sistema cumple con los requerimientos especificados (funcionales y no funcionales)
    - ¿El software está de acuerdo con su especificación?
    - Testing de defectos
  - Validación
    - Busca comprobar que el software hace lo que el usuario espera.
    - ¿El software cumple las expectativas del cliente?
    - Testing de validación

Definición de V&V

- Boehm
  - Verificación
    - ¿Estamos construyendo el producto correctamente?
    - El software debe conformar una especificación.
  - Validación
    - ¿Estamos construyendo el producto correcto?
    - El software debe satisfacer las expectativas.
    - Las especificaciones pueden no representar claramente estas expectativas.

Definición de V&V

- IEEE
  - Verificación
    - Es el proceso de evaluar un sistema o componente, con el fin de determinar si los productos de cierta fase de desarrollo, satisfacen las condiciones impuestas al comienzo de dicha fase
  - Validación
    - Es el proceso de evaluar un sistema o componente, durante o al final del proceso de desarrollo, con el fin de determinar si satisface los requerimientos especificados

Definición de V&V

- El testeo puede demostrar presencia de errores, pero no la ausencia de ellos - Dijkstra
- El testeo exhaustivo no es posible
- El testeo entonces es una actividad basada en el manejo de riesgos y análisis costo beneficio
- La técnica requiere habilidad para determinar:
  - cual es conjunto de casos de prueba adecuado,
  - cuanto esfuerzo es necesario para lograr el nivel adecuado
- V & V apunta a establecer la confianza de que el sistema es “adecuado para el propósito”.

Ejemplo

- El programa lee tres números enteros, los que son interpretados como representaciones de las longitudes de los lados de un triángulo. El programa escribe un mensaje que informa si el triángulo es escaleno, isósceles o equilátero
- Quiero detectar defectos probando (testeando) el programa
- ¿Qué puedo probar?

- Posibles casos a probar:
  - lado1 = 0, lado2 = 1, lado3 = 0 **Resultado** = error
  - lado1 = 2, lado2 = 2, lado3 = 3 **Resultado** = isósceles
- Intuitivamente que otros casos sería bueno probar
  - lado1 = 2, lado2 = 3, lado3 = 4 **Resultado** = escaleno
  - lado1 = 2, lado2 = 2, lado3 = 2 **Resultado** = equilátero
  - ¿Porqué estos casos?
    - Al menos probé un caso para cada respuesta posible del programa (error, escaleno, isósceles, equilátero)
    - Más adelante veremos técnicas para seleccionar casos interesantes
- Estos son *Casos de Prueba*

Ejemplo

- Otra forma para detectar defectos es revisar el código
  - If l1 = l2 or l2 = l3 then

    write (“equilátero”)

    else ...
  - En lugar del *or* me doy cuenta que debería ir un *and*
  - Se puede revisar solo
  - Se puede revisar en grupos
    - Veremos más adelante técnicas conocidas para revisiones en grupo

Tipos de Faltas

- en algoritmos
- de sintaxis
- de precisión y cálculo
- de documentación
- de estrés o sobrecarga
- de capacidad o de borde
- de sincronización o coordinación
- de capacidad de procesamiento o desempeño
- de recuperación
- de estándares y procedimientos
- relativos al hardware o software del sistema

Tipos de Faltas

- En algoritmos
  - Faltas típicas
    - Preguntar por la condición equivocada
    - No inicializar variables
    - No evaluar una condición particular
    - Comparar variables de tipos no adecuados
- De sintaxis
  - Ejemplo: Confundir un 0 por una O
    - Los compiladores detectan la mayoría

Tipos de Faltas

- De precisión o de cálculo
  - Faltas típicas
    - Formulas no implementadas correctamente
    - No entender el orden correcto de las operaciones
    - Faltas de precisión
- De documentación
  - La documentación no es consistente con lo que hace el software
    - Ejemplo: El manual de usuario tiene un ejemplo que no funciona en el sistema

Tipos de Faltas

- De estrés o sobrecarga
  - Exceder el tamaño máximo de un área de almacenamiento intermedio
  - Ejemplos
    - El sistema funciona bien con 100 usuarios pero no con 110
    - Degradación del sistema (memory leak).
- De capacidad o de borde
  - Más de lo que el sistema puede manejar
  - Ejemplos
    - El sistema funciona bien con importes <1.000.000
    - Año 2000: sistema trata bien fechas hasta el 31/12/99

Tipos de Faltas

- De sincronización o coordinación
  - No cumplir requerimiento de tiempo o frecuencia
  - Ejemplo
    - Comunicación entre procesos con faltas
- De capacidad de procesamiento o desempeño
  - No terminar el trabajo en el tiempo requerido
  - Tiempo de respuesta inadecuado
- De recuperación
  - No poder volver a un estado normal luego de una falla

Tipos de Faltas

- De estándares o procedimientos
  - No cumplir con la definición de estándares y/o procedimientos
- De hardware o software del sistema
  - Incompatibilidad entre componentes

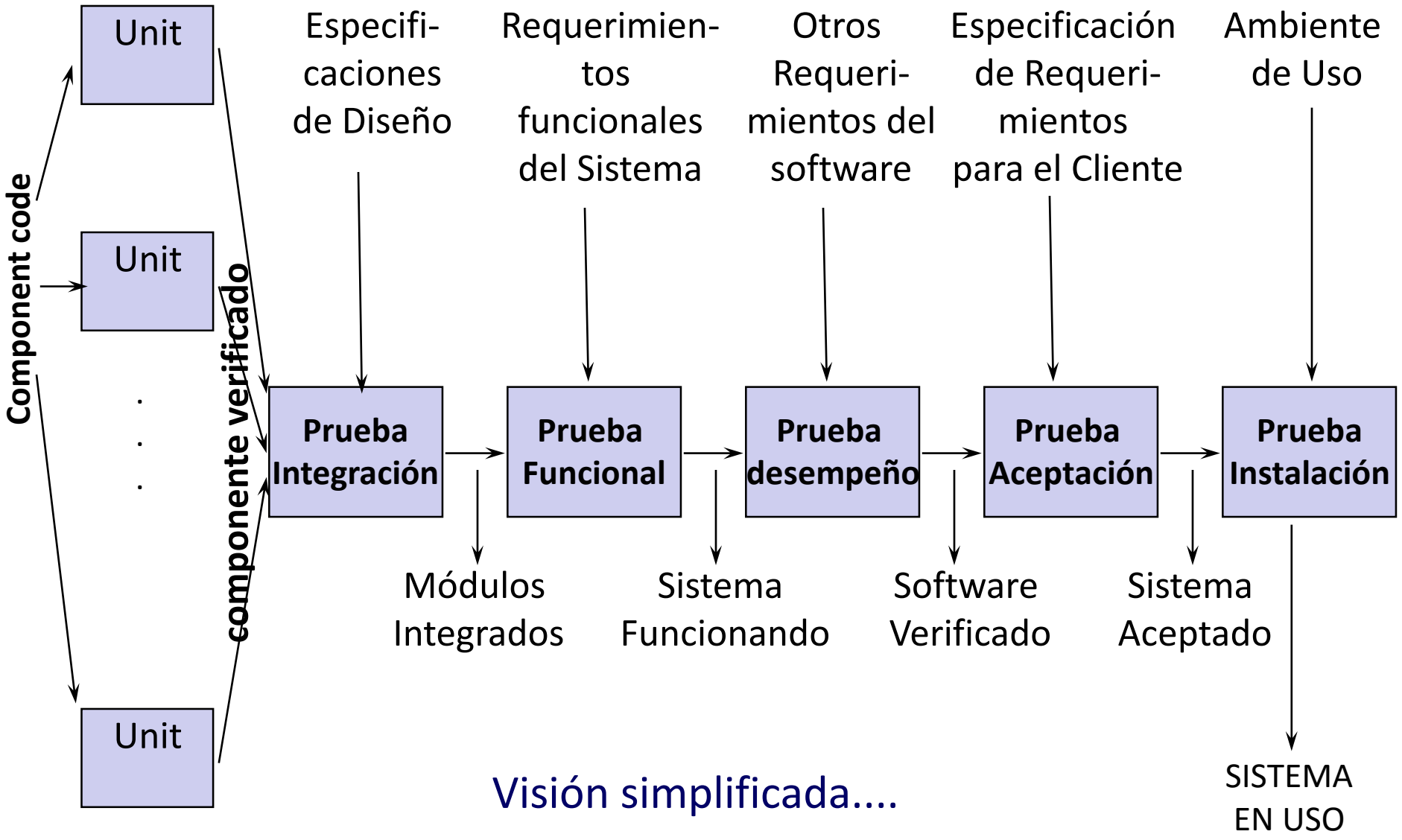
Clasificación de Defectos

- Categorizar y registrar los tipos de defectos
  - Guía para orientar la verificación
    - Si conozco los tipos de defectos en que incurre la organización me puedo ocupar de buscarlos expresamente
  - Mejorar el proceso
    - Si tengo identificada la fase en la cual se introducen muchos defectos me ocupo de mejorarla

- Clasificación Ortogonal
  - Cada defecto queda en una única categoría
  - Si pudiera quedar en más de una de poco serviría

Proceso de V&V

Proceso



Visión simplificada....

Pruebas en el Proceso

- Módulo, Componente o unitaria
  - Verifica las funciones de los componentes
- Integración
  - Verifica que los componentes trabajan juntos como un sistema integrado
- Funcional
  - Determina si el sistema integrado cumple las funciones de acuerdo a los requerimientos

Pruebas en el Proceso

- Desempeño
  - Determina si el sistema integrado, en el ambiente objetivo cumple los requerimientos de tiempo de respuesta, capacidad de proceso y volúmenes
- Aceptación
  - Bajo la supervisión del cliente, verificar si el sistema cumple con los requerimientos del cliente (y lo satisface)
  - Validación del sistema
- Instalación
  - El sistema queda instalado en el ambiente de trabajo del cliente y funciona correctamente

¿Quién Verifica?

- Pruebas Unitarias
  - Normalmente las realiza el equipo de desarrollo. En general la misma persona que lo implementó.
  - Es positivo el conocimiento detallado del módulo a probar
- Pruebas de Integración
  - Normalmente las realiza el equipo de desarrollo
  - Es necesario el conocimiento de las interfaces y funciones en general
- Resto de las pruebas
  - En general un equipo especializado (verificadores)
  - Es necesario conocer los requerimientos y tener una visión global

¿Quién Verifica?

- ¿Por qué un equipo especializado?
  - Maneja mejor las técnicas de pruebas
  - Conoce los errores más comunes realizados por el equipo de programadores
  - Problemas de psicología de pruebas
    - El autor de un programa tiende a cometer los mismos errores al probarlo
    - Debido a que es “SU” programa inconcientemente tiende a hacer casos de prueba que no hagan fallar al mismo
    - Puede llegar a comparar mal el resultado esperado con el resultado obtenido debido al deseo de que el programa pase las pruebas

Verificación Unitaria

Técnicas de Verificación Unitaria

- Técnicas estáticas (analíticas)
  - Analizar el producto para deducir su correcta operación
- Técnicas dinámicas (pruebas)
  - Experimentar con el comportamiento de un producto para ver si el producto actúa como es esperado
- Ejecución simbólica
  - Técnica híbrida

Técnicas Estáticas

- Análisis de código
  - Se revisa el código buscando defectos
  - Se puede llevar a cabo en grupos
  - Recorridas e Inspecciones (técnicas conocidas con resultados conocidos)
- Análisis automatizado de código fuente
  - La entrada es el código fuente del programa y la salida es una serie de defectos detectados
- Verificación formal
  - Se parte de una especificación formal y se busca probar (demostrar) que el programa cumple con la misma

Análisis de Código

- Se revisa el código buscando problemas en algoritmos y otras faltas
- Algunas técnicas:
  - Revisión de escritorio
  - Recorridas e Inspecciones
    - Criticar al producto y no a la persona
    - Permiten
      - Unificar el estilo de programación
      - Igualar hacia arriba la forma de programar
    - No deben usarse para evaluar a los programadores

Análisis de Código

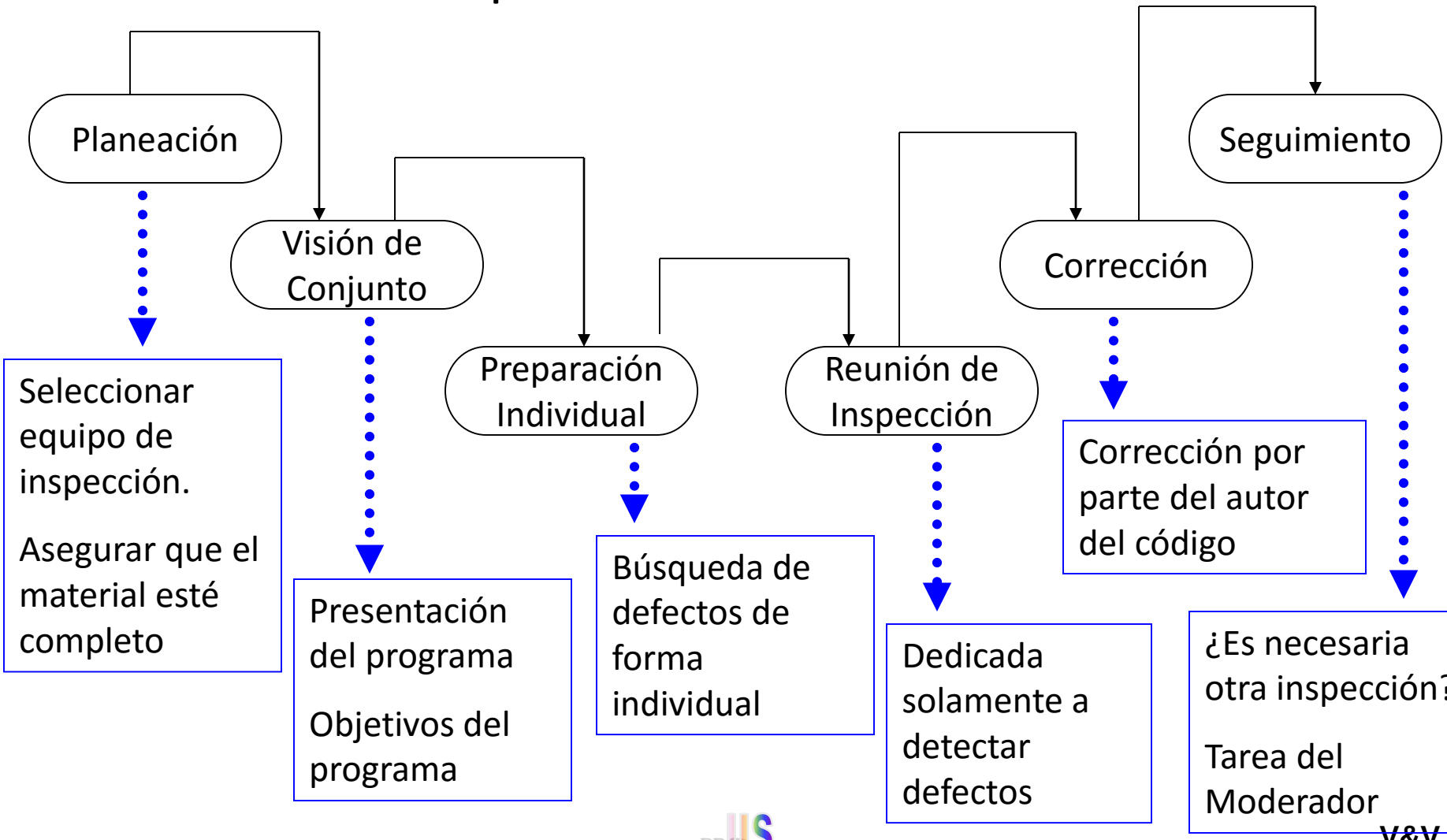
- Recorridas
  - Se simula la ejecución de código para descubrir faltas
  - Número reducido de personas
  - Los participantes reciben antes el código fuente
  - Las reuniones no duran más de dos horas
  - El foco está en detectar faltas y no en corregirlas
  - Los roles clave son:
    - Autor: Presenta y explica el código
    - Moderador: Organiza la discusión
    - Secretario: Escribe el reporte de la reunión para entregarle al autor
  - El autor es el que se encarga de la “ejecución” del código

Análisis de Código

- Inspecciones
  - Se examina el código (no solo aplicables al código) buscando faltas comunes
  - Se usa una lista de faltas comunes (check-list). Estas listas dependen del lenguaje de programación y de la organización. Por ejemplo revisan:
    - Uso de variables no inicializadas
    - Asignaciones de tipos no compatibles
  - Los roles son: Moderador o Encargado de la Inspección, Secretario, Lector, Inspector y Autor
  - Todos son Inspectores. Es común que una persona tenga más de un rol
  - Algunos estudios indican que el rol del Lector no es necesario

Análisis de Código

- Proceso de la Inspección



Análisis de Código

- En un estudio de Fagan:
  - Con Inspección se detectó el 67% de las faltas detectadas
  - Al usar Inspección de Código se tuvieron 38% menos fallas (durante los primeros 7 meses de operación) que usando recorridas
- Ackerman et. al.:
  - reportaron que 93% de todas las faltas en aplicación de negocios fueron detectadas a partir de inspecciones
- Jones:
  - reportó que inspecciones de código permitieron detectar 85% del total de faltas detectadas

Análisis Automatizado

- Herramientas de software que recorren código fuente y detectan posibles anomalías y faltas
- No requieren ejecución del código a analizar
- Es mayormente un análisis sintáctico:
  - Instrucciones bien formadas
  - Inferencias sobre el flujo de control
- Complementan al compilador

Análisis Formal

- Un programa es correcto si cumple con la especificación
- Se busca demostrar que el programa es correcto a partir de una *especificación formal*
- Objeciones
  - Demostración más larga (y compleja) que el propio programa
  - La demostración puede ser incorrecta
  - Demasiada matemática para el programador medio
  - No se consideran limitaciones del hardware
  - La especificación puede ser incorrecta

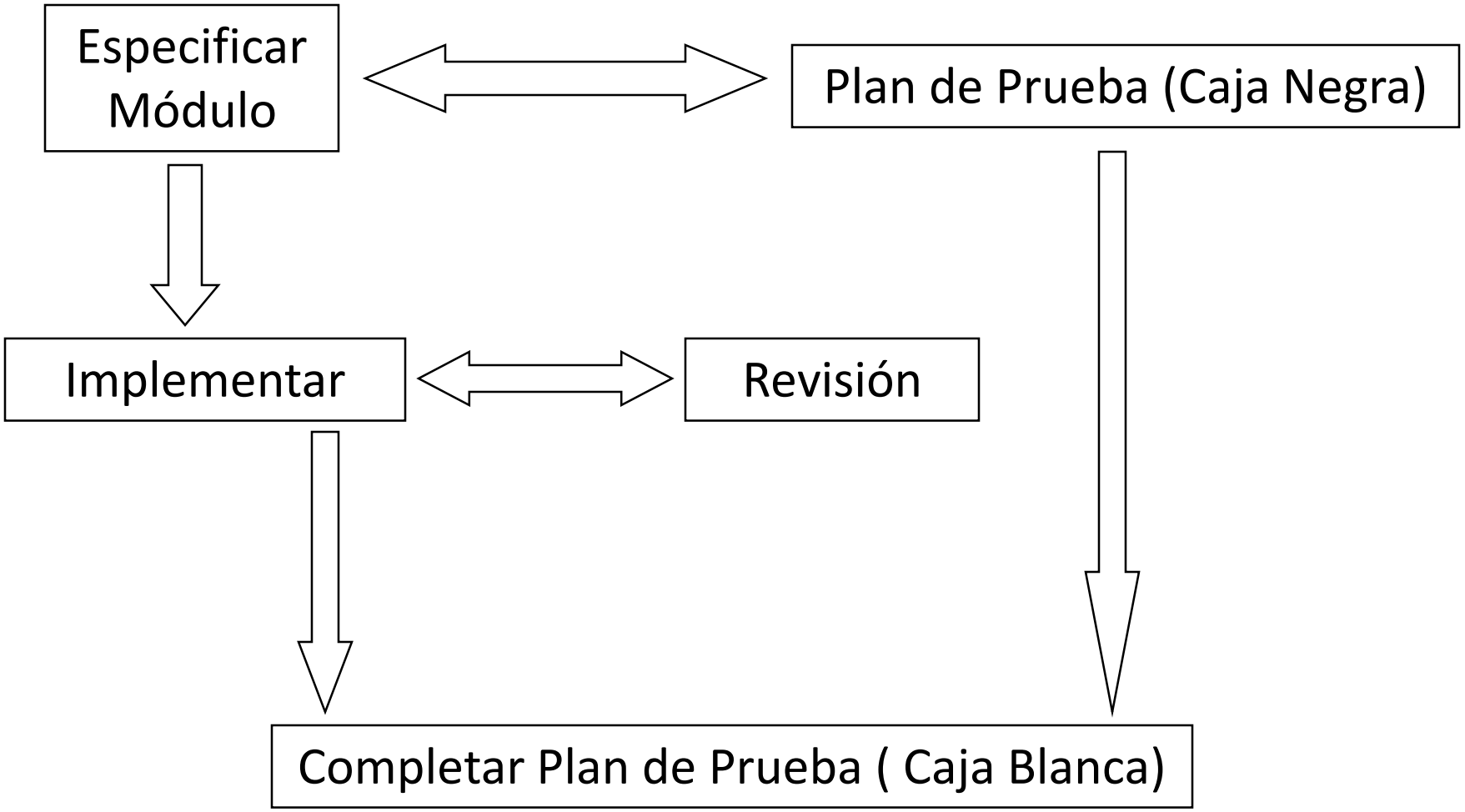
Algunas Definiciones

- Prueba (test)
  - Proceso de ejecutar un programa con el fin de encontrar fallas (G. Myers)
  - Ejecutar un producto para:
    - Verificar que satisface los requerimientos
    - Identificar diferencias entre el comportamiento real y el esperado (IEEE)
- Caso de Prueba (test case)
  - Datos de entrada, condiciones de ejecución y resultado esperado (RUP)
- Conjunto de Prueba (test set)
  - Conjunto de casos de prueba

Visión de los Objetos a Probar

- Caja Negra
  - Entrada a una caja negra de la que no se conoce el contenido y ver que salida genera
    - Casos de prueba – No se precisa disponer del código
    - Se parte de los requerimientos y/o especificación
    - Porciones enteras de código pueden quedar sin ejercitar
- Caja Blanca
  - A partir del código identificar los casos de prueba interesantes
    - Casos de prueba – Se necesita disponer del código
    - Tiene en cuenta las características de la implementación
    - Puede llevar a soslayar algún requerimiento no implementado

Un Proceso para un Módulo



Conceptos Básicos

- Es imposible realizar pruebas exhaustivas y probar todas las posibles secuencias de ejecución

- ***La prueba (test) demuestra la presencia de faltas y nunca su ausencia*** (Dijkstra)
- Es necesario elegir un subconjunto de las entradas del programa para testear
  - Conjunto de prueba (test set)

Conceptos Básicos

- El test debe ayudar a localizar faltas y no solo a detectar su presencia
- El test debe ser repetible
  - En programas concurrentes es difícil de lograr
- ¿Cómo se hacen las pruebas?
  - Se ejecuta el caso de prueba y se compara el resultado esperado con el obtenido.

Principios Empíricos

- Se necesitan “estrategias” para seleccionar casos de prueba “significativos”
- Test Set de Significancia
  - Tiene un alto potencial para descubrir errores
  - La ejecución correcta de estos test aumenta la confianza en el producto
- Más que correr una gran cantidad de casos de prueba nuestra meta en las pruebas debe ser correr un **suficiente número de casos de prueba significativos** (tiene alto porcentaje de posibilidades de descubrir un error)

Principios Empíricos

- ¿Como intentamos definir test sets de significancia?
  - Agrupamos elementos del dominio de la entrada en clases D_i, tal que, elementos de la misma clase se comportan (o suponemos se comportan) exactamente de la misma manera (consistencia)
  - De esta manera podemos elegir un único caso de prueba para cada clase

Principios Empíricos

- Si las clases D_i cumplen U D_i = D → El test set satisface el *principio de cubrimiento completo*
- Extremos de criterios
  - Divido las entradas en una única clase
    - No tiene sentido ya que no se provocarán muchas fallas
  - Divido las entradas en tantas clases como casos posibles (exhaustiva)
    - Es imposible de realizar
- Para asegurarnos mas, en clases D_i que tenemos dudas de que sus elementos se comporten de la misma manera tomamos mas de un caso representativo

¿Qué Estamos Buscando?

¿Cuál es el subconjunto de todos los posibles casos de prueba que tiene la mayor probabilidad de detectar el mayor número posible de errores dadas las limitaciones de tiempo, costo, tiempo de computador, etc?

Caja Blanca

- Tipos de técnicas de caja blanca
  - **Basadas en el flujo de control del programa**
    - Expresan los cubrimientos del testing en términos del grafo de flujo de control del programa
  - **Basadas en el flujo de datos del programa**
    - Expresan los cubrimientos del testing en términos de las asociaciones definición-uso del programa

Caja Blanca

- Criterio de cubrimiento de sentencias
  - Asegura que el conjunto de casos de pruebas (CCP) ejecuta al menos una vez cada instrucción del código

```
If (a > 1) and (b = 0) {
    x = x / a
}
If (a = 2) or (x > 1) {
   x = x + 1
}
```

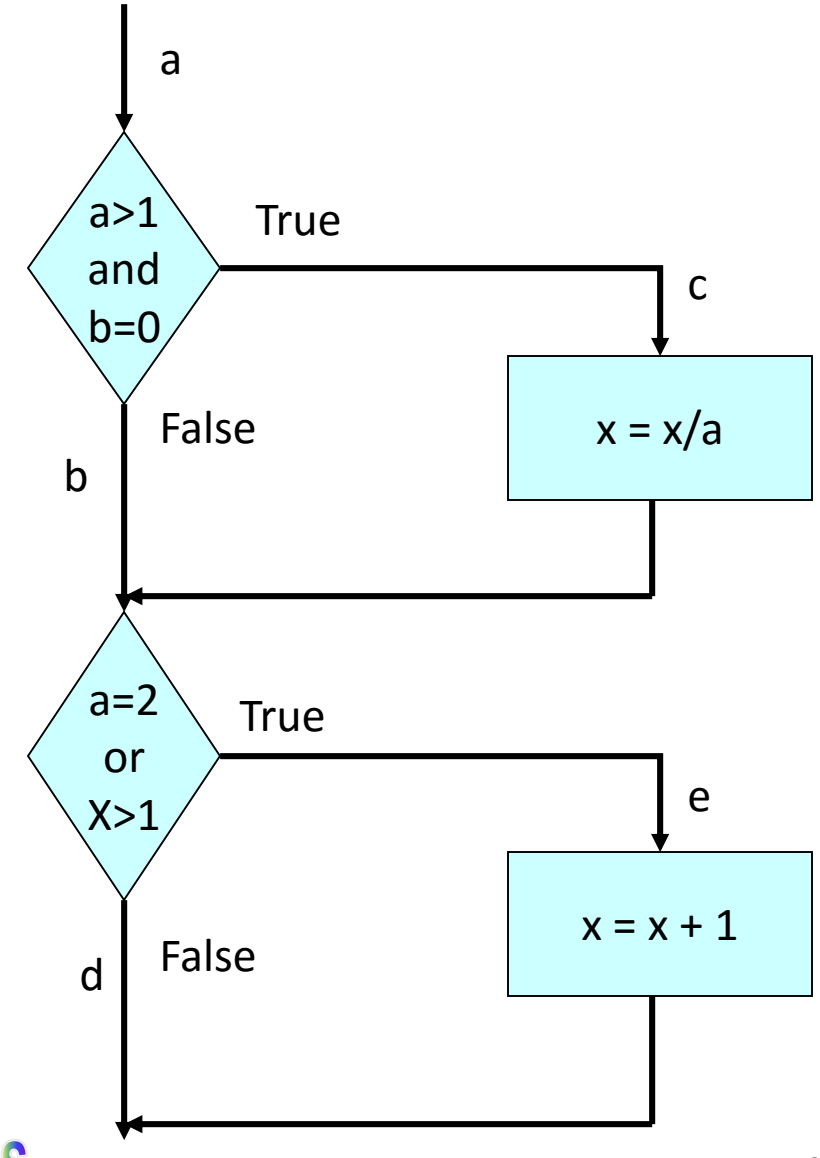
CP : Entrada a=2, b=0, x=3

- ¿Que pasa si tenía un or en lugar del and de la primera decisión?
- ¿Que pasa si tenía x > 0 en lugar de x > 1 en la segunda decisión?
- Hay una secuencia en la cual x se mantiene sin cambios y esta no es probada

Este criterio es tan débil que normalmente se lo considera inútil. Es necesario pero no suficiente (Myers)

Caja Blanca

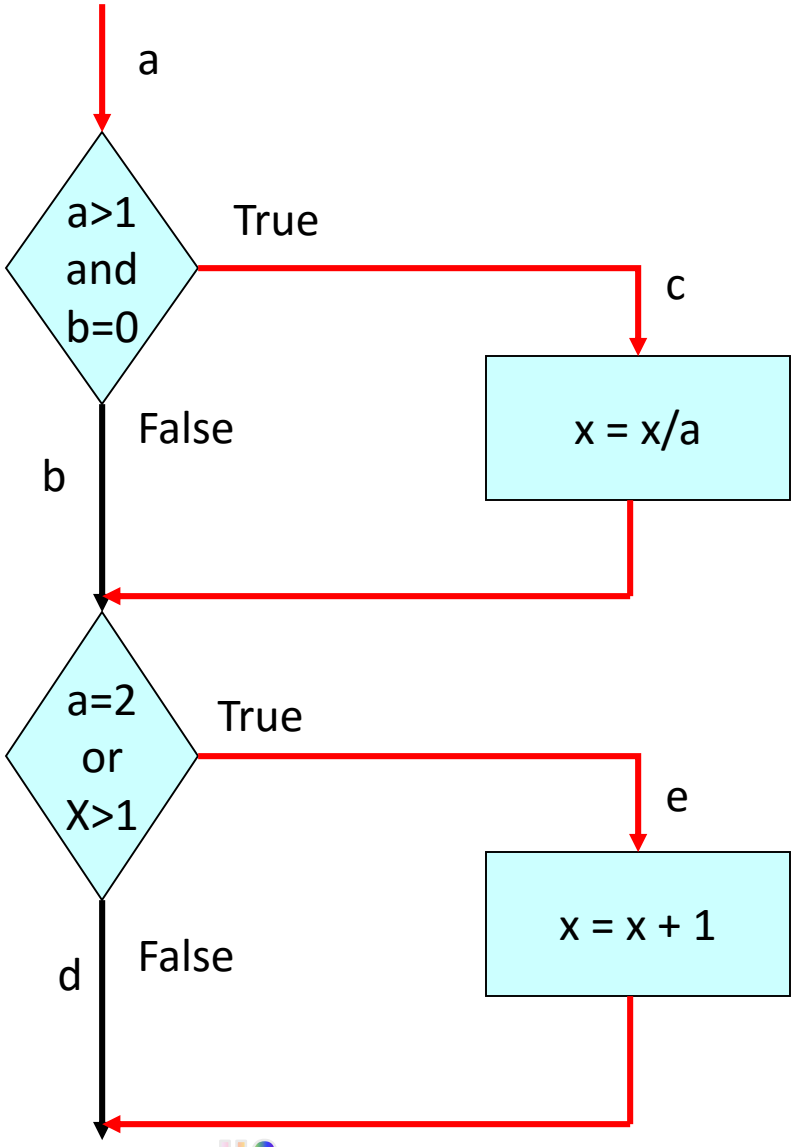
```
If (a > 1) and (b = 0) {
    x = x / a
}
If (a = 2) or (x > 1) {
   x = x + 1
}
```



Caja Blanca

CP a = 2, b = 0, x = 3

Secuencia: ace



Caja Blanca

- Criterio de cubrimiento de decisión
  - Cada decisión dentro del código toma al menos una vez el valor true y otra vez el valor false para el CCP

```
If (a > 1) and (b = 0) {
    x = x / a
}
If (a = 2) or (x > 1) {
   x = x + 1
}
```

CP1 a=3, b=0, x=3

CP2 a=2, b=1, x=1

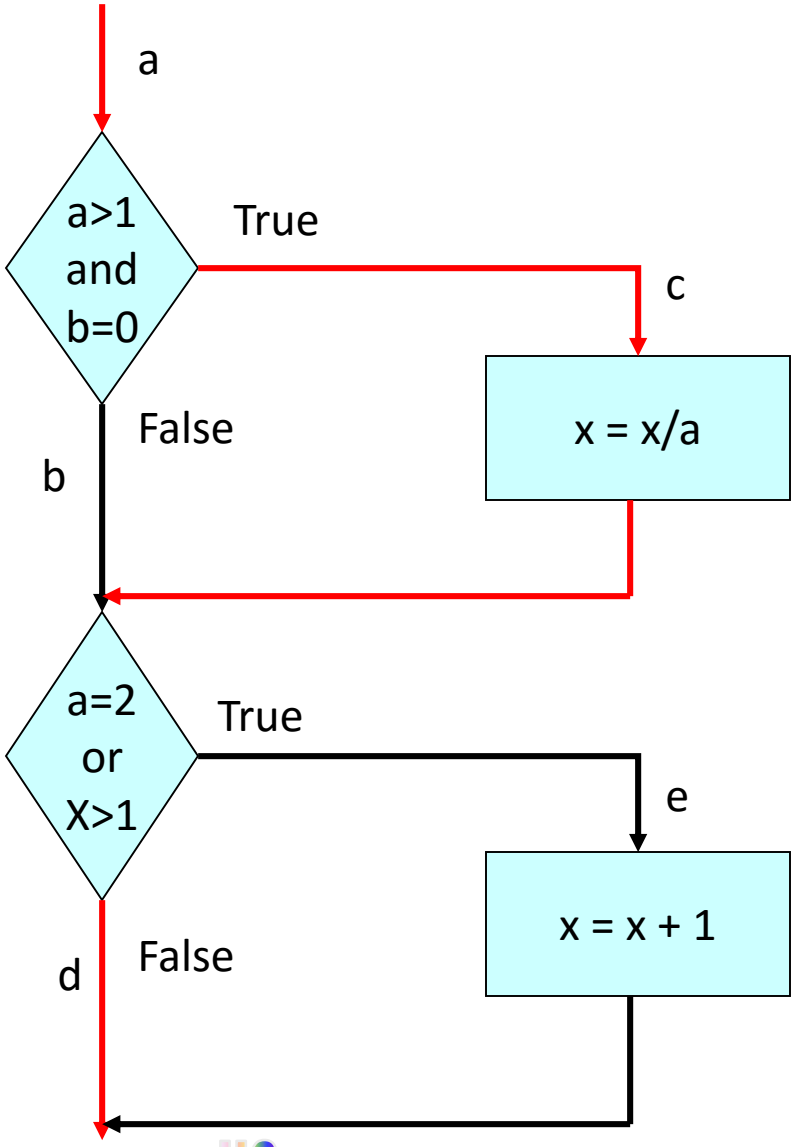
- ¿Qué pasa si en la segunda decisión tuviera x < 1 en lugar de x > 1?
- Hay una secuencia en la cual x se mantiene sin cambios y esta no es probada
- Hay que extender el criterio para sentencias del tipo CASE

Es más fino que el criterio de sentencias

Caja Blanca

CP1 a = 3, b = 0, x = 3

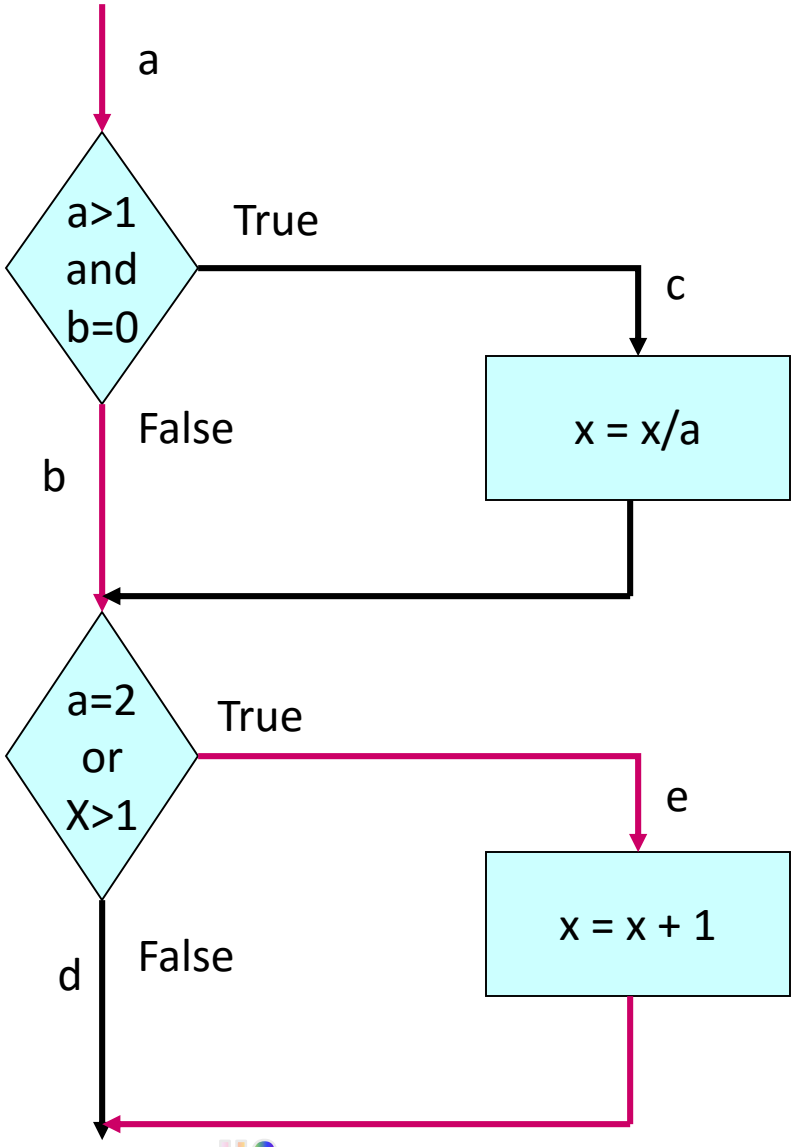
Secuencia: acd



Caja Blanca

CP2 a = 2, b = 1, x = 1

Secuencia: abe



Caja Blanca

- Criterio de cubrimiento de condición
  - Cada condición dentro de una decisión debe tomar al menos una vez el valor true y otra el false para el CCP

a

If (a > 1) and (b = 0) {x = x / a}

b

If (a = 2) or (x > 1) {x = x + 1}

Hay que tener casos tal que a>1, a<=1, b=0 y b<>0 en el punto a y casos en los cuales a=2, a<>2, x>1 y x<=1 en el punto b

CP1 a=2, b=0, x=4

CP2 a=1, b=1, x=1

- Si se tiene If (A and B) …. el criterio de condición se puede satisfacer con estos dos casos de prueba:
  - C1: A=true B=false
  - C2: A=false B=true

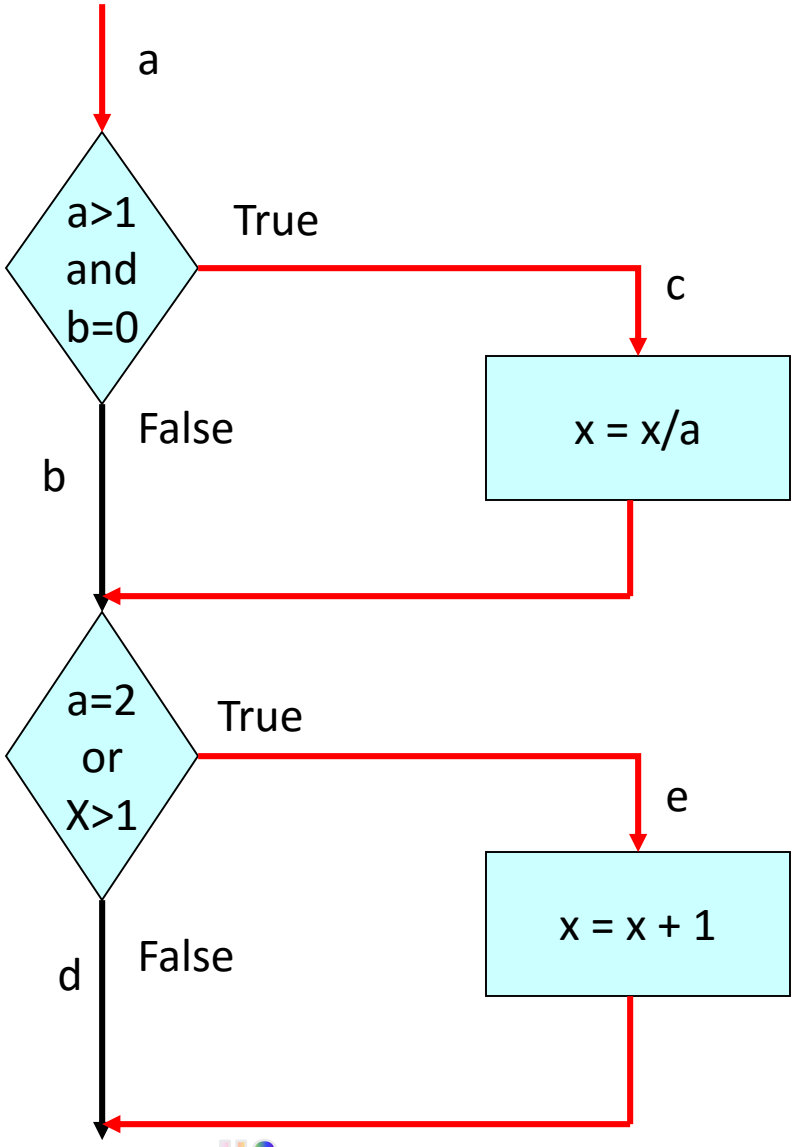
Esto **no** satisface el criterio de decisión

Este criterio es generalmente más fino que el de decisión

Caja Blanca

CP1 a = 2, b = 0, x = 4

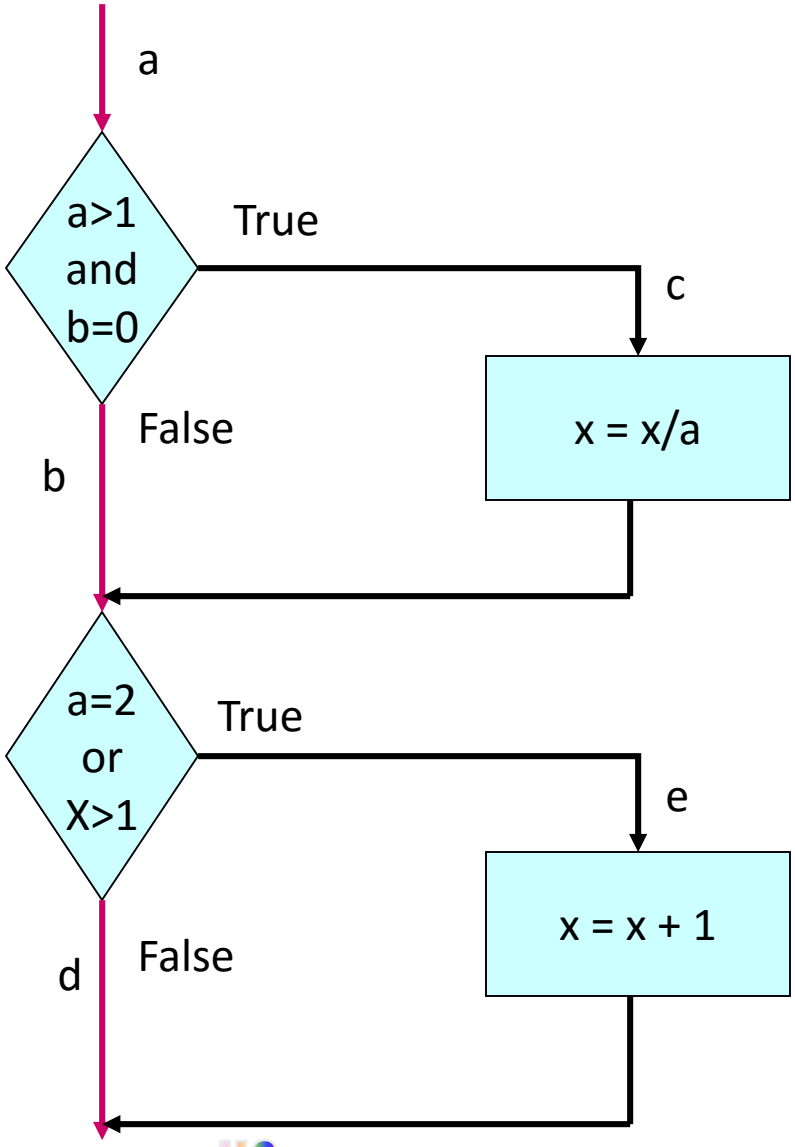
Secuencia: ace



Caja Blanca

CP1 a = 1, b = 1, x = 1

Secuencia: abd



Caja Blanca

- Criterio de cubrimiento de decisión/condición
  - Combinación de los dos criterios anteriores

a

If (a > 1) and (b = 0) {x = x / a}

b

If (a = 2) or (x > 1) {x = x + 1}

CP1 a=2, b=0, x=4

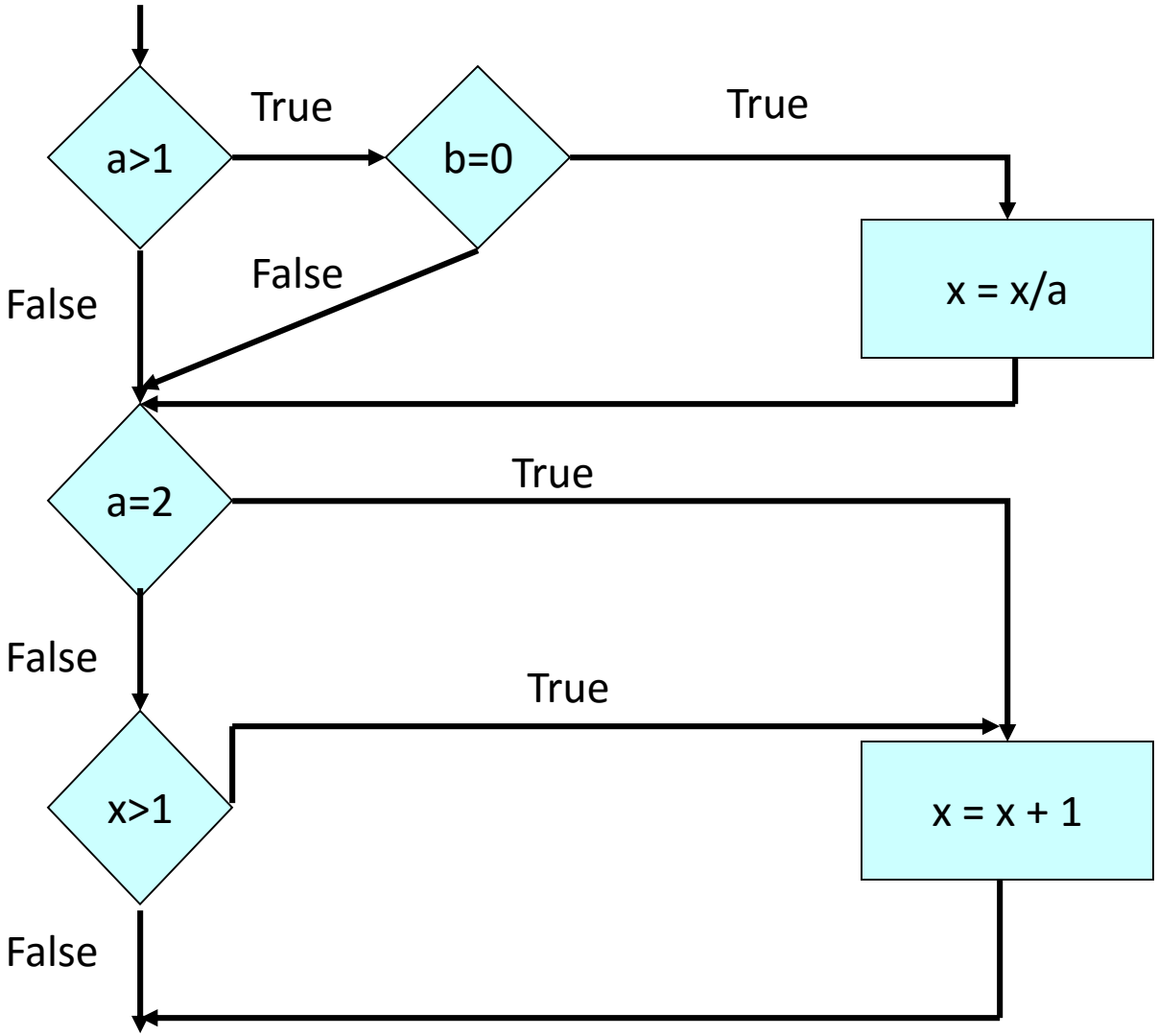
CP2 a=1, b=1, x=1

- Este criterio no tiene porque invocar a todas las salidas producidas por el código máquina → No todas las faltas debido a errores en expresiones lógicas son detectadas

Es un criterio bastante fino dentro de caja blanca (Myers)

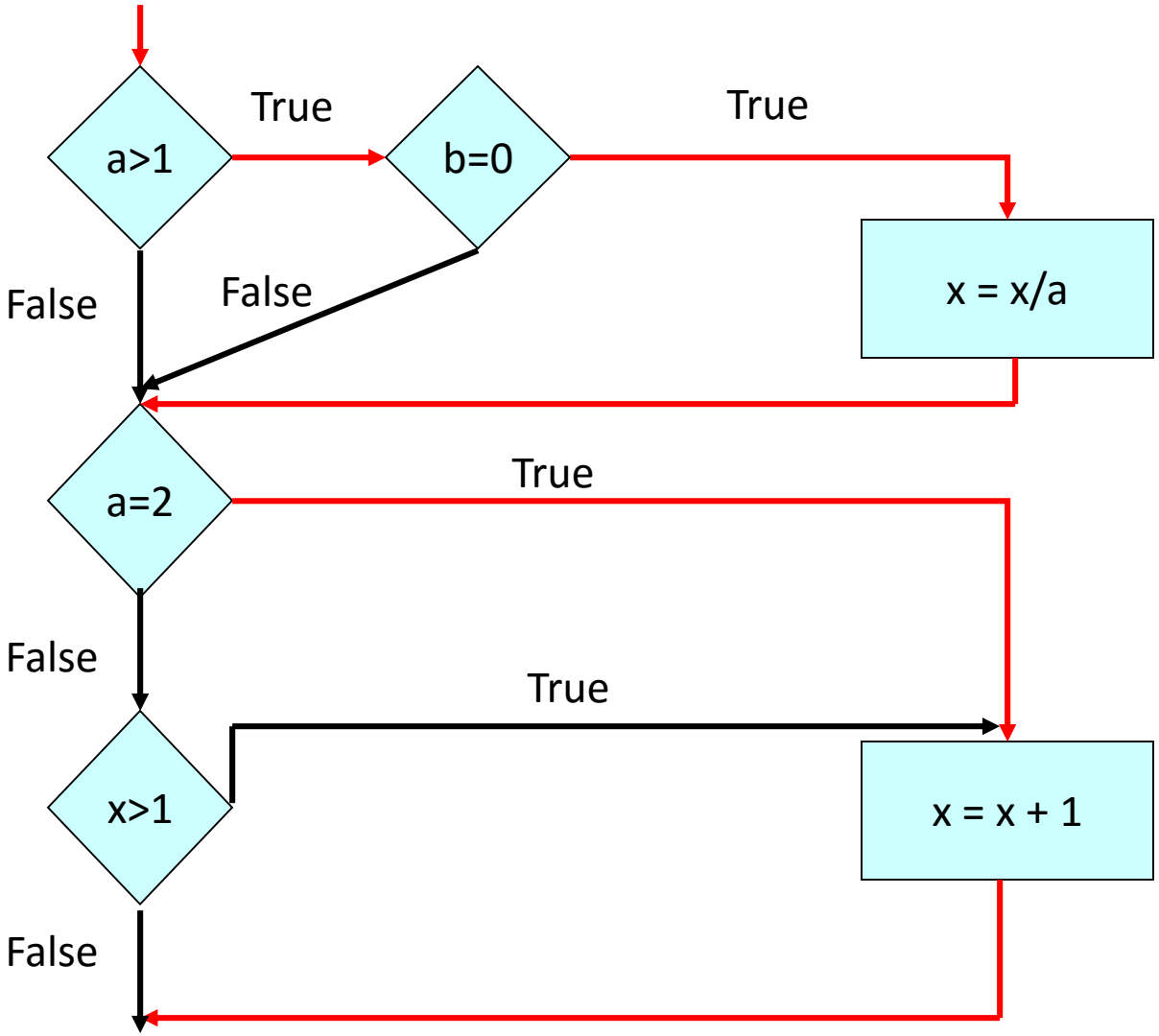
Caja Blanca

```
If (a > 1) {
   If (b > 0)
      {x = x / a}
}
If (a = 2)
   goto Sum
If (x > 1)
   goto Sum
goto Fin
Sum: x = x + 1
Fin:
```



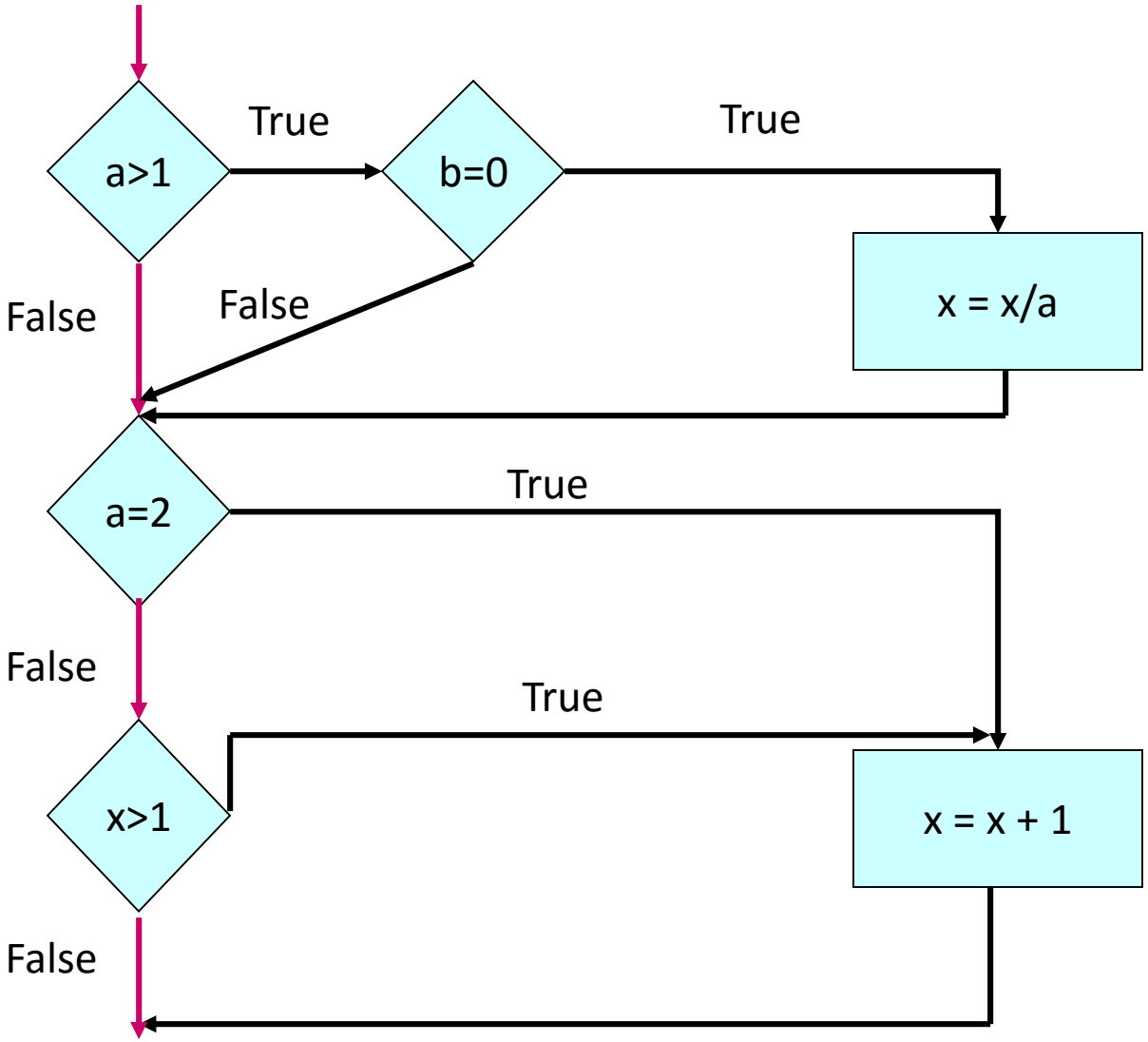
Caja Blanca

CP1 a = 2, b = 0, x = 4



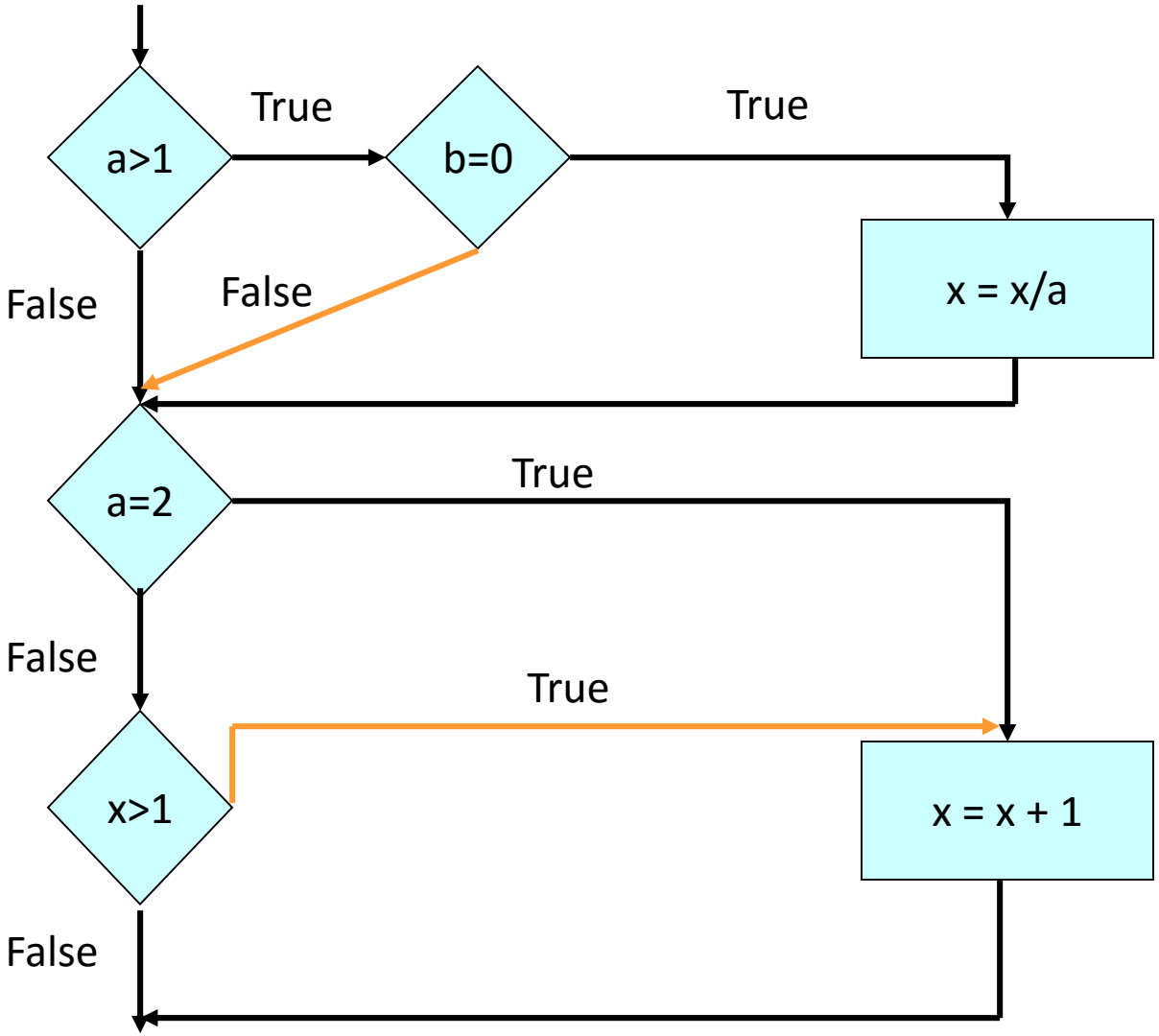
Caja Blanca

CP1 a = 1, b = 1, x = 1



Caja Blanca

FALTO TESTEAR



Caja Blanca

- Criterio de cubrimiento de condición múltiple
  - Todas las combinaciones posibles de resultados de condición dentro de una decisión se ejecuten al menos una vez

Se deben satisfacer 8 combinaciones:

1) a>1, b=0 2) a>1, b<>0

3) a<=1, b=0 4) a<=1, b<>0

5) a=2, x>1 6) a=2, x<=1

7) a<>2, x>1 8) a<>2, x<=1

Los casos 5 a 8 aplican en el punto b

CP1 a=2, b=0, x=4 cubre 1 y 5

CP2 a=2, b=1, x=1 cubre 2 y 6

CP3 a=1, b=0, x=2 cubre 3 y 7

CP4 a=1, b=1, x=1 cubre 4 y 8

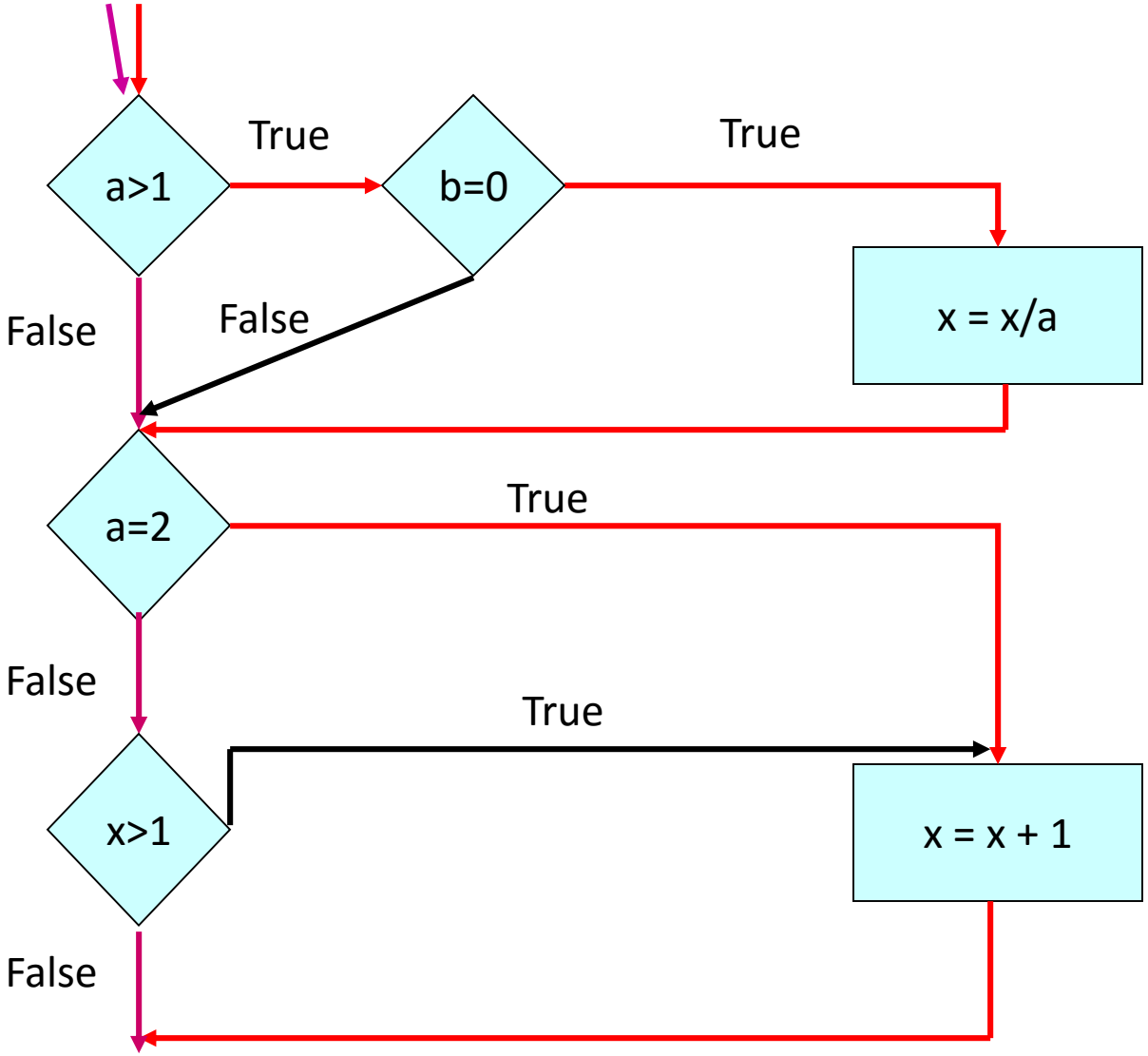
- La secuencia acd queda sin ejecutar → este criterio no asegura testear todos los caminos posibles

Incluye al criterio de condición/decisión.

Myers lo considera un criterio aceptable

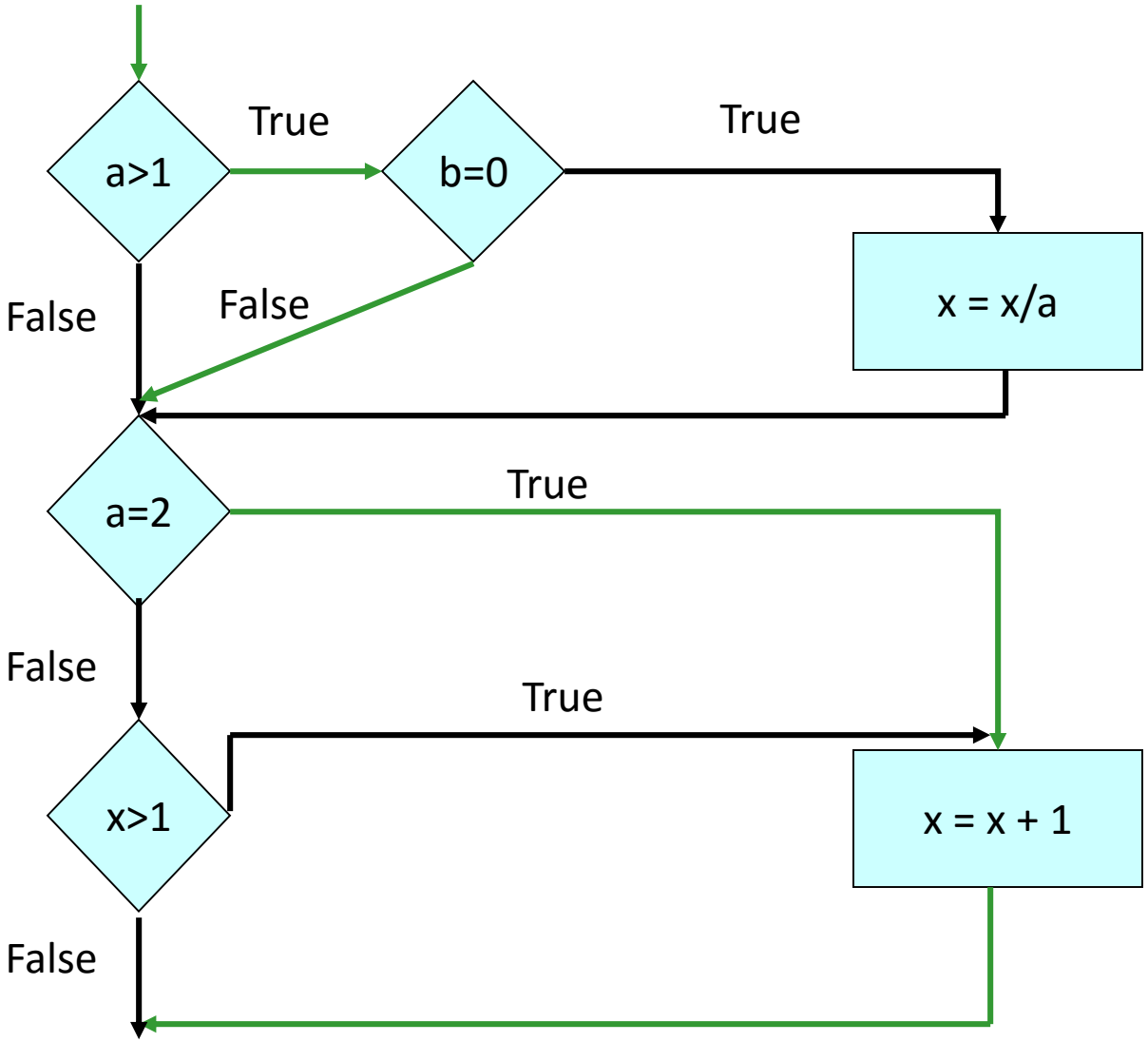
Caja Blanca

CP1 y CP4 ya los vimos



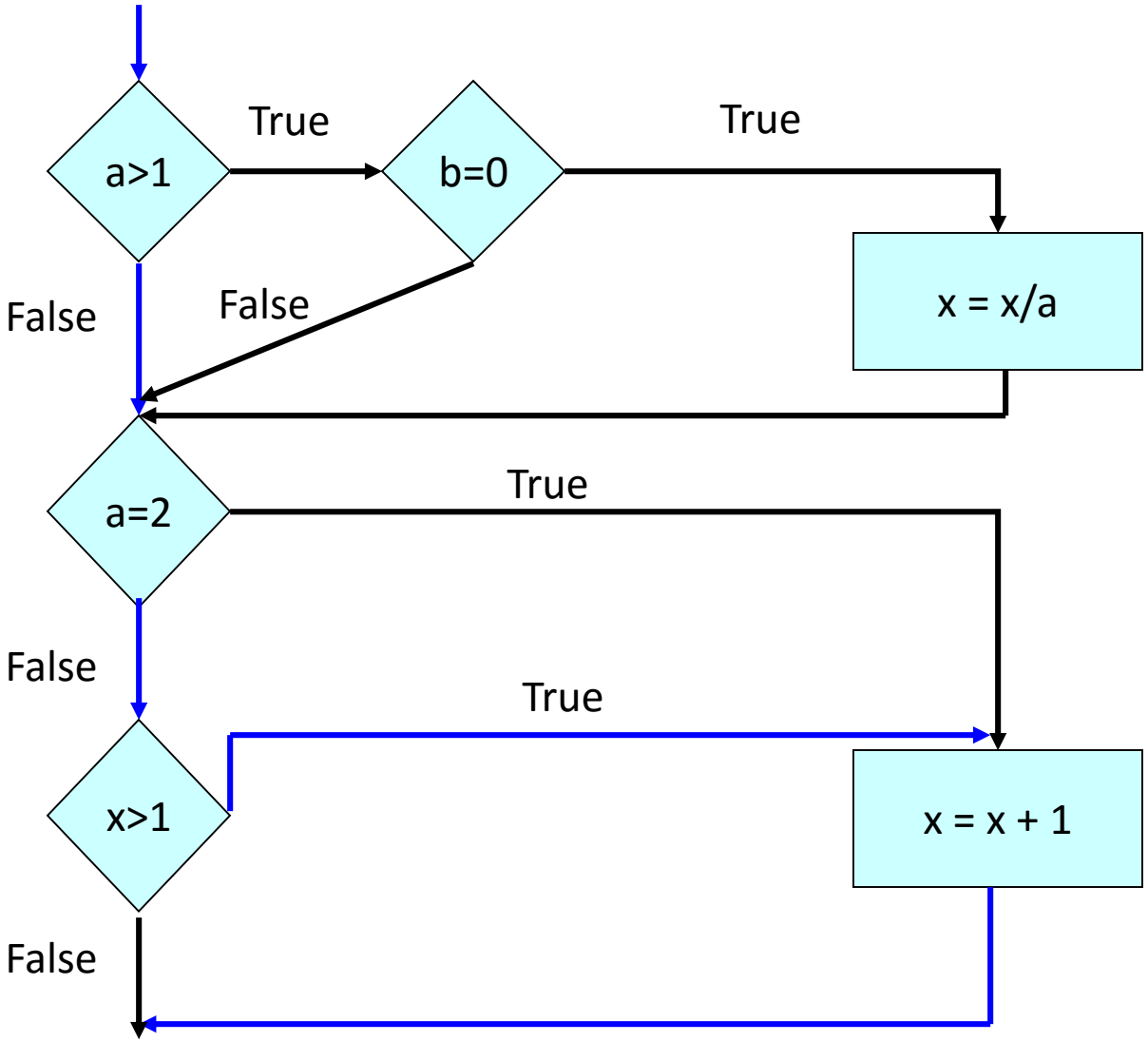
Caja Blanca

CP2 a = 2, b = 1, x = 1



Caja Blanca

CP3 a = 1, b = 0, x = 2



Caja Blanca

- Falta no detectada con el CCCM

```
If x <> 0
    y = 5
else
    z = z – x
If z > 1
    z = z / x
else
    z = 0
```

- El siguiente conjunto de casos de prueba cumple con el criterio de condición múltiple
  - C1 x=0, z=1
  - C2 x=1 z=3
- Este CCP no controla una posible división entre cero. Por ejemplo:
  - C3 x=0 z=3
- El criterio no asegura recorrer todos los caminos posibles del programa. Como el caso acd del ejemplo anterior
- En el ejemplo puede haber una división por cero no detectada

Caja Blanca

- Criterio de cubrimiento de arcos
  - Se “pasa” al menos una vez por cada arco del grafo de flujo de control del programa

Caja Blanca

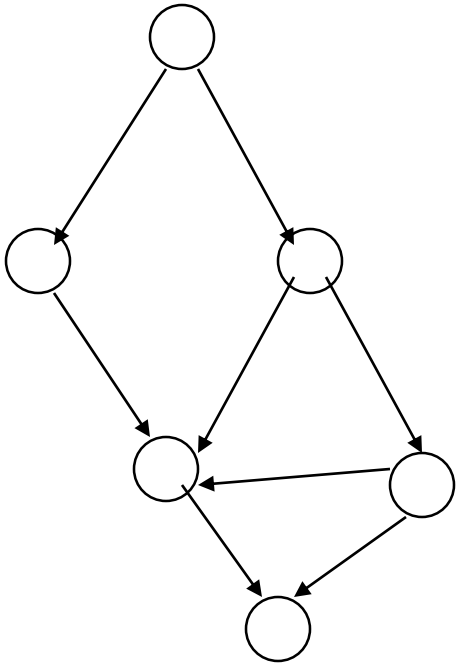
- Criterio de cubrimiento de trayectorias indep.
  - Ejecutar al menos una vez cada trayectoria independiente

El número de trayectorias independientes se calcula usando la complejidad ciclomática

CC = Arcos – Nodos + 2

El CC da el número mínimo de casos de prueba necesarios para probar todas las trayectorias independientes y un número máximo de casos necesarios para cubrimiento de arcos

En el ejemplo tengo que tener 4 casos de prueba para cumplir con el criterio de cubrimiento de trayectoria

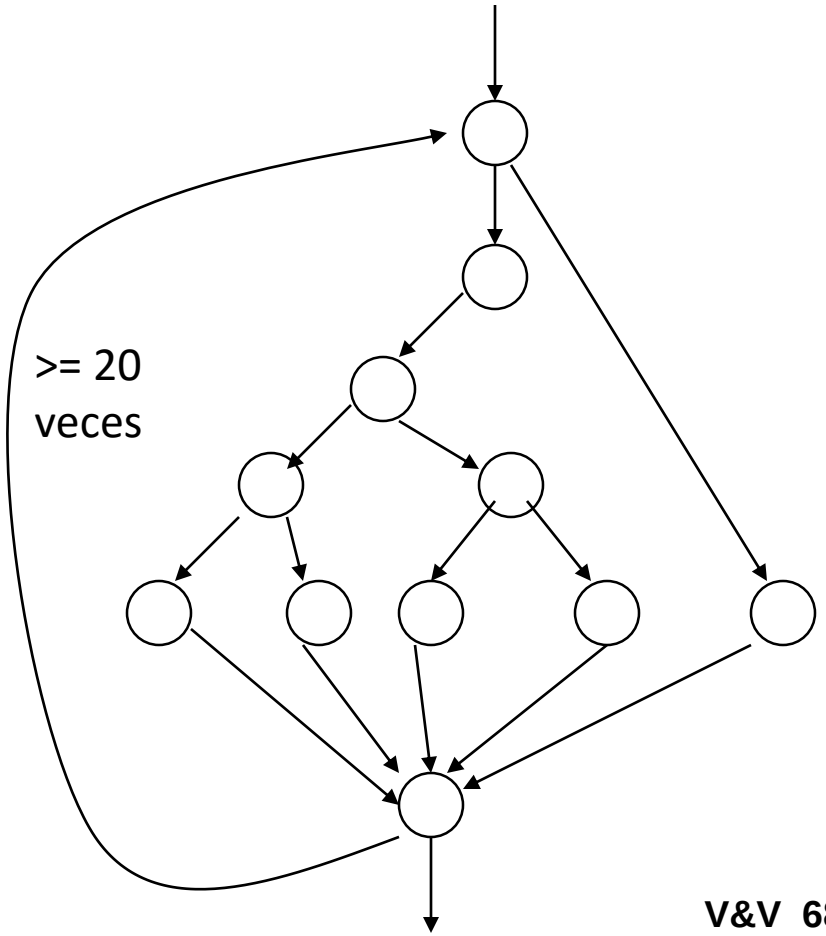


La cantidad de casos de prueba suele ser demasiado grande. Es un criterio de referencia

Caja Blanca

- Criterio de cubrimiento de caminos
  - Se ejecutan al menos una vez todos los caminos posibles (combinaciones de trayectorias)

- En el ejemplo hay 100 trillones de caminos posibles.
- Si determinar cada dato de prueba, el resultado esperado, ejecutar el caso y verificar si es correcto lleva 5 minutos → la tarea llevará aproximadamente un billon de años



Caja Negra

- Las pruebas se derivan solo de la especificación. Solo interesa la funcionalidad y no su implementación
- El “probador” introduce las entradas en los componentes y examina las salidas correspondientes
- Si las salidas no son las previstas probablemente se detecto una falla en el software
- Problema clave = Seleccionar entradas con alta probabilidad de hacer fallar al software (experiencia y algo más…)

Caja Negra

- El programa lee tres números enteros, los que son interpretados como representaciones de las longitudes de los lados de un triángulo. El programa escribe un mensaje que informa si el triángulo es escaleno, isósceles o equilátero.

- Escriban casos de prueba para esta especificación.

Caja Negra

- Triángulo escaleno válido
- Triángulo equilátero válido
- Triángulo isósceles
- 3 permutaciones de triángulos isósceles 3,3,4 –3,4,3 – 4,3,3
- Un caso con un lado con valor nulo
- Un caso con un lado con valor negativo
- Un caso con 3 enteros mayores que 0 tal que la suma de 2 es igual a la del 3 (esto no es un triángulo válido)
- Las permutaciones del anterior
- Suma de dos lados menor que la del tercero (todos enteros positivos)
- Permutaciones
- Todos los lados iguales a cero
- Valores no enteros
- Numero erróneo de valores (2 enteros en lugar de 3)

¿Todos los casos tienen especificada la salida esperada?

Caja Negra

- Myers presenta 4 formas para guiarnos a elegir los casos de prueba usando caja negra
  - Particiones de equivalencia
  - Análisis de valores límites
  - Grafos causa-efecto
  - Conjetura de errores
- Vamos a ver los dos primeros

Caja Negra – Part. Eq.

- Propiedades de un buen caso de prueba
  - Reduce significativamente el número de los otros casos de prueba
  - Cubre un conjunto extenso de otros casos de prueba posibles
- Clase de equivalencia
  - Conjunto de entradas para las cuales suponemos que el software se comporta igual
- Proceso de partición de equivalencia
  - Identificar las clases de equivalencia
  - Definir los casos de prueba

Caja Negra – Part. Eq.

- Identificación de las clases de equivalencia
  - Cada condición de entrada separarla en 2 o más grupos
  - Identificar las *clases válidas* así como también las *clases inválidas*
  - *Ejemplos:*
    - "la numeración es de 1 a 999" → clase válida 1 <= num <= 999, 2 clases inválidas num < 1 y num > 999
    - "el primer carácter debe ser una letra" → clase válida el primer carácter es una letra, clase inválida el primer carácter no es una letra
  - Si se cree que ciertos elementos de una clase de eq. no son tratados de forma idéntica por el programa, dividir la clase de eq. en clases de eq. menores

Caja Negra – Part. Eq.

- Proceso de definición de los casos de prueba
  1. Asignar un número único a cada clase de equivalencia
  2. Hasta cubrir todas las clases de eq. con casos de prueba, escribir un nuevo caso de prueba que cubra tantas clases de eq. válidas, no cubiertas, como sea posible
  3. Escribir un caso de prueba para cubrir una y solo una clase de equivalencia para cada clase de equivalencia inválida (evita cubrimiento de errores por otro error)

Caja Negra – Valores Límite

- La experiencia muestra que los casos de prueba que exploran las *condiciones límite* producen mejor resultado que aquellas que no lo hacen
- Las *condiciones límite* son aquellas que se hallan “arriba” y “debajo” de los márgenes de las clases de equivalencia de entrada y de salida (aplicable a caja blanca)
- Diferencias con partición de equivalencia
  - Elegir casos tal que los márgenes de las clases de eq. sean probados
  - Se debe tener muy en cuenta las clases de eq. de la salida (esto también se puede considerar en particiones de equivalencia)

Caja Negra – Valores Límite

- Ejemplos
  - La entrada son valores entre -1 y 1 → Escribir casos de prueba con entrada 1, -1, 1.001, -1.001
  - Un archivo de entrada puede contener de 1 a 255 registros → Escribir casos de prueba con 1, 255, 0 y 256 registros
  - Se registran hasta 4 mensajes en la cuenta a pagar (UTE, ANTEL, etc) → Escribir casos de prueba que generen 0 y 4 mensajes. Escribir un caso de prueba que pueda causar el registro de 5 mensajes. LÍMITES DE LA SALIDA
  - USAR EL INGENIO PARA ENCONTRAR CONDICIONES LÍMITE

Caja Negra

- En el ejemplo del triángulo detectar:
  - Clases de equivalencia de la entrada
  - Valores límite de la entrada
  - Clases de equivalencia de la salida
  - Valores límite de la salida

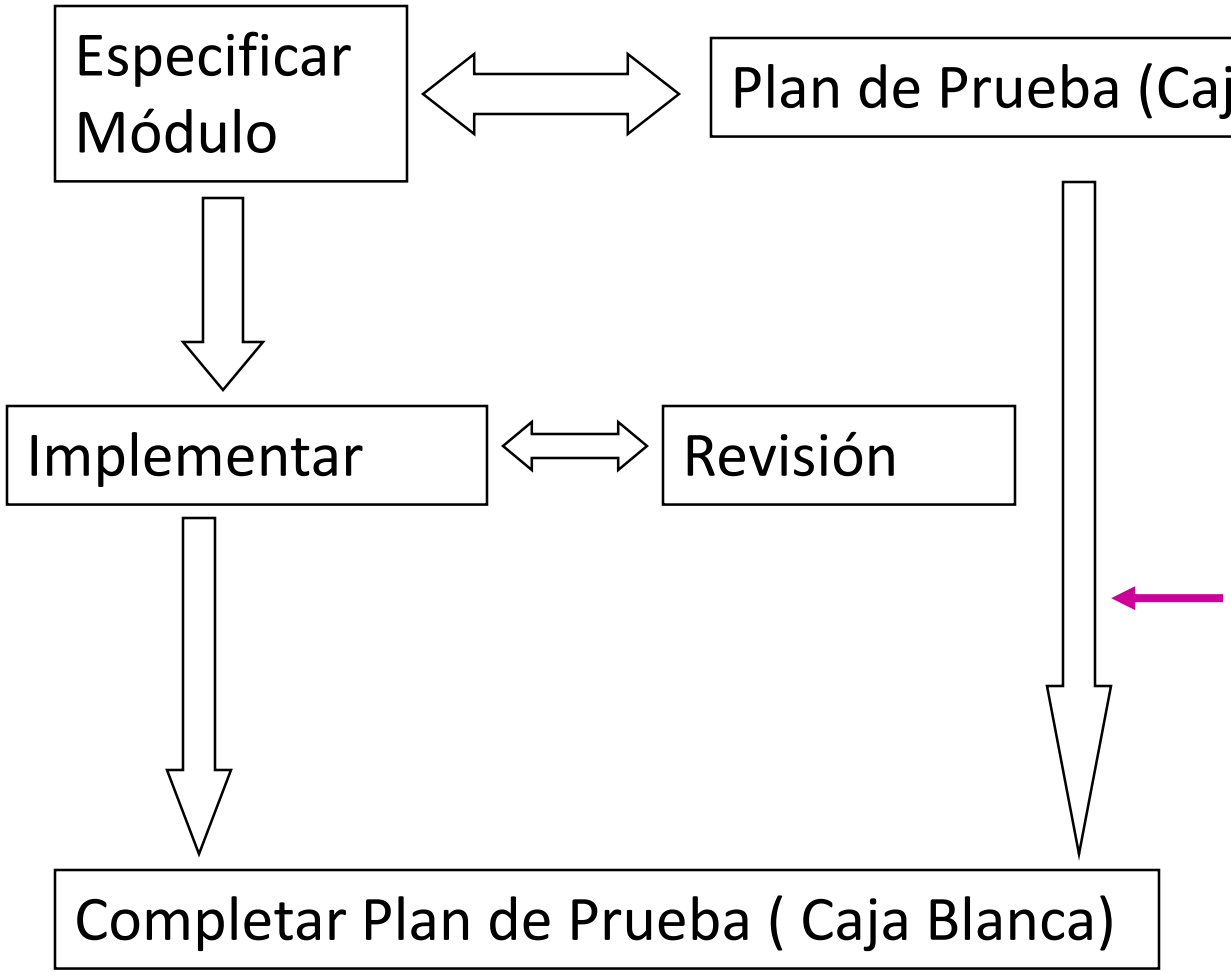
Caja Negra

- Algunas clases de equivalencia de la entrada
  - La suma de dos lados es siempre mayor que la del tercero
  - La suma de dos lados no siempre es mayor que la del tercero
    - Combinación para todos los lados
  - No ingreso todos los lados
- Algunos valores límite de la entrada
  - La suma de dos lados es igual a la del tercero
    - Combinación para todos los lados
  - No ingreso ningún lado

Caja Negra

- Algunas clases de equivalencia de la salida
  - Triángulo escaleno
  - Triángulo isósceles
  - Triangulo equilátero
  - No es un triángulo válido
- Algunos valores límite de la salida
  - Quizás un “triángulo casi válido”
    - Ejemplo: lado1 = 1, lado2 =2, lado3= 3

Un Proceso para un Módulo



Jacobson sugiere ejecutar primero las pruebas de caja negra y cuando estas sean todas correctas completar con caja blanca. Esto se debe a que la caja blanca depende del código y cuando se detecten errores con caja negra el código cambia al corregir los errores detectados.

Comparación de las Técnicas

- Estáticas (análisis)
  - ✓ Efectivas en la detección temprana de defectos
  - ✓ Sirven para verificar cualquier producto (requerimientos, diseño, código, casos de prueba, etc)
  - ✓ Conclusiones de validez general
  - ✗ Sujeto a los errores de nuestro razonamiento
  - ✗ No se usan para validación (solo verificación)
  - ✗ No consideran el hardware o el software de base

Comparación de las Técnicas

- Dinámicas (pruebas)
  - ✓ Se considera el ambiente donde es usado el software (realista)
  - ✓ Sirven tanto para verificar como para validar
  - ✓ Sirven para probar otras características además de funcionalidad
  - ✗ Está atado al contexto donde es ejecutado
  - ✗ Su generalidad no es siempre clara (solo se aseguran los casos probados)
  - ✗ Solo sirve para probar el software construido
  - ✗ Normalmente se detecta un único error por prueba (los errores cubren a otros errores o el programa colapsa)