Verificación y Validación

Verificación y Validación

- Temario
  - Introducción
  - Proceso de V&V
  - Verificación Unitaria
    - Técnicas Estáticas (análisis)
    - Ejecución Simbólica
    - Técnicas Dinámicas (pruebas)
  - Pruebas de Integración
  - Pruebas de Sistemas Orientados a Objetos
  - Pruebas de Sistema
  - Herramientas
  - Planificación de V&V
  - Terminación de la prueba

Introducción

- Temario
  - Errores, faltas y fallas.
  - Objetivos y definición de V&V
  - Ejemplo
  - Tipos de faltas
  - Clasificación de defectos

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
- La idea es detectar y corregir estas faltas antes de liberar el producto

Objetivos de V&V

- Descubrir defectos (para corregirlos)
  - 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
  - Es el proceso de determinar que defecto o defectos causaron la falla
- Corrección de defectos
  - Es el proceso de 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?
  - Validación
    - Busca comprobar que el software hace lo que el usuario espera.
    - ¿El software cumple las expectativas del cliente?

Definición de V&V

- Boehm
  - Verificación
    - ¿Estamos construyendo el producto correctamente?
  - Validación
    - ¿Estamos construyendo el producto correcto?

- Ghezzi
  - Verificación
    - Todas las actividades que son llevadas a cabo para averiguar si el software cumple con sus objetivos

Definición de V&V

- IEEE
  - Verificación
    - The process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of the phase
  - Validación
    - The process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements

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
- Posibles casos a probar:
  - lado1 = 0, lado2 = 1, lado3 = 0 **Resultado** = error
  - lado1 = 2, lado2 = 2, lado3 = 3 **Resultado** = isósceles
- Estos son *Casos de Prueba*

Ejemplo

- Comparo el resultado esperado con el obtenido
  - Si son distintos probablemente haya fallado el programa
- 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

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
    - Bifurcar a destiempo
    - 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 como un truncamiento no esperado
- 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
    - Sistema que funciona bien al principio del día y se va degradando paulatinamente el desempeño hasta ser espantoso al caer la tarde. Falta: había tareas que no liberaban memoria
- De capacidad o de borde
  - Más de lo que el sistema puede manejar
  - Ejemplos
    - El sistema funciona bien con importes <1000000
    - 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

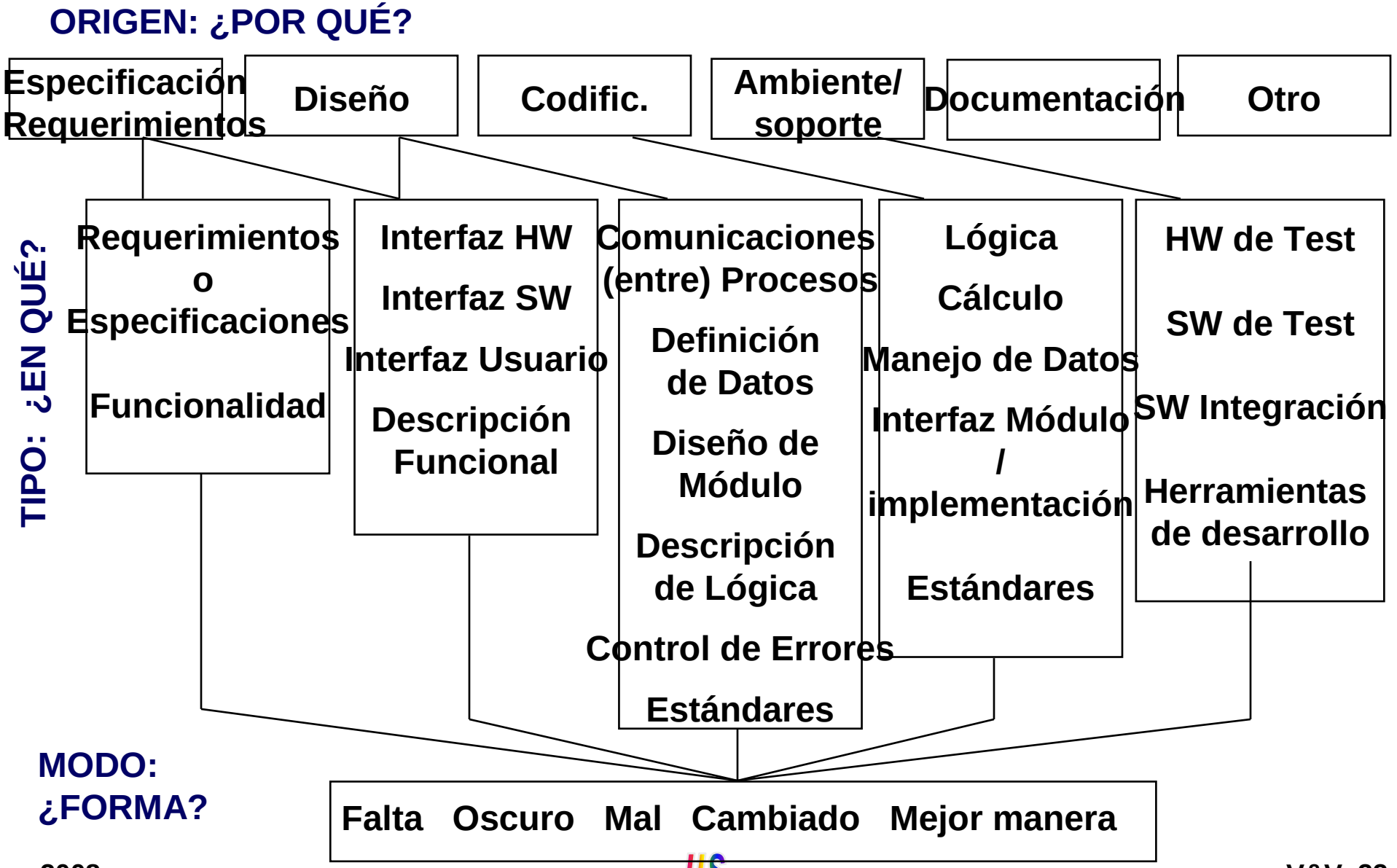
- Categorizar y registrar los tipos de defectos
  - Guía para orientar la verificación
    - Si conozco los tipos de defectos en que incurre la organización me puedo ocupar de buscarlos expresamente
  - Mejorar el proceso
    - Si tengo identificada la fase en la cual se introducen muchos defectos me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

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

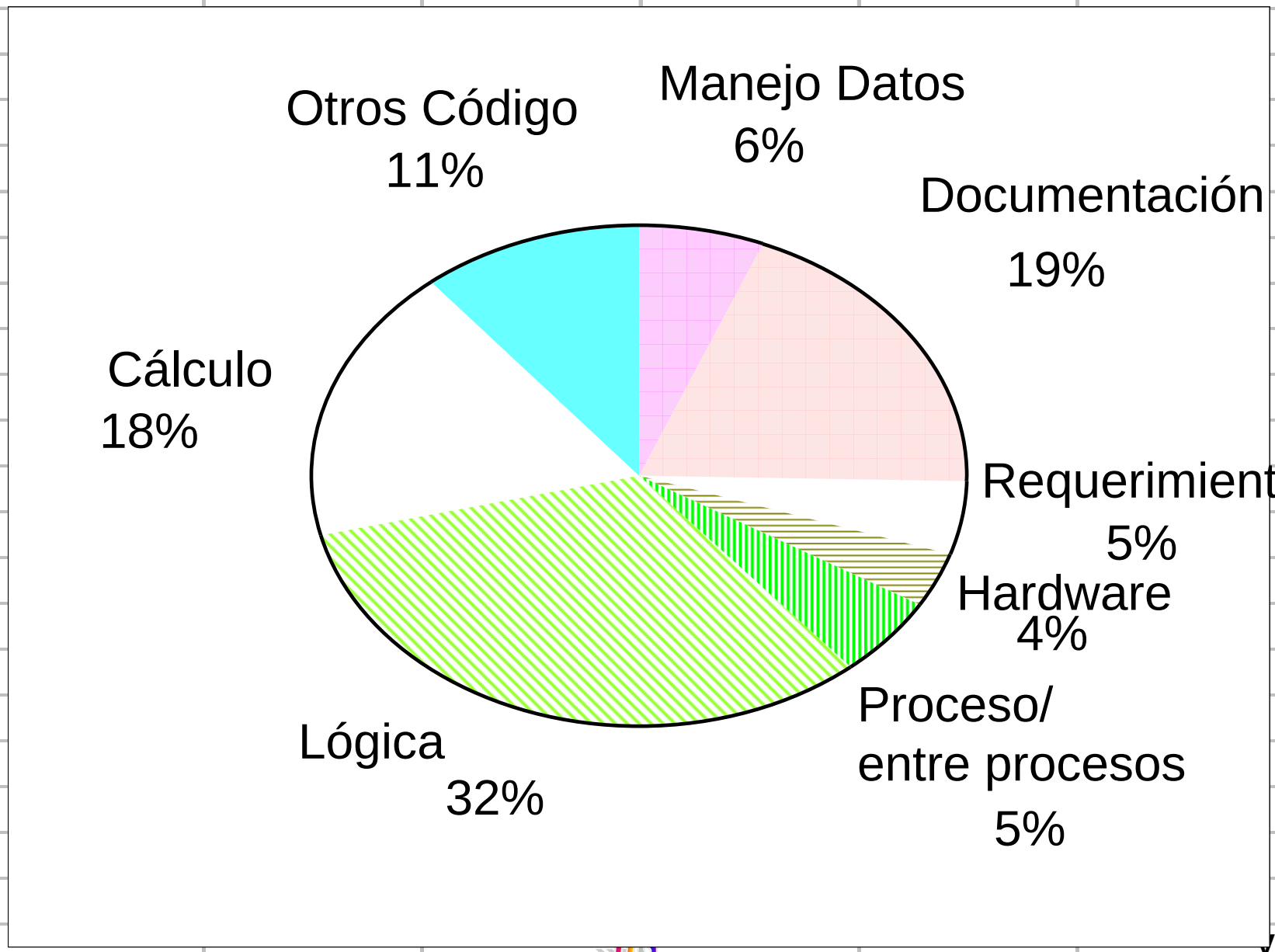
Clasificación de Defectos

- Clasificación Ortogonal de IBM
  - **Función:** afecta capacidad de brindarla, interfaz usuario, de producto, con hardware o estructura global de datos
  - **Interfaz:** al interactuar con otros componentes o drivers vía call, macros, control blocks o lista de parámetros
  - **Validación:** no valida de forma adecuada los datos antes de usarlos
  - **Asignación:** falta en inicialización de estructura de datos o bloque
  - **Tiempo/serialización:** recursos compartidos o tiempo real
  - **Build/package/merge:** problemas en repositorios, control de cambios o versiones
  - **Documentación:** publicaciones o notas de mantenimiento
  - **Algoritmo:** involucra eficiencia o correctitud de algoritmo o estructura de datos, pero no diseño

Clasificación de Defectos - HP



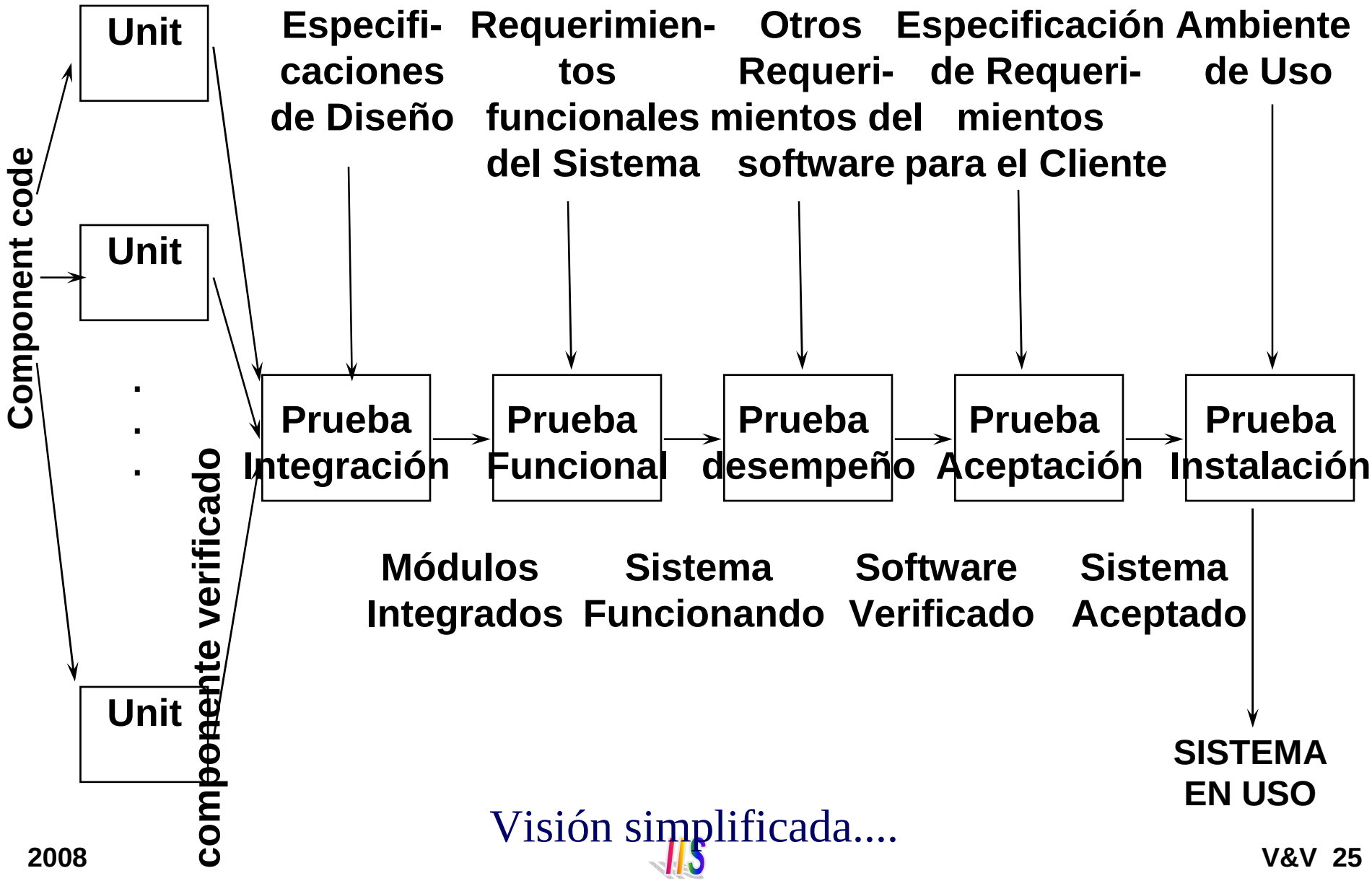
Faltas en un departamento de HP



Proceso de V&V

- Temario
  - Proceso y pruebas en el proceso
  - ¿Quién verifica?
  - Actitudes respecto a la verificación

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

Actitudes Respecto a la Verificación

- ¿Qué sabemos de un programa si pasó exitosamente un conjunto de pruebas?
- Orgullo de nuestra obra (como programador) "no es mi culpa"
- Conflictos posibles entre
  - encargado de verificación (encontrar faltas)
  - Desarrollador (mostrar las bondades de su obra)
- Soluciones:
  - Trabajo en equipo (roles distintos, igual objetivo)
  - Se evalúa al producto (no la persona)
  - Voluntad de Mejora (personal y del equipo)

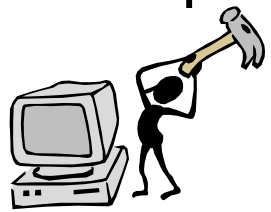
Verificación Unitaria

- Temario
  - Técnicas de verificación unitaria
  - Técnicas estáticas - Análisis
    - Análisis de código fuente
    - Análisis automatizado de código fuente
    - Análisis formal
  - Ejecución simbólica
  - Técnicas dinámicas – Pruebas
    - Definiciones, proceso para un módulo, conceptos básicos, teoría
    - Caja Blanca
    - Caja Negra
  - Comparación de las técnicas

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

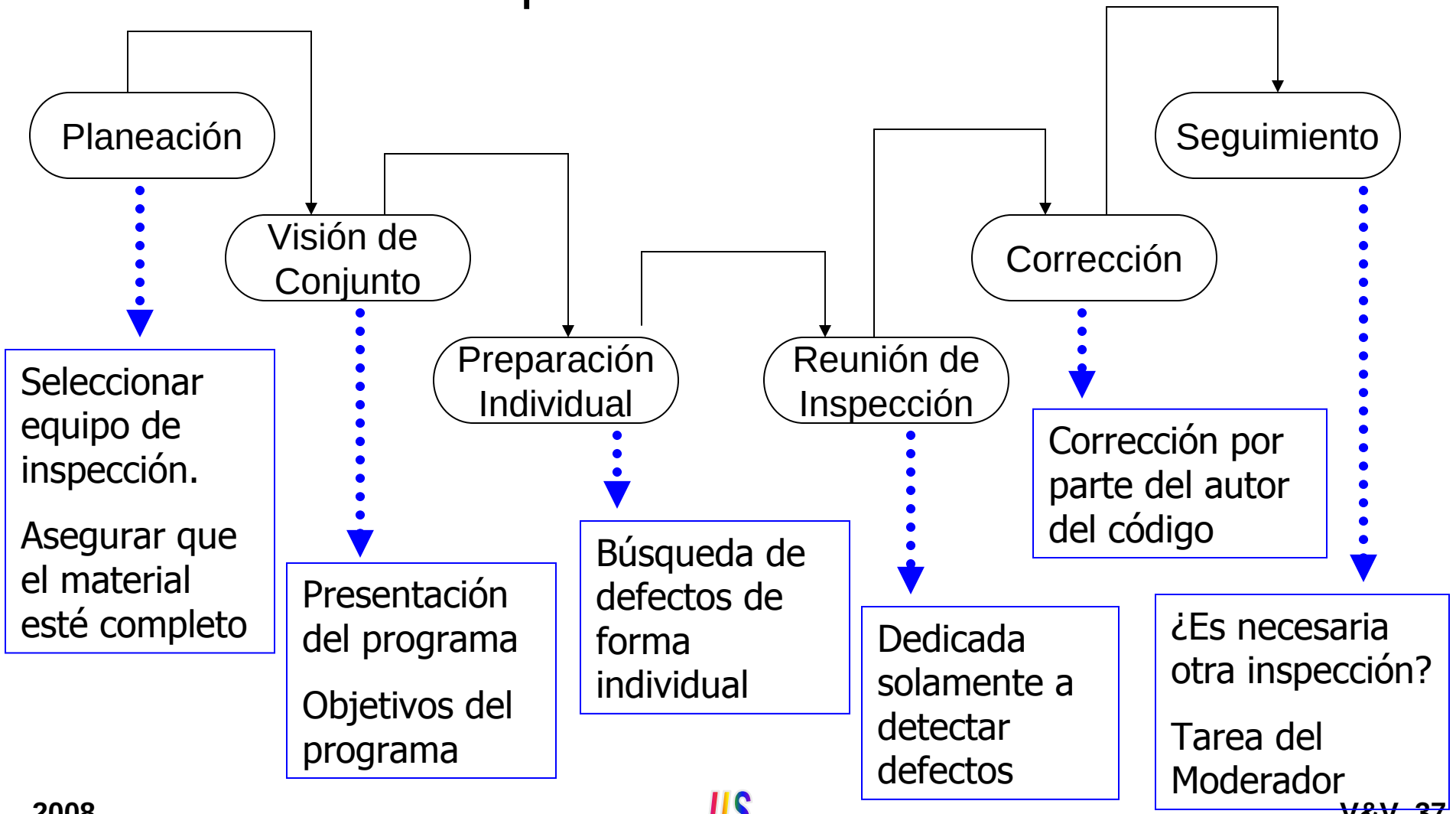
- Recorridas
  - Se simula la ejecución de código para descubrir faltas
  - Número reducido de personas
  - Los participantes reciben antes el código fuente
  - Las reuniones no duran más de dos horas
  - El foco está en detectar faltas y no en corregirlas
  - Los roles clave son:
    - Autor: Presenta 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
- Ejemplo

```
a := a + 1
a := 2
```

El analizador detecta que se asigna dos veces la variable sin usarla entre las asignaciones

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 → igual hay que validar mediante pruebas. Esto ocurre siempre ***(nada es 100% seguro)***

Análisis Formal

- Que sea la máquina la que demuestre
- Desarrollar herramientas que tomen:
  - Especificación formal
  - Código del componente a verificar
- Responda al usuario:
  - (1) el componente es correcto o
  - (2) muestre un contraejemplo para mostrar que la entrada no se transforma de forma adecuada en la salida
- Esta herramienta no puede ser construida (problema de la parada) → *Demostración Asistida*

Ejecución Simbólica

- Ejemplo:

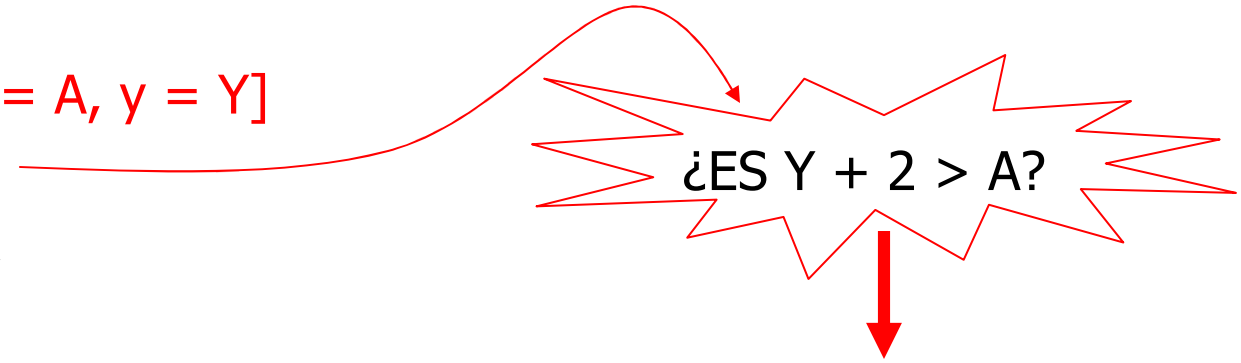
```
read(a);
read(b);
x := a + 1;
y := x * b;
write(y);
```

```
read(a);
[a = A]
read(b);
[a = A, b = B]
x := a + 1;
[a = A, b = B, x = A + 1]
y := x * b;
[a = A, b = B, x = A + 1, y = (A + 1) * B]
write(y);
[se despliega el valor (A + 1) * B]
```

Ejecución Simbólica

- Ejemplo 2:

```
[x = X, a = A, y = Y]
x := y + 2;
[x = Y + 2, a = A, y = Y]
if x > a then
    a := a + 2;
else
    y := x + 3;
[x = Y + 2, a = A, y = Y + 5] [Y + 2 <= A]
end if
x := x + a + y;
[x = 2Y + A + 7, a = A, y = Y + 5] [Y + 2 <= A]
```



HACEMOS CADA CASO Y REGISTRAMOS LAS CONDICIONES ASUMIDAS

Ejecución Simbólica

- ✓ El resultado es más genérico que las pruebas
- ✗ Es experimental
- ✗ Tiene problemas de escala
- ✗ ¿Qué pasa con las interfaces gráficas?
- ✗ ¿Y con SQL?

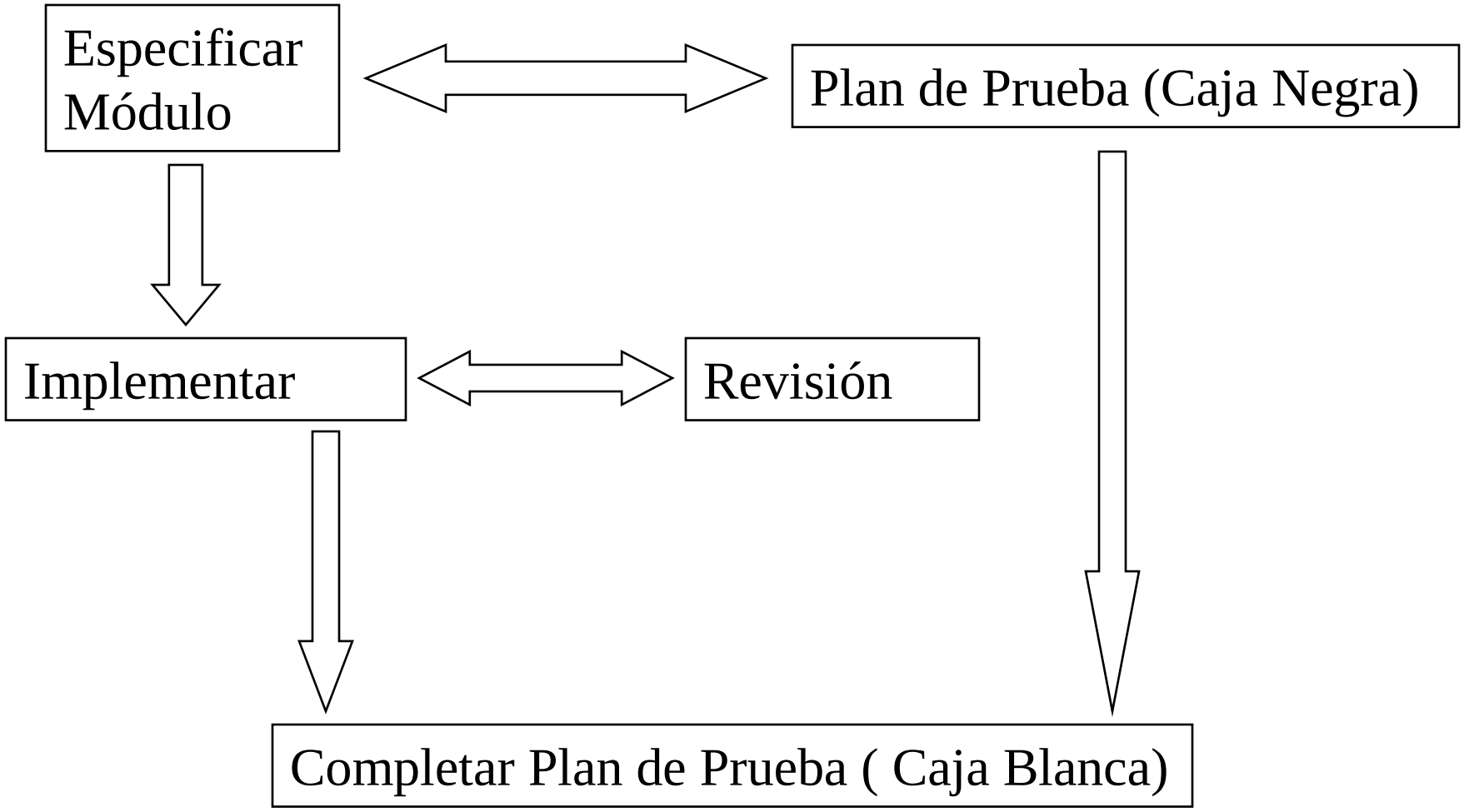
Algunas Definiciones

- 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

Únicas que asegurarían correctitud

- 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.

Fundamentos Teóricos

- Consideremos a un programa **P** como una función (posiblemente parcial) con dominio **D** (entradas de P) y codominio **R** (salidas de P)
- Sean **RS** los requerimientos sobre la salida de **P**
- Para cierto **d** $\in$ **D** decimos que **P** es correcto para **d** si **P(d)** satisface **RS**. **P** es correcto sii es correcto para todo **d** en **D**
- Un *caso de prueba* es un elemento **d** de **D**
- Un *conjunto de prueba* es un conjunto finito de casos de prueba

Fundamentos Teóricos

- Decimos que **P** es correcto para un conjunto de pruebas **T**, si es correcto para todos los casos de prueba de **T**. En este caso decimos que **P** es *exitoso* en **T**
- Un *criterio de selección* **C** es un subconjunto del conjunto de subconjuntos finitos de **D**
- Un conjunto de prueba **T** *satisface* **C** si pertenece a **C**
- Un criterio de selección **C** es *consistente* si para cualquier par **T1**,**T2** que satisfacen **C**, **P** es exitoso en **T1** si y sólo si es exitoso en **T2**

Fundamentos Teóricos

- Un criterio de selección **C** es *completo* si, siempre que **P** sea incorrecto, existe un conjunto de prueba **T** que satisface **C** para el que **P** no es exitoso
- Si **C** fuera a la vez *consistente* y *completo*, podríamos discriminar (de forma automática) si un programa es o no correcto
- ¿Podremos construir un algoritmo para generar **C**?
  - No. Este es otro problema indecidible

Fundamentos Teóricos

- Entrada:
  - Un número del 1 al 3
- Conjunto de subconjuntos finitos:
  - { {1}, {2}, {3}, {1,2}, {1,3}, {2,3}, {1,2,3} }
- Criterio
  - Que al menos esté el número 1
  - **C** = { {1}, {1,2}, {1,3}, {1,2,3} }
- Conjuntos de prueba que satisfacen **C**
  - **T1** = {1} **T2** = {1,2} **T3** = {1,3} **T4** = {1,2,3}

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
  - Mutation testing
    - Se basan en crear mutaciones del programa original provocando distintos cambios en este último
    - La idea atrás de esta forma de test es poder distinguir (usando los casos de test) los mutantes del original
    - No se va a entrar en detalle

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
}
```

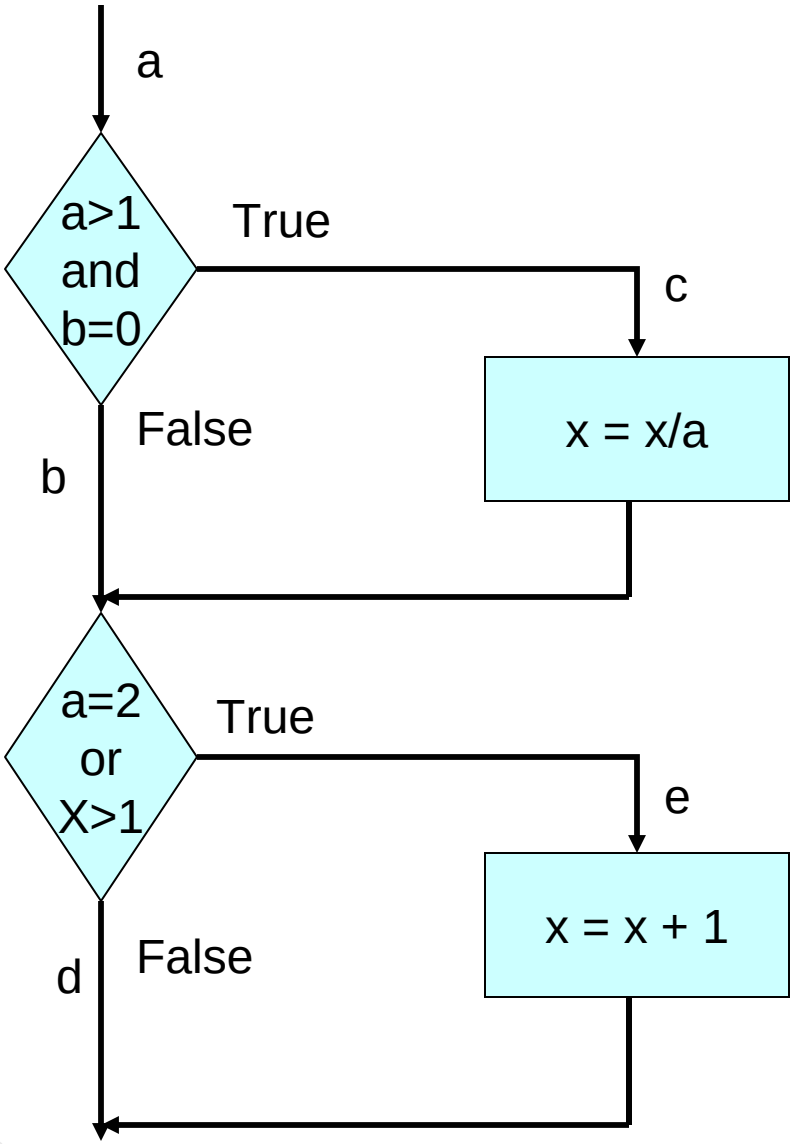
CP : Entrada a=2, b=0, x=3

- ¿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

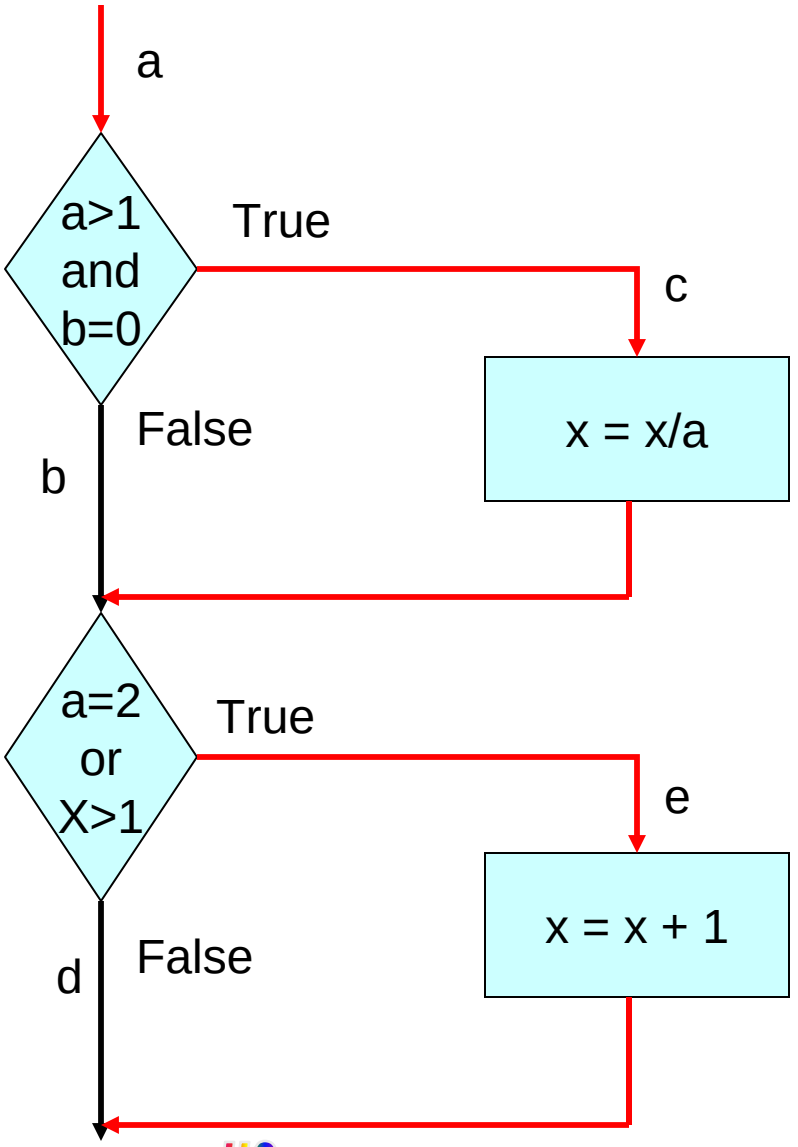
```
If (a > 1) and (b = 0) {
    x = x / a
}
If (a = 2) or (x > 1) {
    x = x + 1
}
```



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

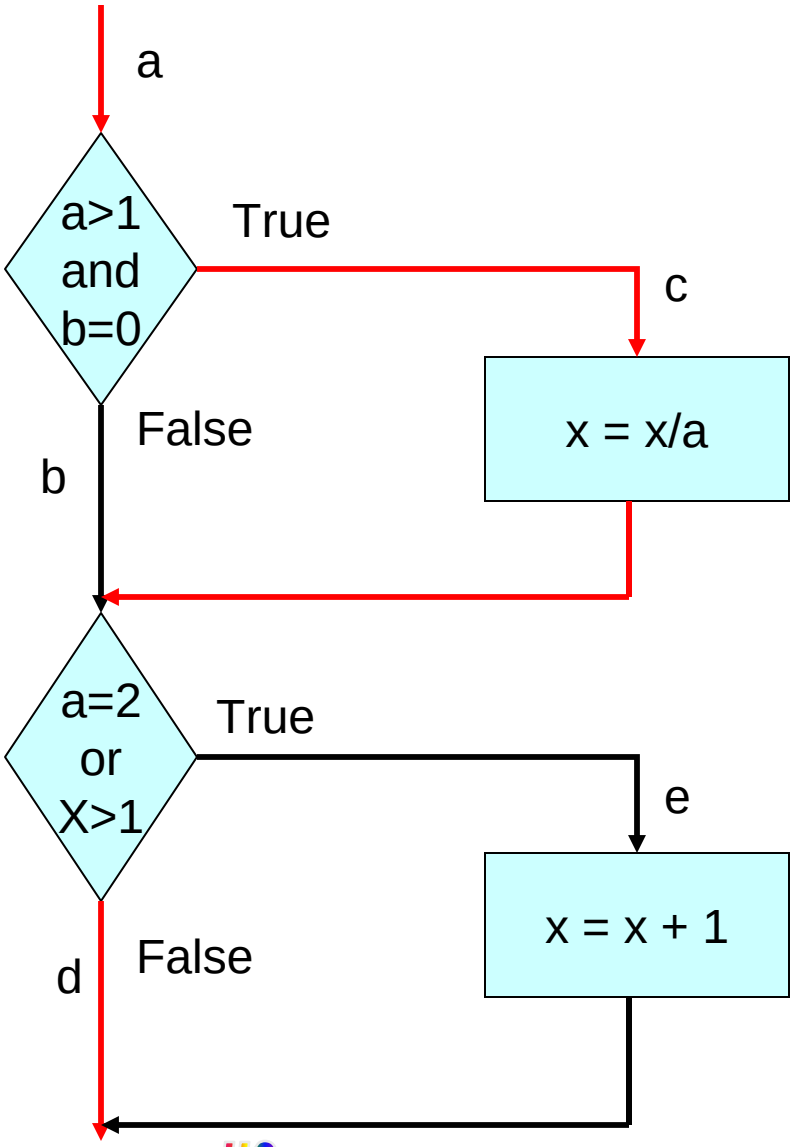
- ¿Qué pasa si en la segunda decisión tuviera x < 1 en lugar de x > 1?
- Hay una secuencia en la cual x se mantiene sin cambios y esta no es probada
- Hay que extender el criterio para sentencias del tipo CASE

Es más fino que el criterio de sentencias

Caja Blanca

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

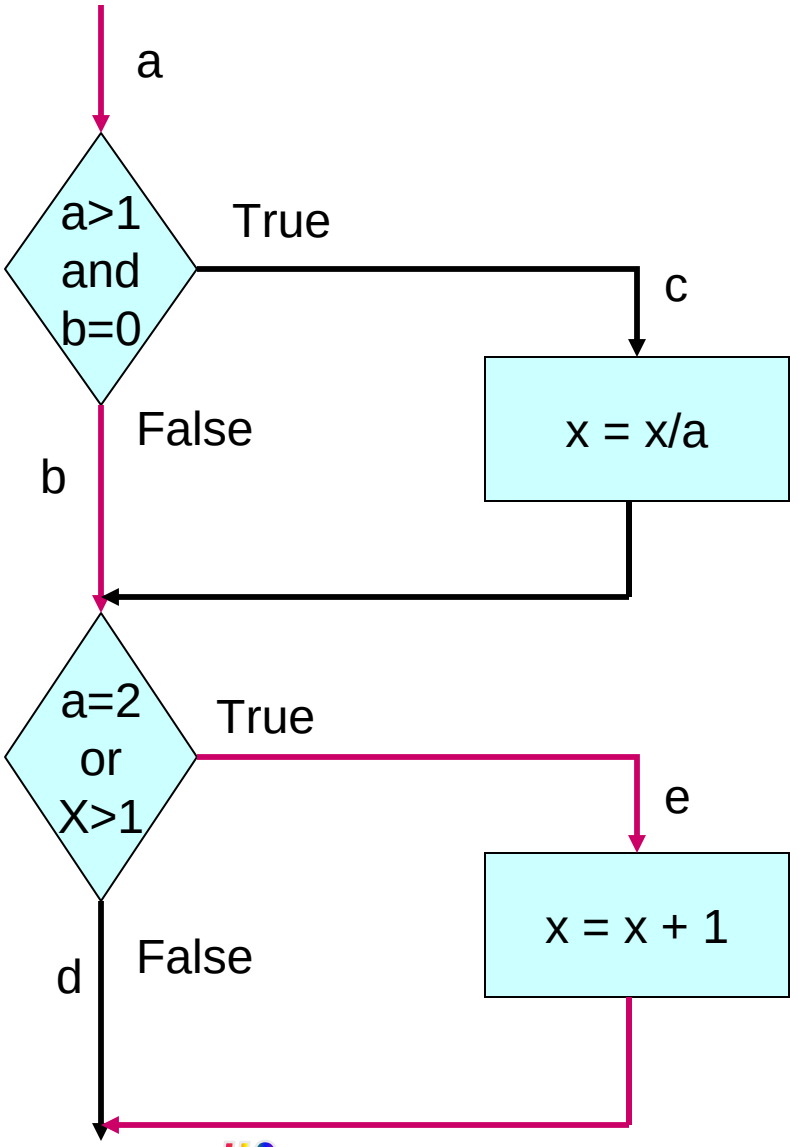
Secuencia: acd



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

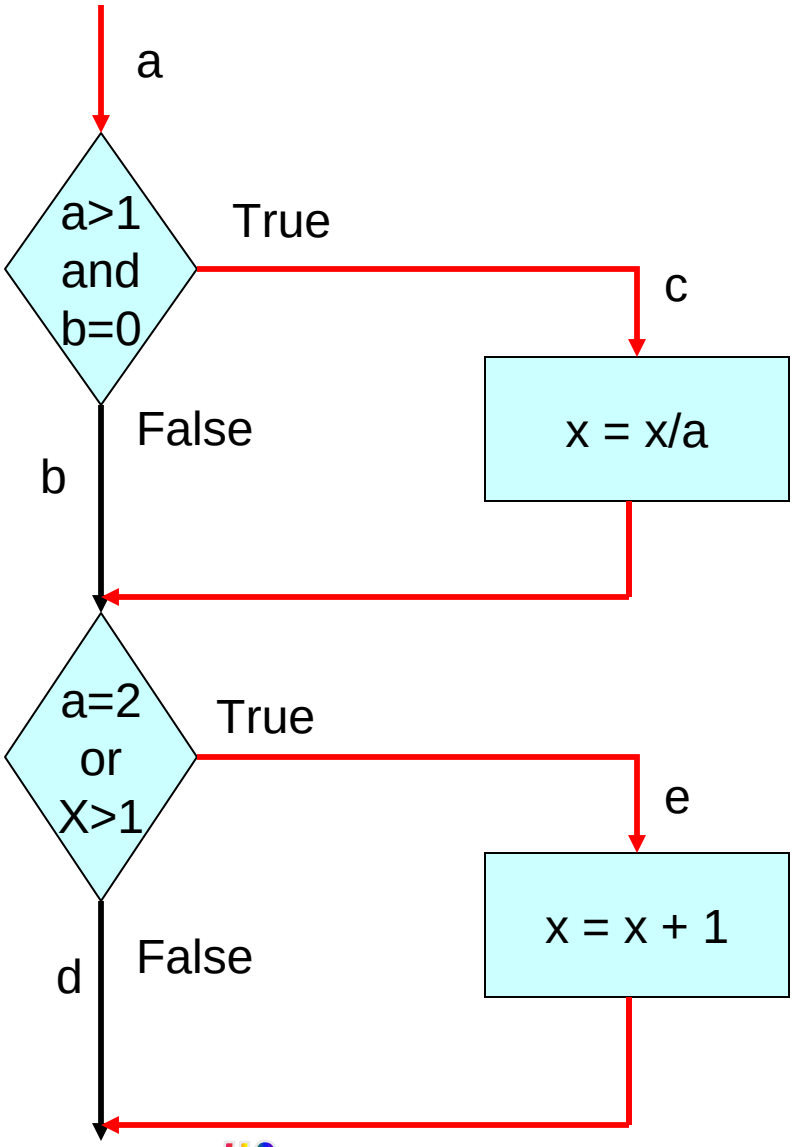
Esto no satisface el criterio de decisión

Este criterio es generalmente más fino que el de decisión

Caja Blanca

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

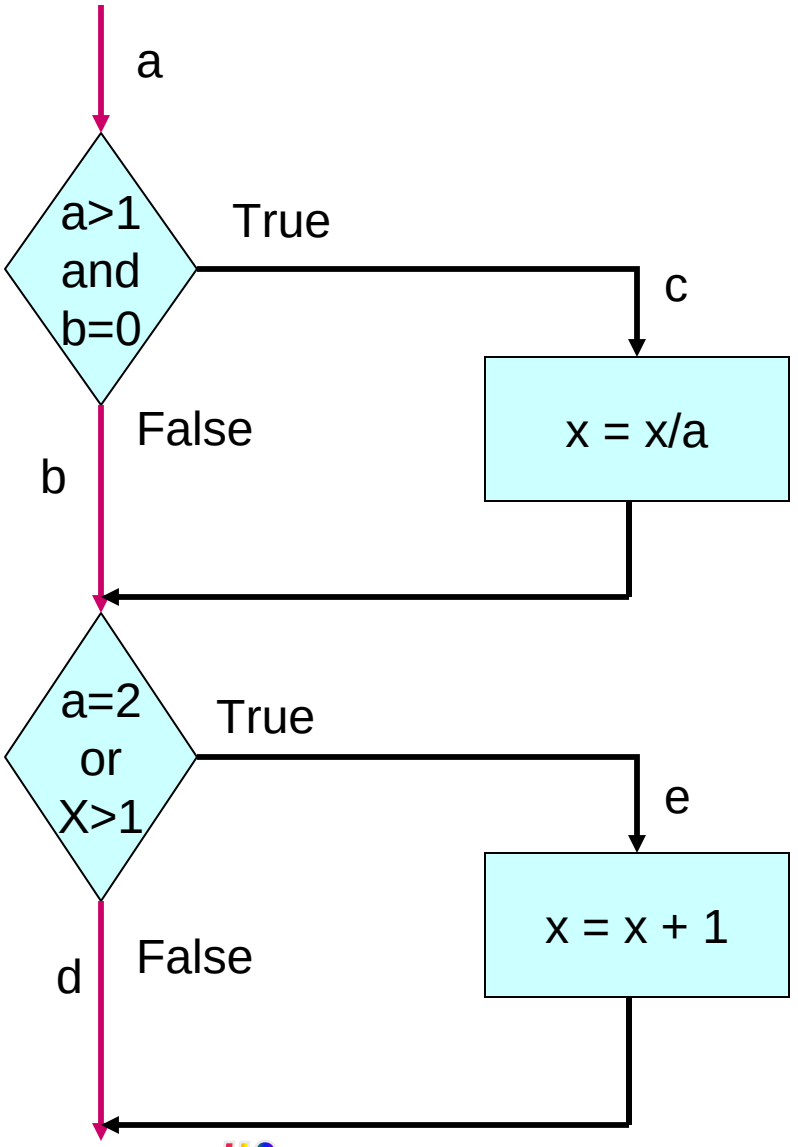
Secuencia: ace



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}

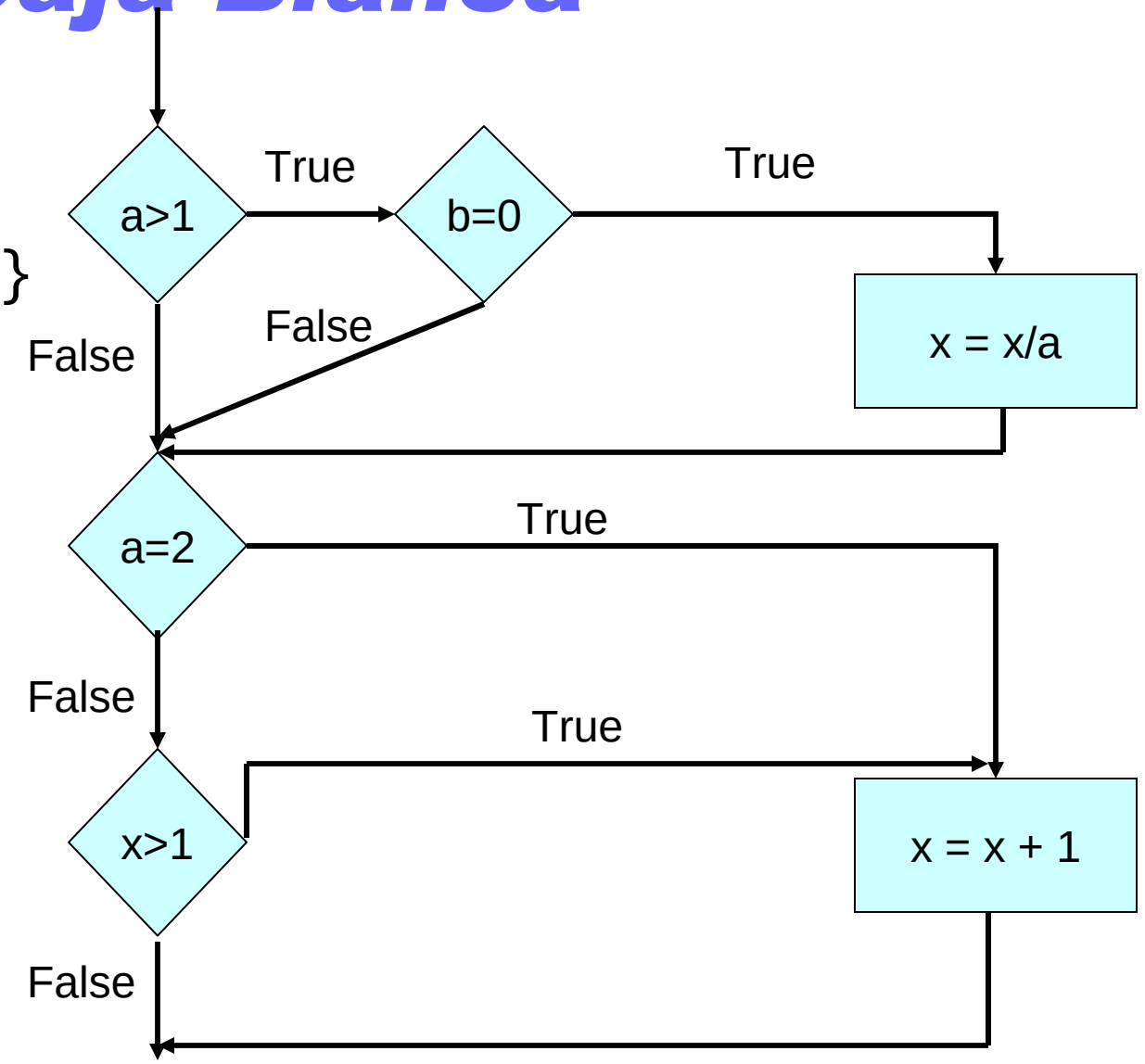
CP1 a=2, b=0, x=4
CP2 a=1, b=1, x=1

- 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)

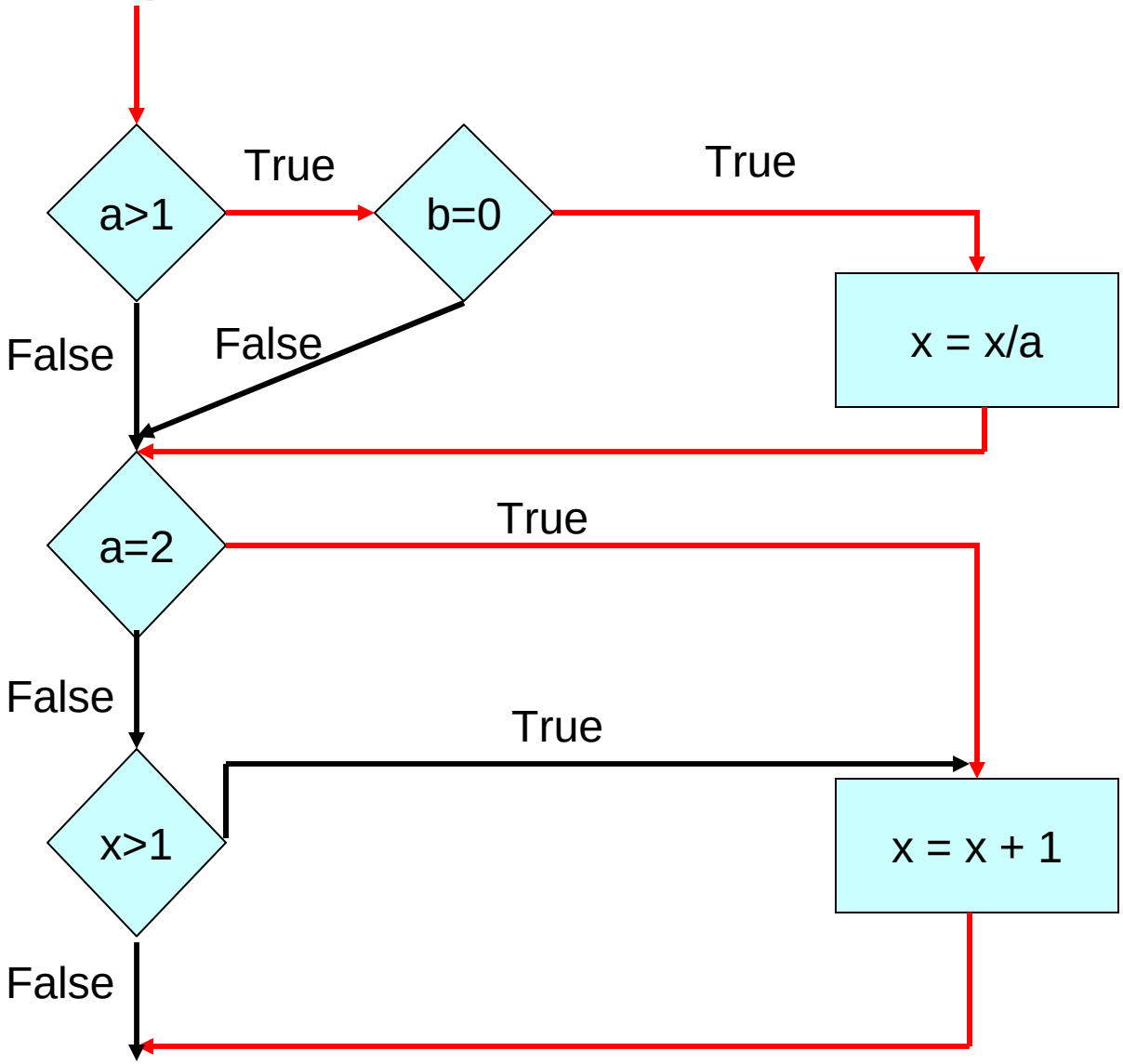
Caja Blanca

```
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:
```



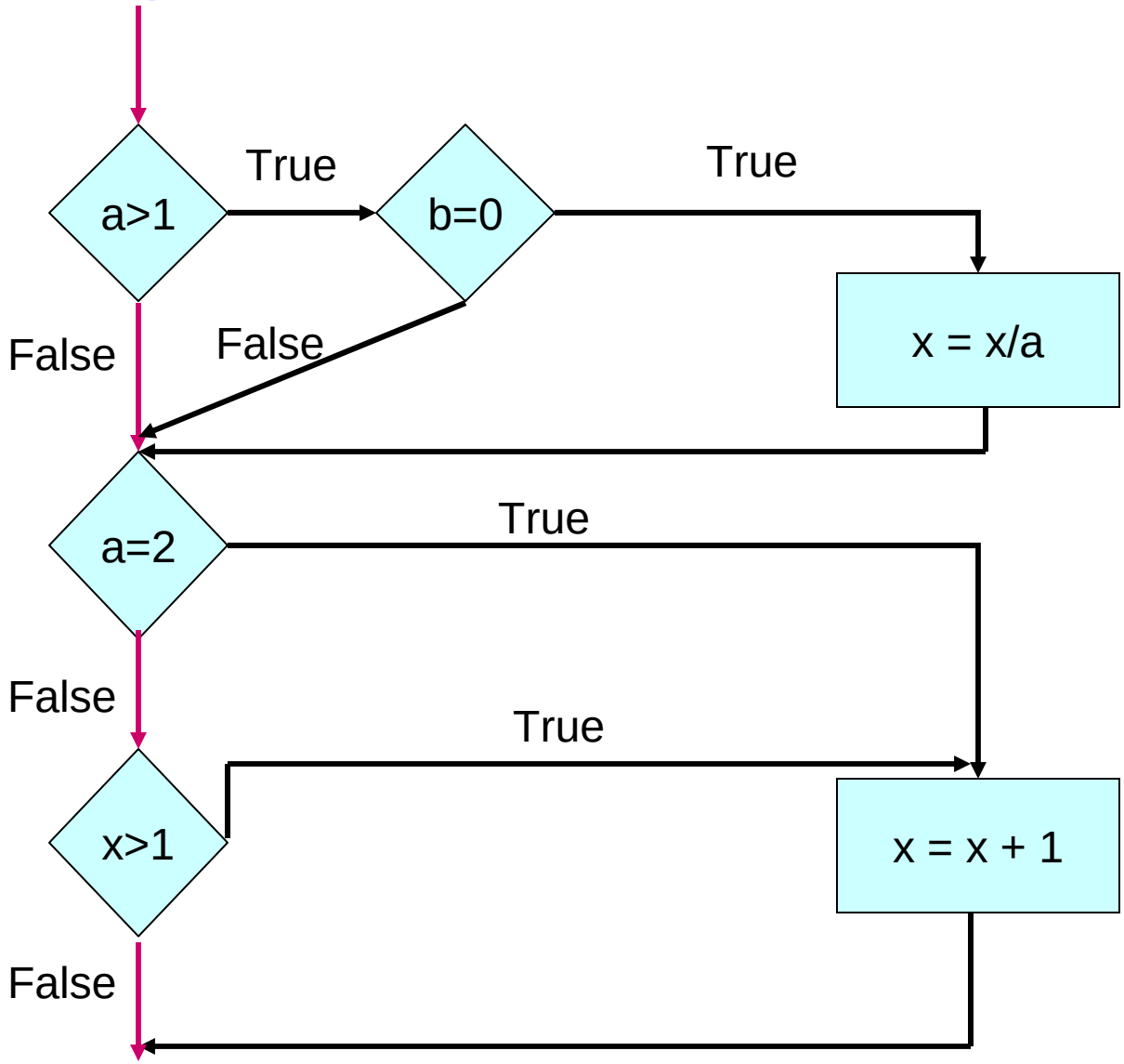
Caja Blanca

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



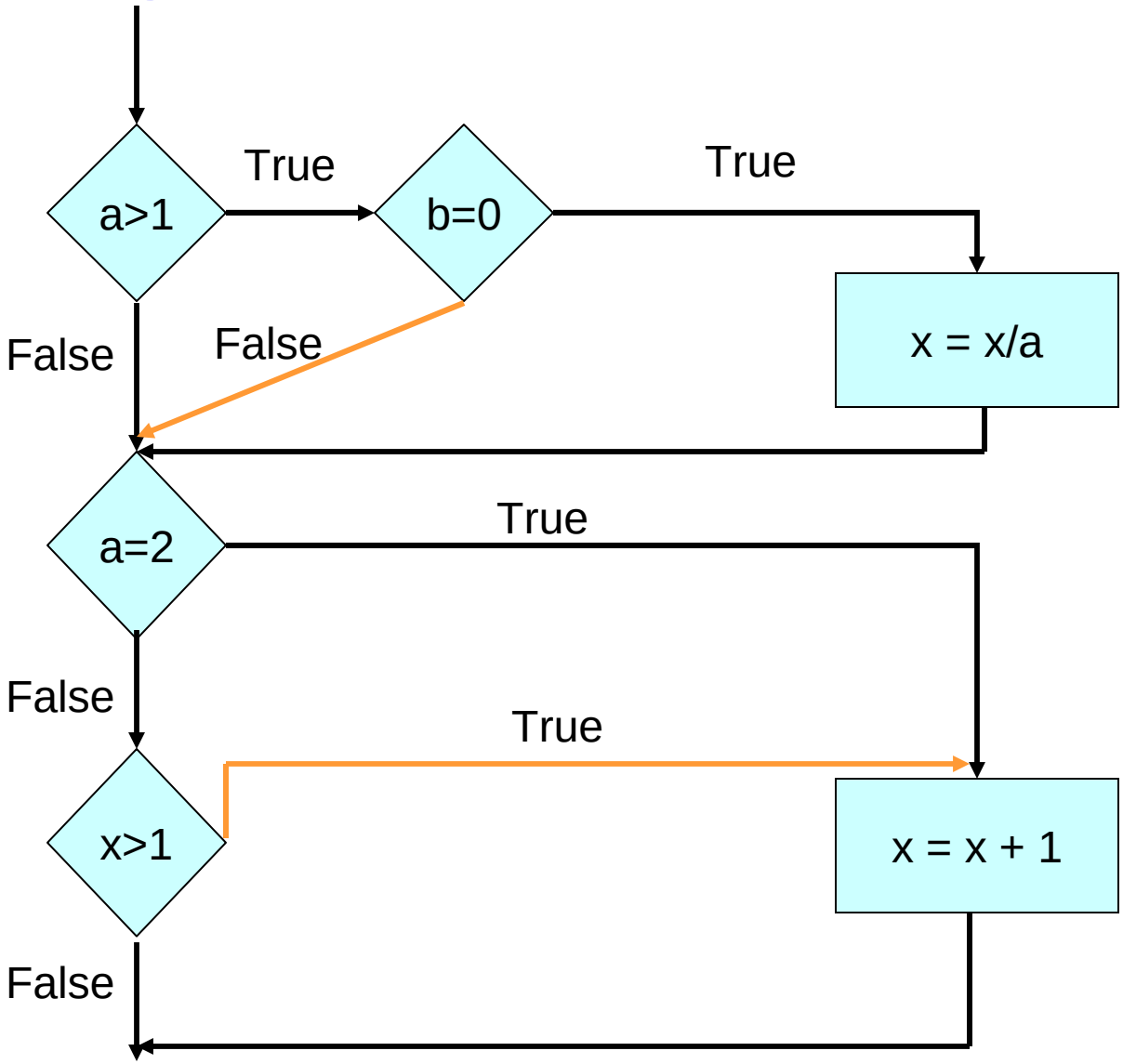
Caja Blanca

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

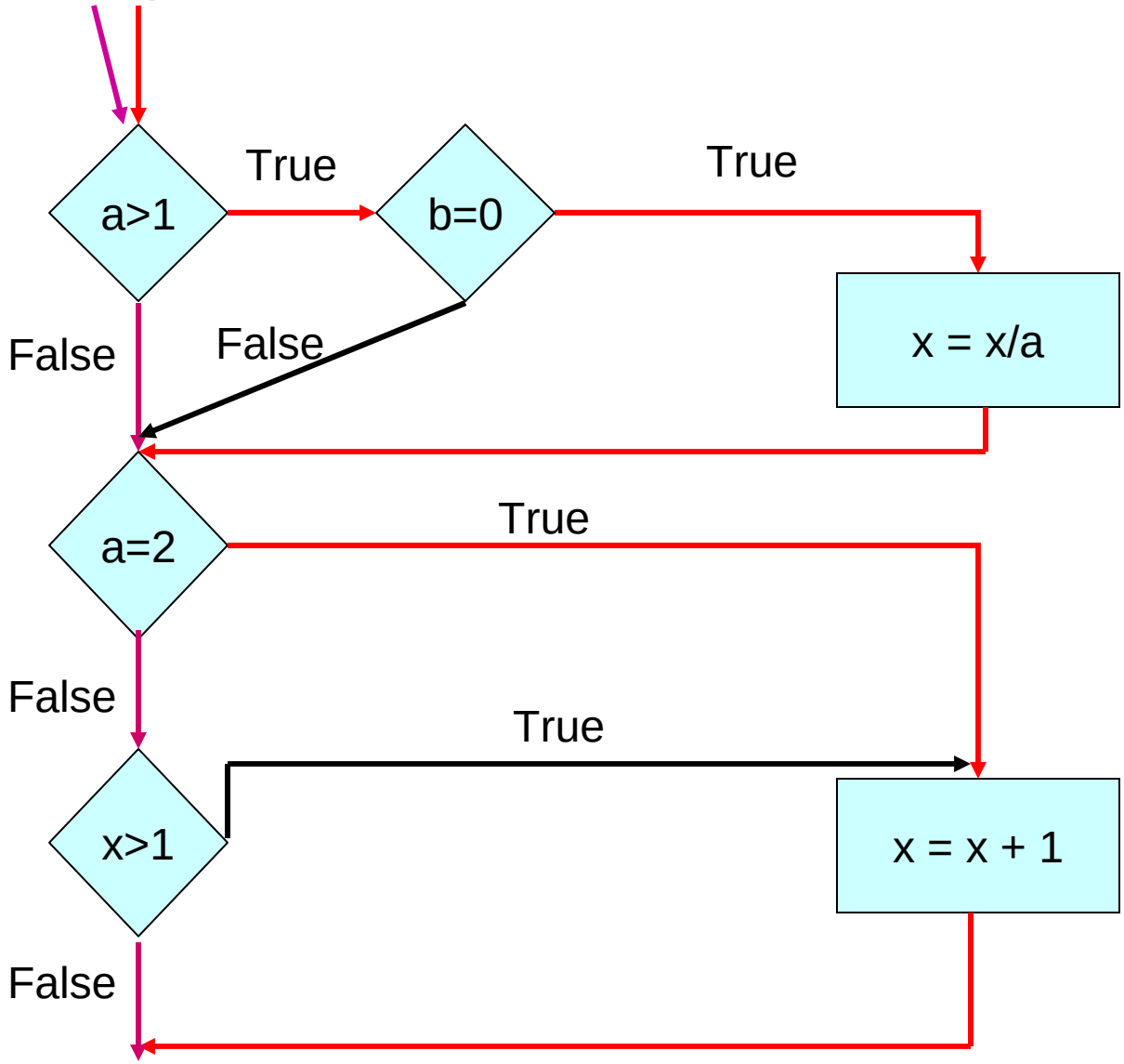
- La secuencia acd queda sin ejecutar → este criterio no asegura testear todos los caminos posibles

Incluye al criterio de condición/decisión.

Myers lo considera un criterio aceptable

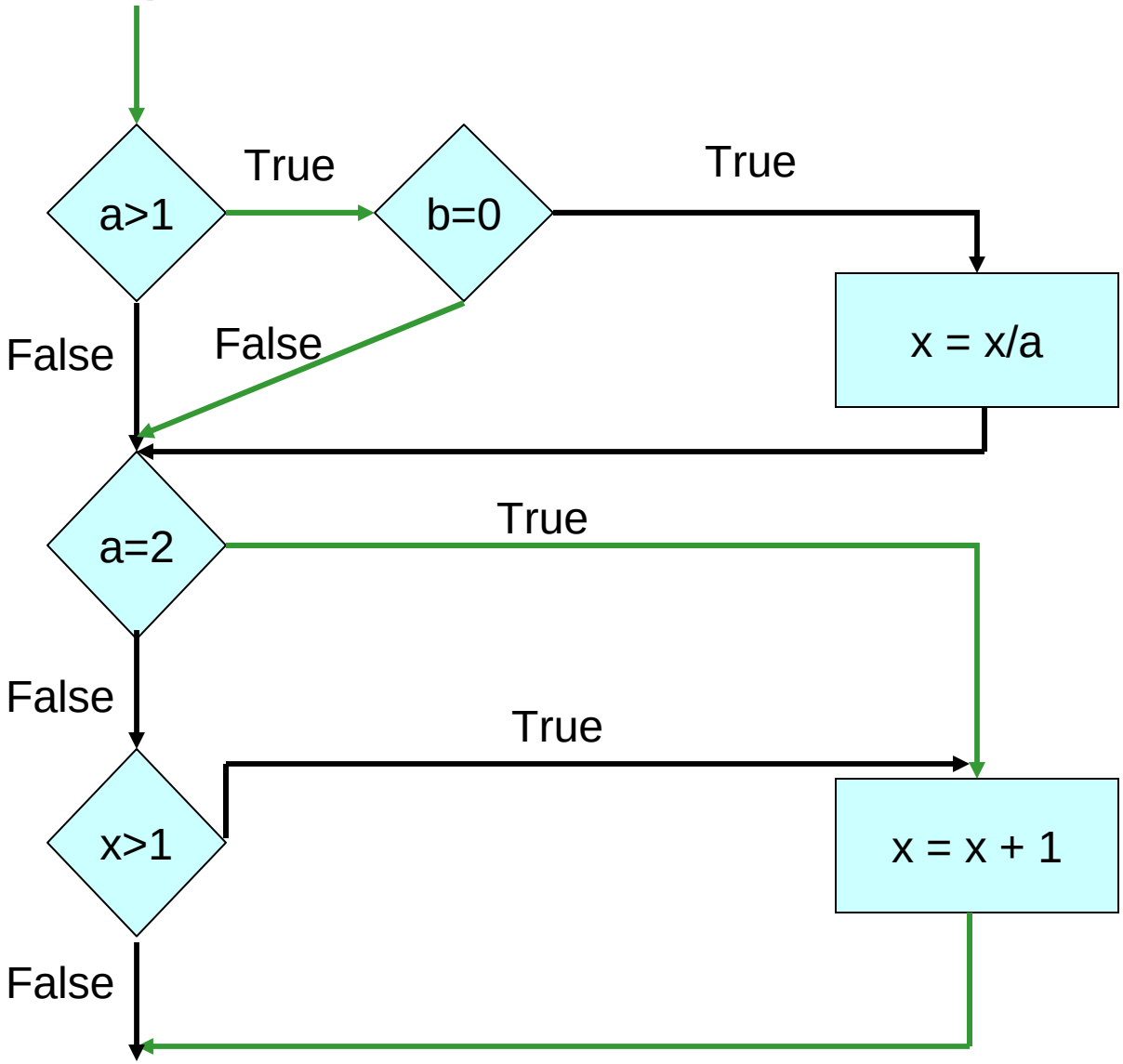
Caja Blanca

CP1 y CP4 ya los vimos



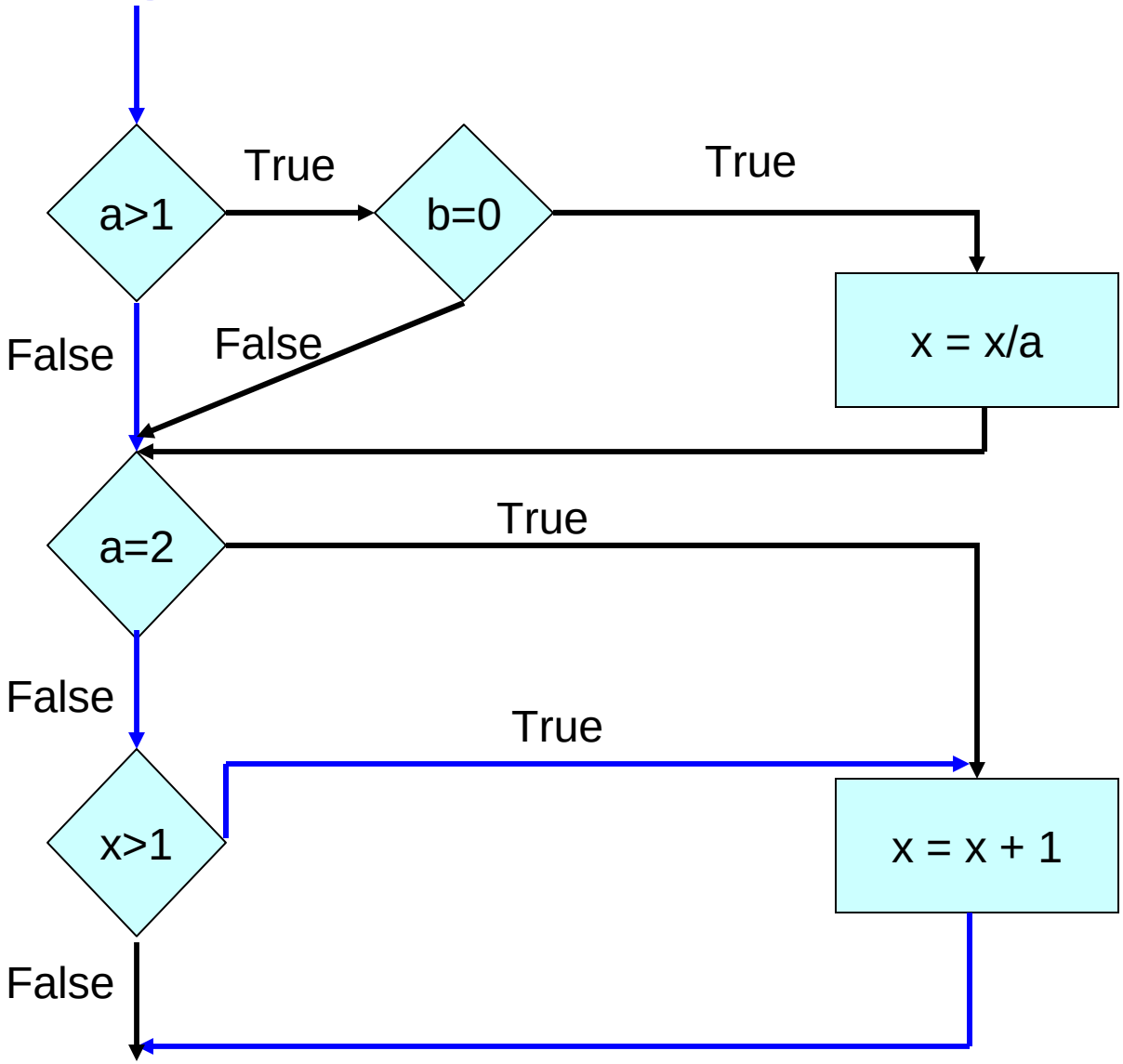
Caja Blanca

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
  - El criterio no especifica con qué grafo se trabaja; el común o el extendido (grafo de flujo de control dónde está dividida cada decisión en decisiones simples)
  - Si es con el común este criterio es igual al de decisión

Caja Blanca

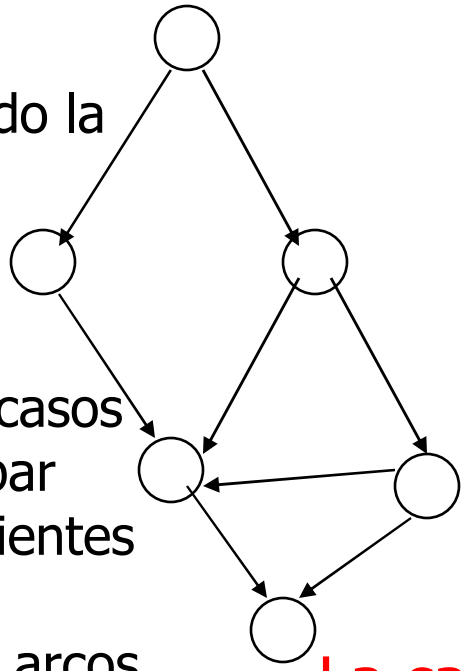
- Criterio de cubrimiento de trayectorias indep.
  - Ejecutar al menos una vez cada trayectoria independiente

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



Si se divide en decisiones de una única condición (esto no está especificado en el criterio), es el más fino de los vistos hasta ahora.

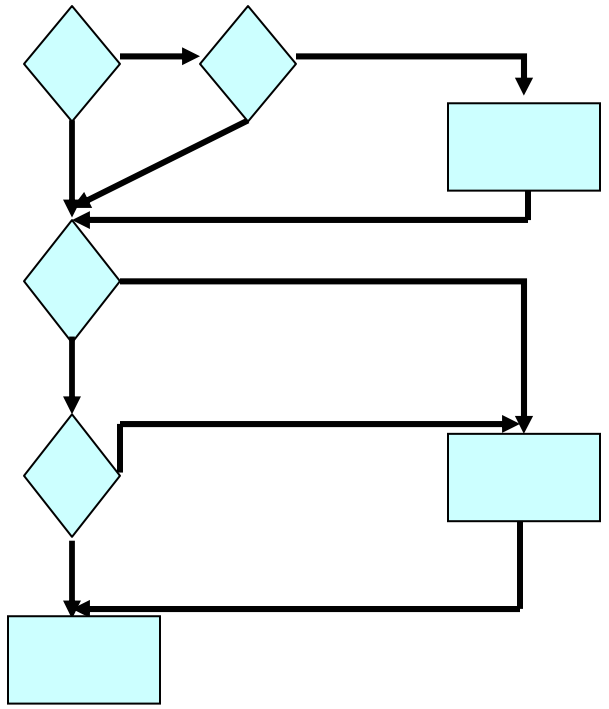
La cantidad de casos de prueba suele ser demasiado grande. Es un criterio de referencia

Caja Blanca

- Trayectorias independientes
  - El ejemplo que venimos siguiendo

- ¿Cuántas trayectorias hay desde el nodo inicial al nodo final?
  - Repuesta: 9

- ¿Cuántas trayectorias independientes hay?
  - Respuesta: CC = 5



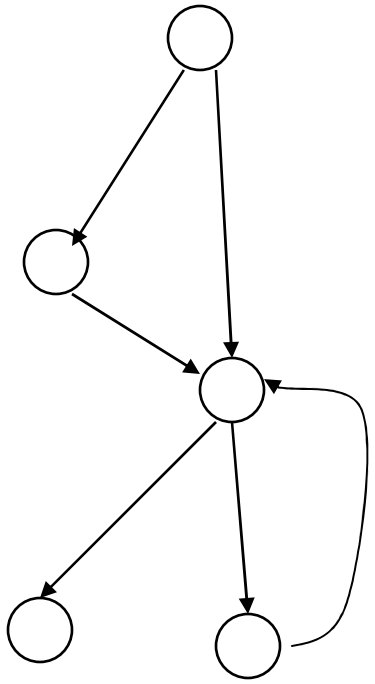
Caja Blanca

- Trayectorias independientes
  - Un ejemplo más sencillo

- ¿Cuántas trayectorias hay desde el nodo inicial al nodo final sin repetir el bucle?
  - Repuesta: 4
- ¿Y repitiendo el bucle?
  - Respuesta: Puede ser infinito

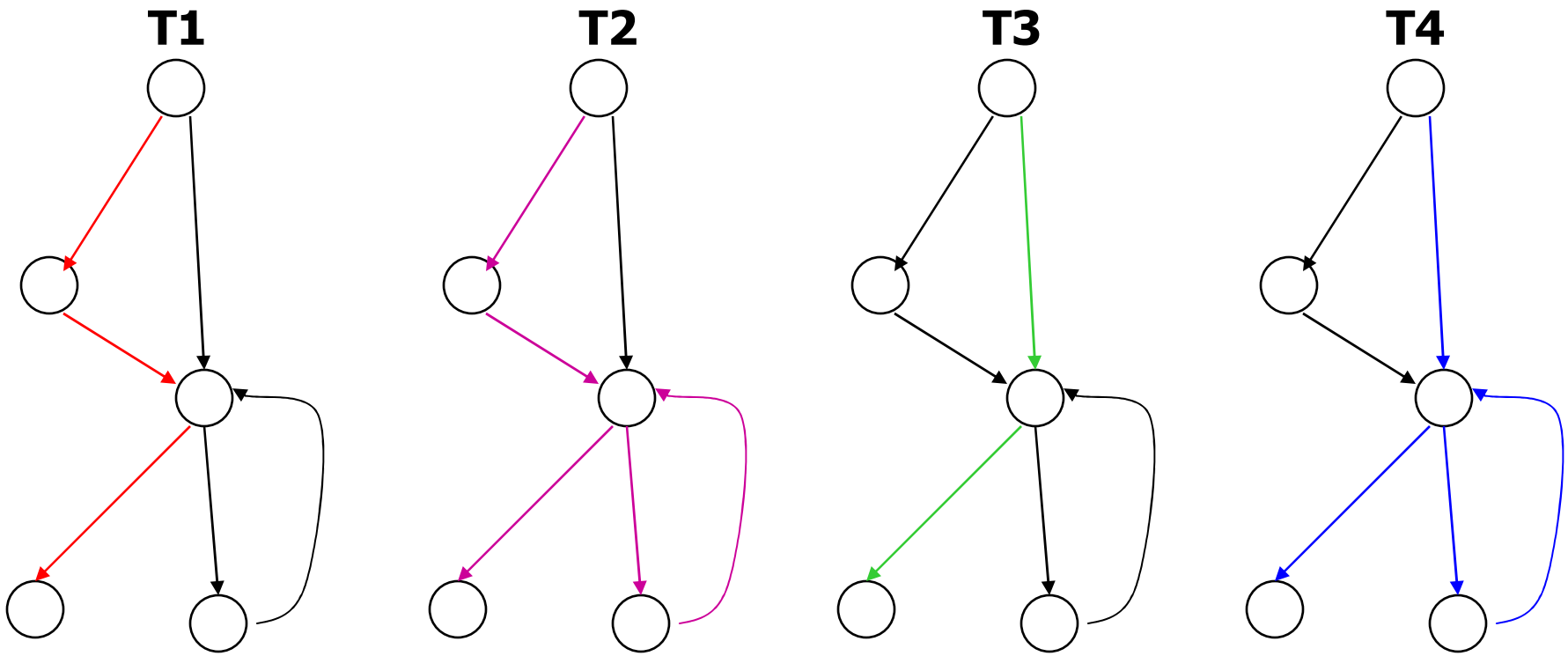
¿Cuántas trayectorias independientes hay?

- Respuesta: CC = 3



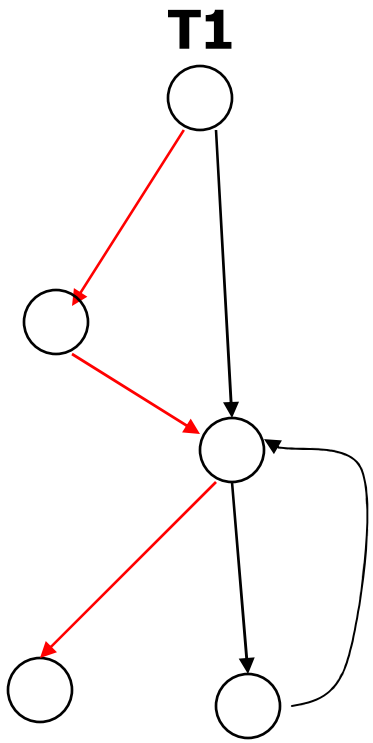
Caja Blanca

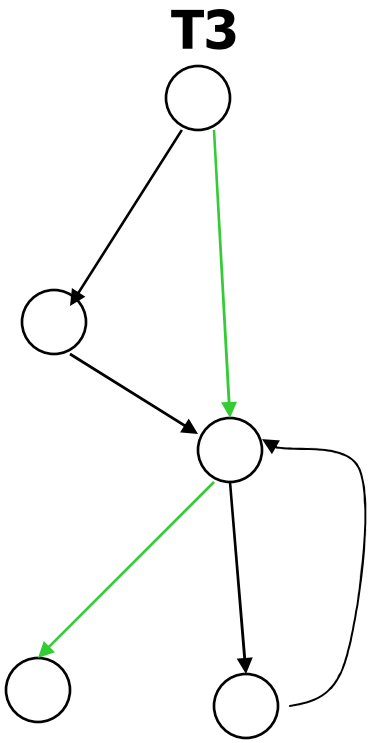
- Trayectorias independientes
  - Un ejemplo más sencillo – Las 4 trayectorias

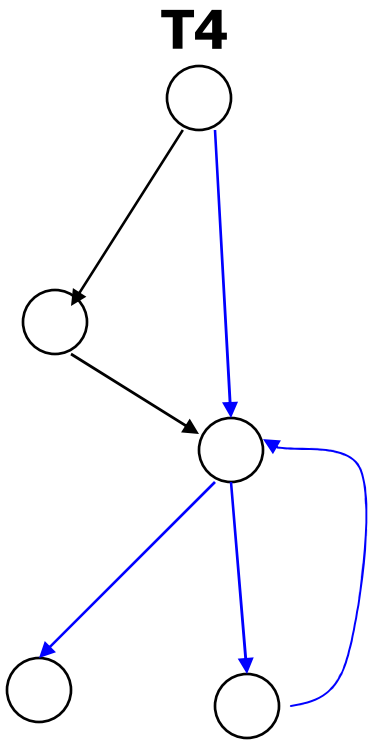


Caja Blanca

- Trayectorias independientes
  - Un ejemplo más sencillo – 3 Trayectorias independientes

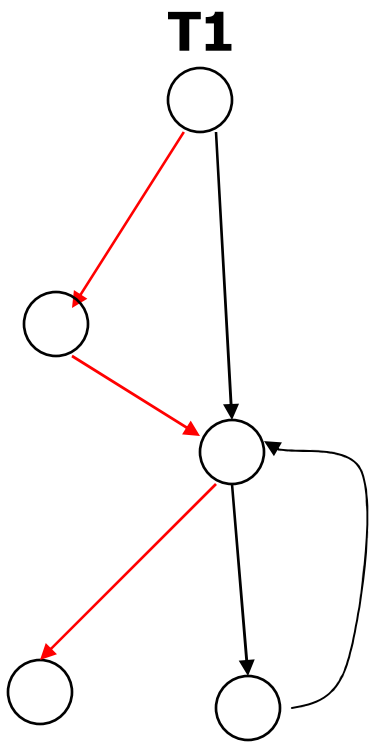


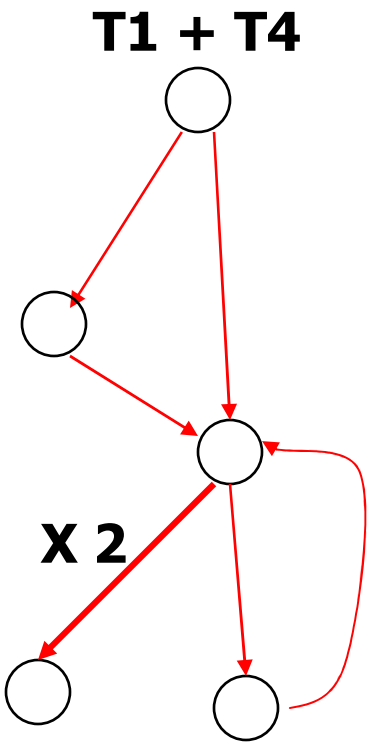


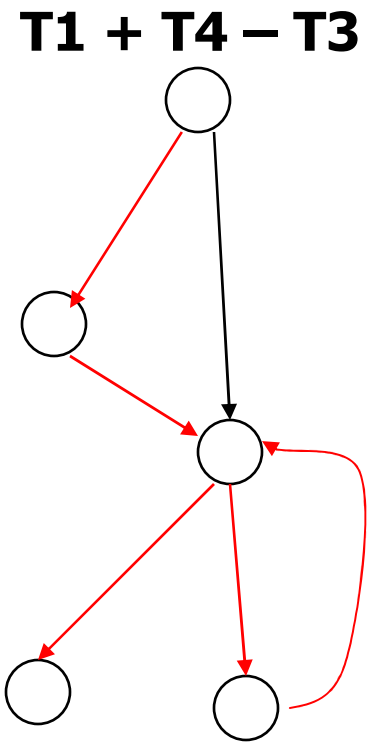


Caja Blanca

- Trayectorias independientes
  - Un ejemplo más sencillo – Como llego a T2



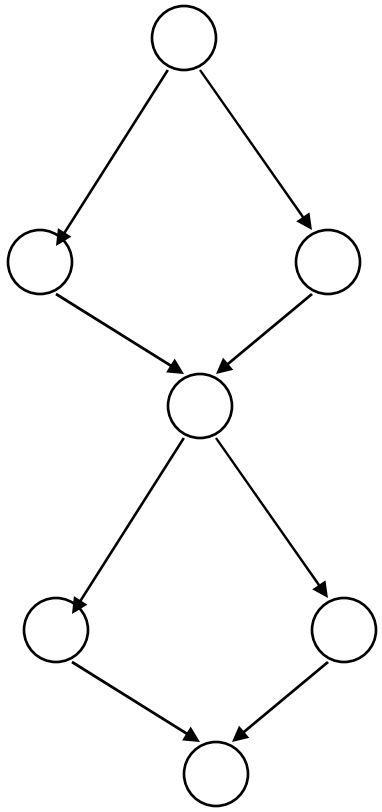




Caja Blanca

- Trayectorias independientes
  - ¿Este criterio detecta el defecto de ejemplo que no detecta CCCM?

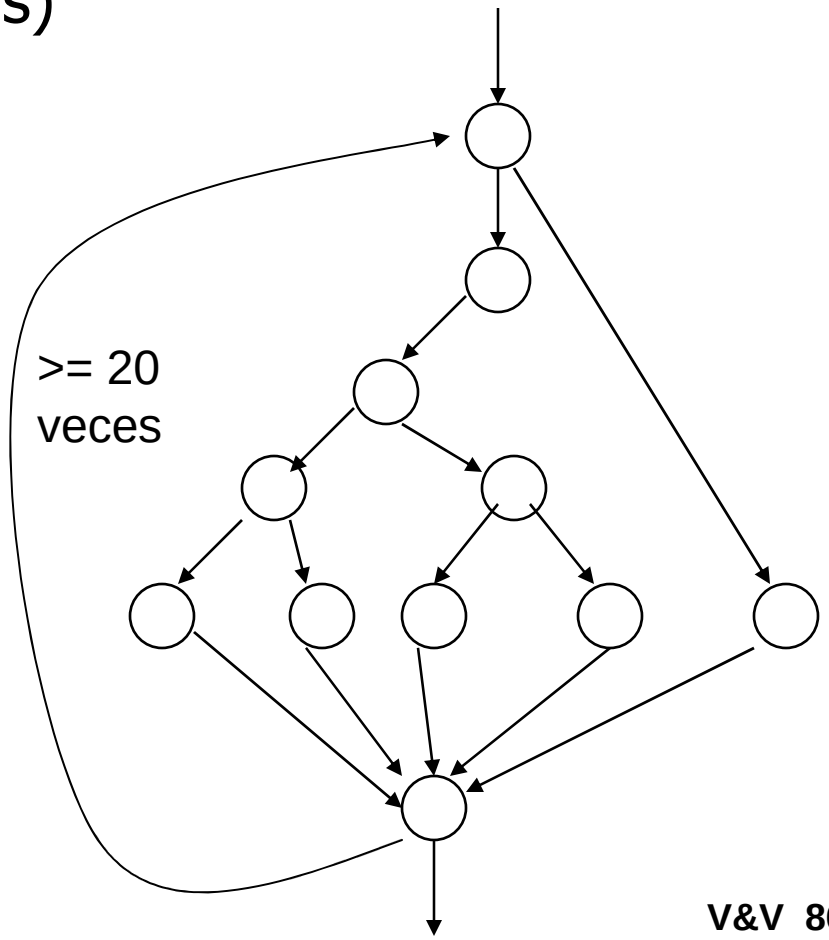
```
If x <> 0
      y = 5
else
      z = z – x
If z > 1
      z = z / x
else
      z = 0
```



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 Blanca

- Criterios basados en el flujo de datos
- Definiciones
  - Una variable en el lado izquierdo de una asignación es una ***definición*** de la variable
  - Una variable en el lado derecho de una asignación es llamada un ***uso computacional*** o un ***c-uso***
  - Una variable en un predicado booleano resulta en un par ***uso predicado*** o ***p-uso*** correspondiendo a las evaluaciones verdadero y falso del predicado
  - Un camino $(i,n_1,n_2,...,n_m,j)$ es un camino ***limpio-definición*** desde la salida de i hasta la entrada a j si no contiene una definición de x en los nodos de n_1 a n_m

Caja Blanca

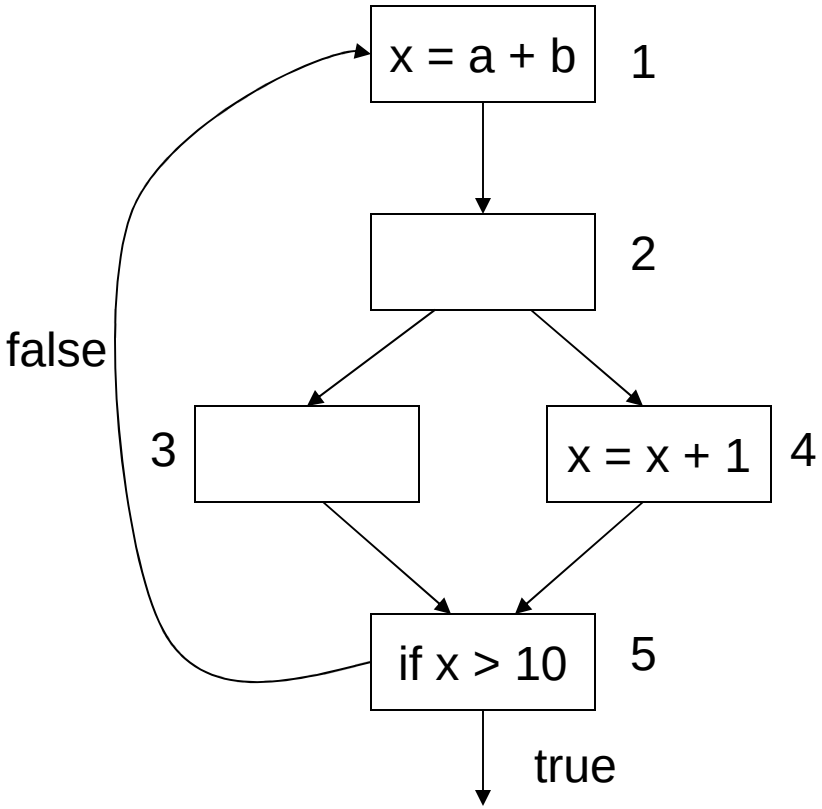
- Más definiciones
  - Dada una *definición* de x en el nodo n_d y un *c-uso* de x en el nodo $n_{c\text{-}uso}$, la presencia de un camino *limpio-definición* para x desde n_d hasta $n_{c\text{-}uso}$ establece la ***asociación definición-c-uso*** (n_d, $n_{c\text{-}uso}$, x)
  - Similar al anterior pero con *p-uso* establece un par de ***asociaciones definición-p-uso*** (n_d, ($n_{p\text{-}uso}$,t), x) y (n_d, ($n_{p\text{-}uso}$,f), x) correspondiendo a las evaluaciones de true y false en el nodo $n_{p\text{-}uso}$ respectivamente

Caja Blanca

- Más definiciones
  - ***camino-du*** para una variable x:
  - Es un ***camino-du*** si se cumple alguna de las dos condiciones siguientes
    - $(n_1,...,n_j,n_k)$ y n_1 contiene una *definición* de x y n_k es un *c-uso* de x y $(n_1,...,n_j,n_k)$ es *limpio-definición* para x y solamente pueden ser iguales n_1 y n_k (es decir, no se repiten nodos a menos de n_1 y n_k)
    - Similar para *p-uso*. La diferencia es que el camino desde n_1 hasta n_k no contiene nodos iguales

Caja Blanca

- Ejemplo



Definiciones de x: Nodos 1 y 4

C-uso de x: Nodo 4

P-uso de x: Nodo 5

Camino libre-def: (1,2,4) y (1,2,3,5)

Camino-du (1,2,4) establece una **definición-c-uso** (1,4,x)

Camino-du (1,2,3,5) establece dos **definición-p-uso** (1,(5,t),x) y (1,(5,f),x)

Caja Blanca

- Criterios de cobertura
  - Todas las definiciones
    - Se ejecuta para cada definición **al menos un** camino libre-definición para **algún** uso correspondiente (c-uso o p-uso)
  - Todos los c-usos
    - Para cada definición se ejecuta **al menos un** camino libre-definición hasta **todos** los c-usos correspondientes
  - Todos los c-usos/algún p-uso
    - Como el anterior pero si no existe ningún c-uso entonces tiene que ir a **algún** p-uso
  - Todos los p-usos
    - Similar a todos los c-usos
  - Todos los p-usos/algún c-uso
    - Similar a todos los c-usos/algún p-uso

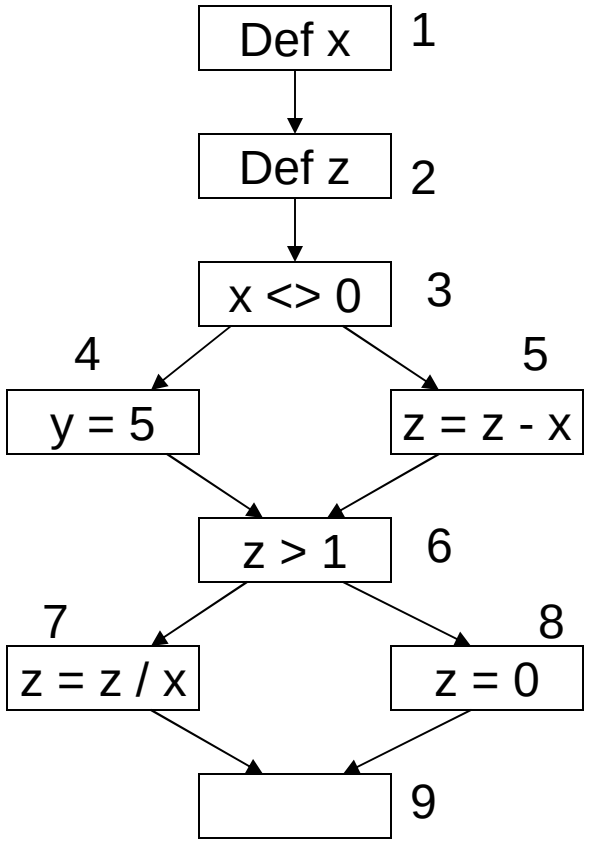
Caja Blanca

- Criterios de cobertura
  - Todos los usos
    - Se ejecuta para cada definición **al menos un** camino libre-definición que lleve a **todos** los c-usos y p-usos correspondientes
  - Todos los caminos-du
    - Se ejecutan para cada definición **todos** los caminos-du. Esto quiere decir que si hay múltiples caminos-du entre una definición y un uso **todos** deben ser ejecutados

Caja Blanca

- Volvamos al ejemplo no resuelto

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



Definiciones de z: Nodos 2, 5, 7 y 8

C-uso de z: Nodos 5 y 7

P-uso de z: Nodo 6

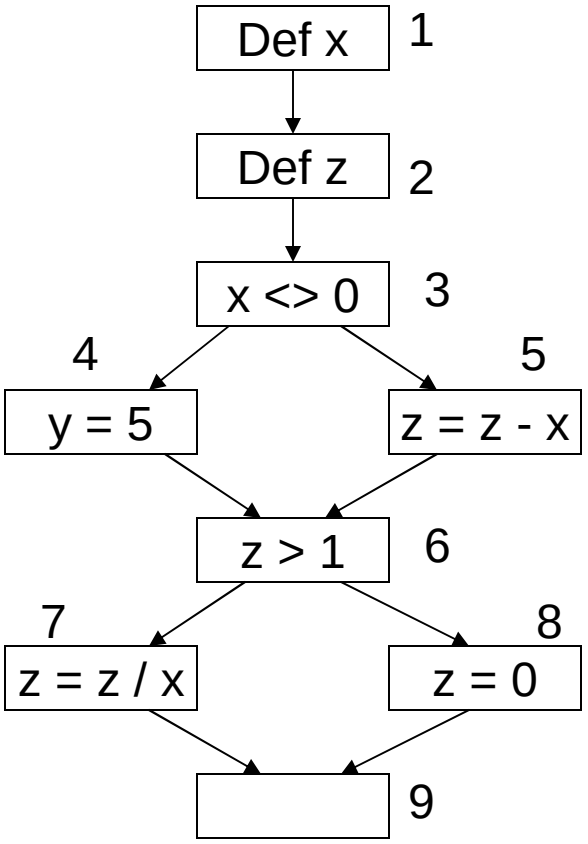
Definición-c-uso que resuelve nuestro problema: (5,7,z)

Esto es válido ya que (5,6,7) es un camino libre-definición

Observar que la definición-uso (2,7,z) no tiene porqué detectar el defecto

Caja Blanca

- Consideremos criterio "Todas las definiciones"



Definición-c-uso que resuelve nuestro problema: (5,7,z)

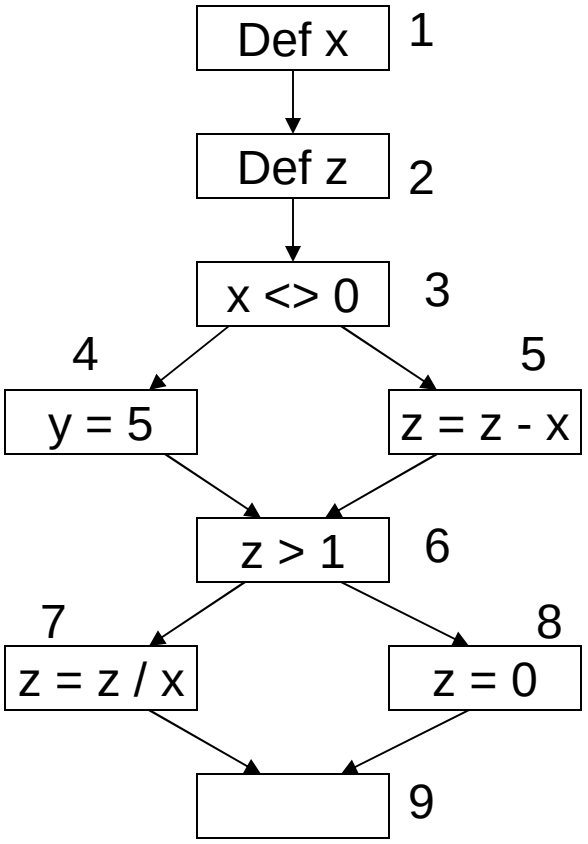
Este criterio nos dice que si consideramos la definición de z en el nodo 5 tenemos que incluir **al menos un** camino libre-definición para **algún** uso (p-uso o c-uso)

Basta con tomar (5,6,z) para cumplir con el criterio para el nodo 5

Este criterio de cubrimiento no nos asegura encontrar el defecto

Caja Blanca

- Consideremos criterio "Todos los c-usos"



Definición-c-uso que resuelve nuestro problema: (5,7,z)

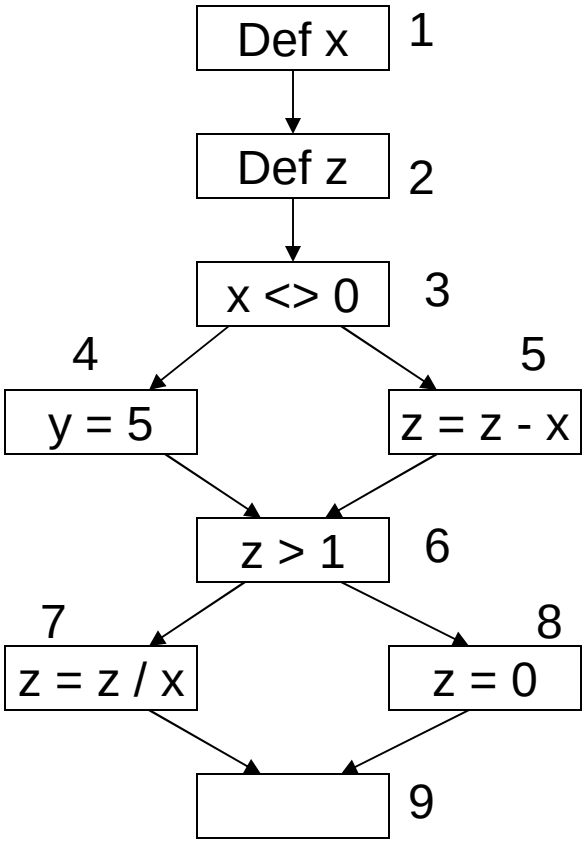
Este criterio nos dice que si consideramos la definición de z en el nodo 5 tenemos que incluir **al menos un** camino libre-definición para **todos** los c-usos

Para cumplir con este criterio a partir de la definición de z en el nodo 5 hay que tomar el siguiente caso: (5,7,z)

Este criterio de cubrimiento nos asegura encontrar el defecto

Caja Blanca

- Como seleccionamos los casos de prueba



Supongamos que queremos cumplir con el criterio de cubrimiento "Todos los c-usos", entonces hay que cumplir con las siguientes definiciones-c-uso (sin considerar variable y):

(1,5,x) (1,7,x) (2,5,z) (2,7,z) (5,7,z)

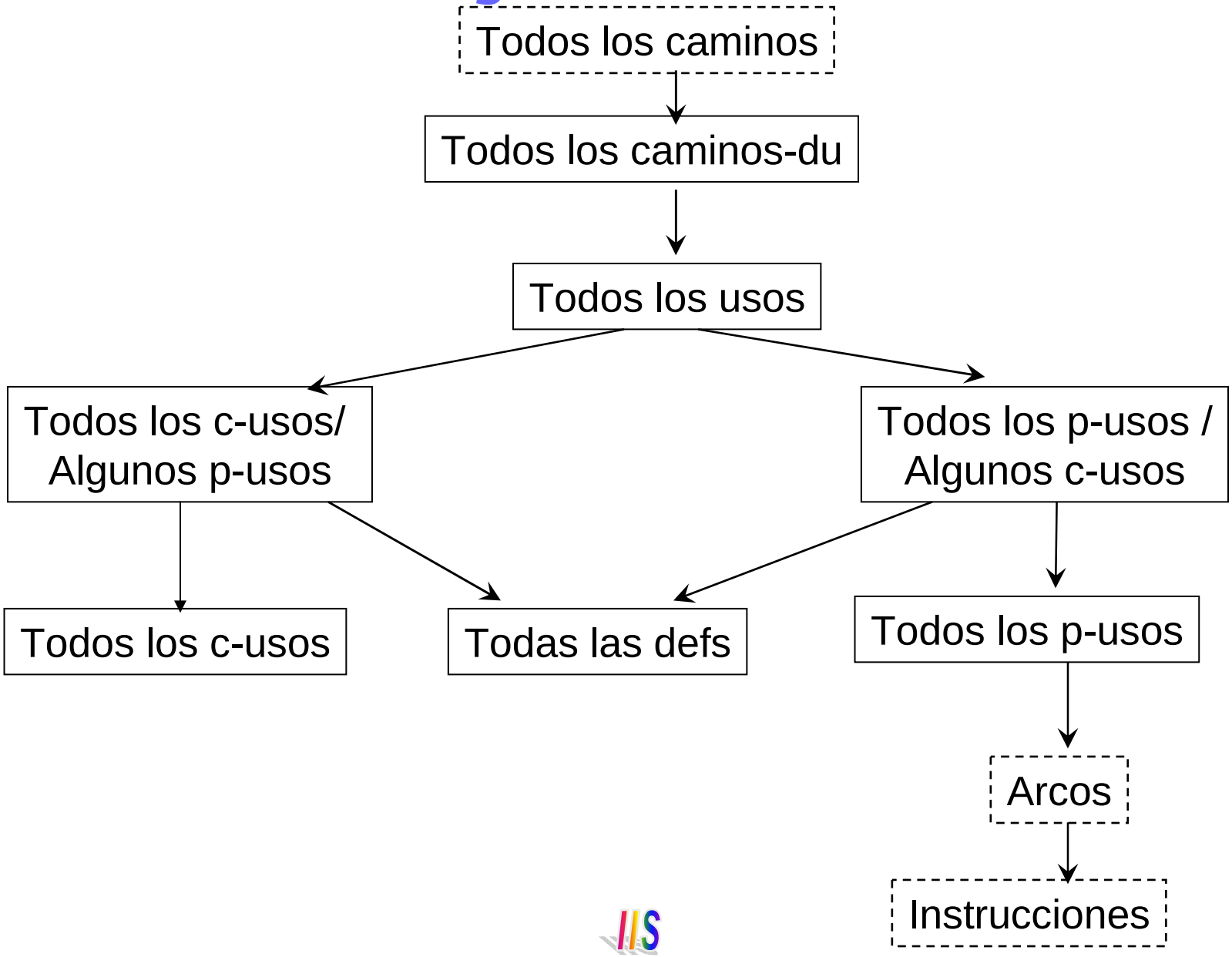
Para todas estas definiciones-c-uso existe al menos un camino libre-definición

"Descubrir" las definiciones-c-uso para cumplir con el criterio es muy complejo. Seleccionar los casos de prueba a partir de estas también es muy complejo → Herramientas automatizadas

Caja Blanca

- El testing basado en flujo de datos es menos usado que el testing basado en el flujo de control
- Es altamente complejo seleccionar los casos de test

Caja Blanca


Todos los caminos
Todos los caminos-du
Todos los usos
Todos los c-usos/
Algunos p-usos
Todos los p-usos /
Algunos c-usos
Todos los c-usos
Todas las defs
Todos los p-usos
Arcos
Instrucciones

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

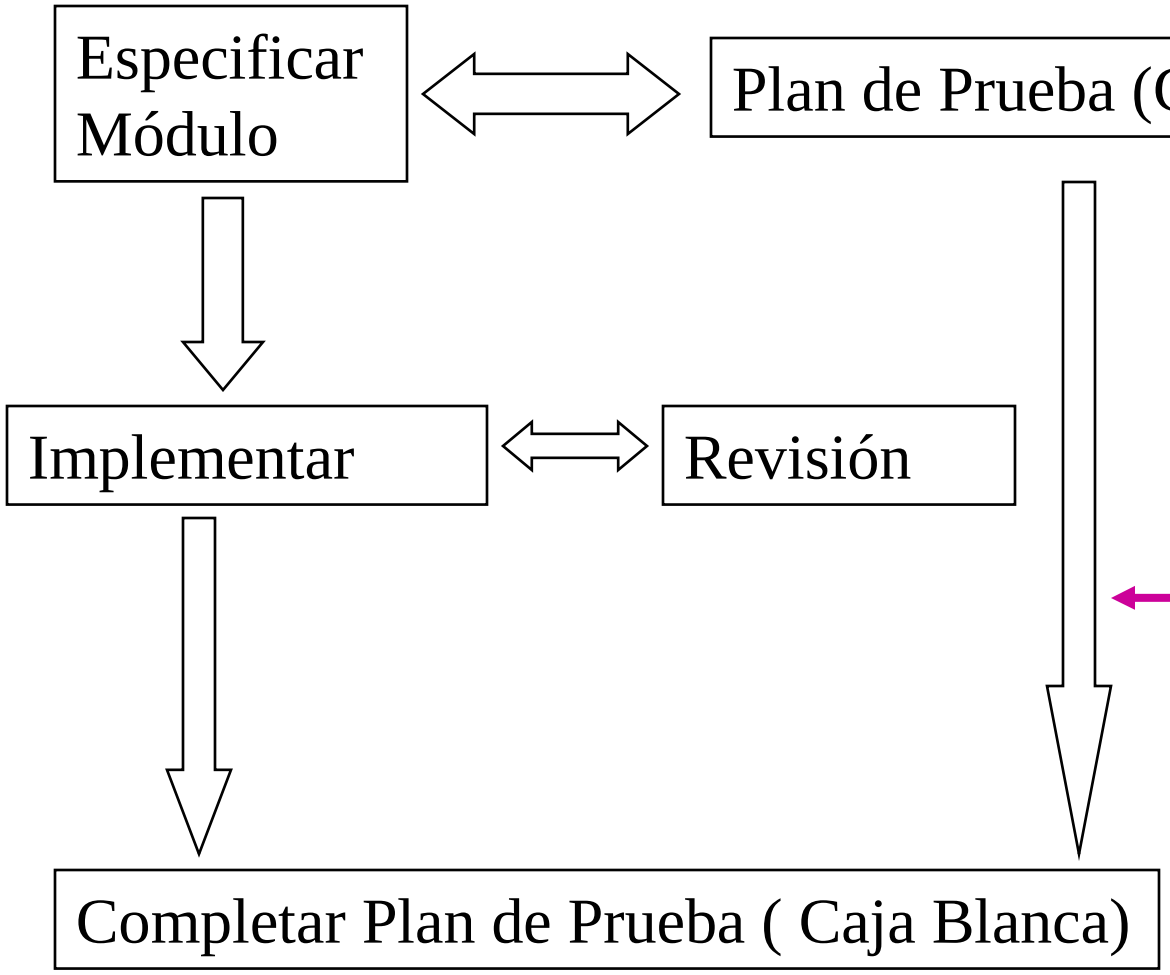
Caja Negra

- 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
  - ✗ Normalmente basadas en un modelo del producto y no en el producto
  - ✗ No se usan para validación (solo verificación)
  - ✗ Dependen de la especificación
  - ✗ No consideran el hardware o el software de base

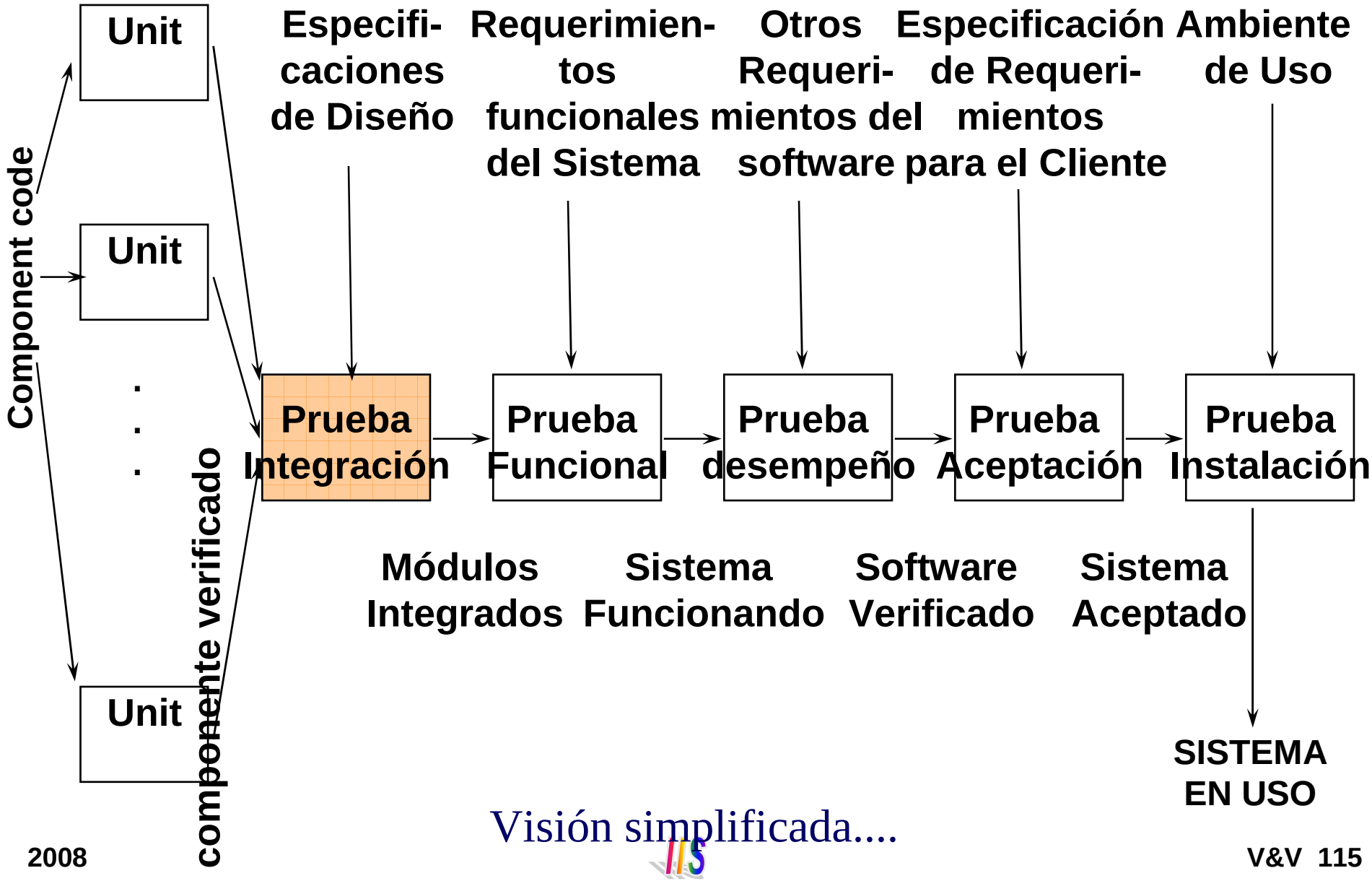
Comparación de las Técnicas

- 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)

Pruebas de Integración

- Temario
  - Pruebas de Módulos
  - Estrategias de Integración
    - Big-Bang
    - Bottom-Up
    - Top-Down
    - Sandwich
    - Por disponibilidad
  - Comparación de Estrategias
  - Builds en Microsoft

Proceso de V&V



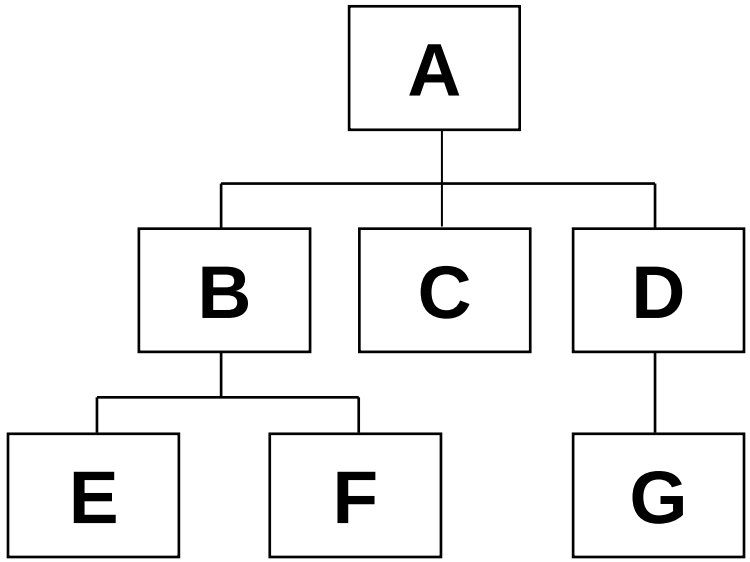
Visión simplificada....

Pruebas de Módulos

El módulo A "usa" a los módulos B, C, D.

El módulo B "usa" a los módulos E y F

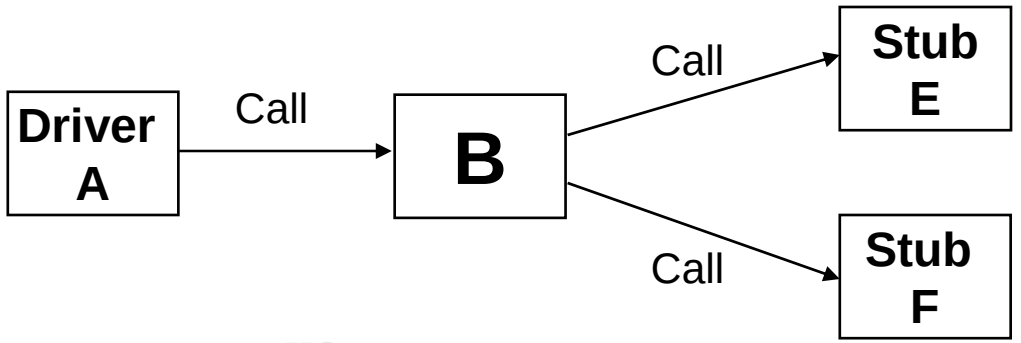
El módulo D "usa" al módulo G



Pruebas de Módulos

- Quiero probar al módulo B de forma aislada – No uso los módulos A, E y F
  - El módulo B es usado por el módulo A
    - Debo simular la llamada del modulo A al B – *Driver*
    - Normalmente el Driver es el que suministra los datos de los casos de prueba
  - El módulo B usa a los módulos E y F
    - Cuando llamo desde B a E o F debo simular la ejecución de estos módulos – *Stub*
  - Se prueba al módulo B con los métodos vistos en pruebas unitarias

Esto quiere decir definir criterios y generar test set que cubran esos criterios



Pruebas de Módulos

- Stub (simula la actividad del componente omitido)
  - Es una pieza de código con los mismos parámetros de entrada y salida que el módulo faltante pero con una alta simplificación del comportamiento. De todas maneras es costoso de realizar
  - Por ejemplo puede producir los resultados esperados leyendo de un archivo, o pidiéndole de forma interactiva a un testeador humano o no hacer nada siempre y cuando esto sea aceptable para el módulo bajo test
  - Si el stub devuelve siempre los mismos valores al módulo que lo llama es probable que esté módulo no sea testeado adecuadamente. Ejemplo: el Stub E siempre devuelve el mismo valor cada vez que B lo llama → Es probable que B no sea testeado de forma adecuada

Pruebas de Módulos

- Driver
  - Pieza de código que simula el uso (por otro módulo) del módulo que está siendo testeado. Es menos costoso de realizar que un stub
  - Puede leer los datos necesarios para llamar al módulo bajo test desde un archivo, GUI, etc
  - Normalmente es el que suministra los casos de prueba al módulo que está siendo testeado

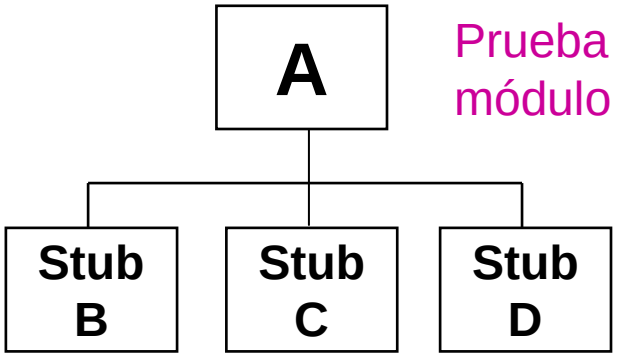
Estrategias de Integración

- No incremental
  - Big-Bang
- Incrementales
  - Bottom-Up (Ascendente)
  - Top-Down (Descendente)
  - Sandwich (Intercalada)
  - Por disponibilidad

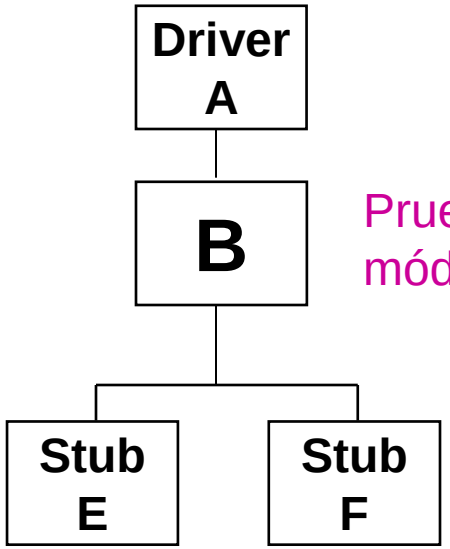
- El objetivo es lograr combinar módulos o componentes individuales para que trabajen correctamente de forma conjunta

Big-Bang

- Se prueba cada módulo de forma aislada y luego se prueba la combinación de todos los módulos a la vez



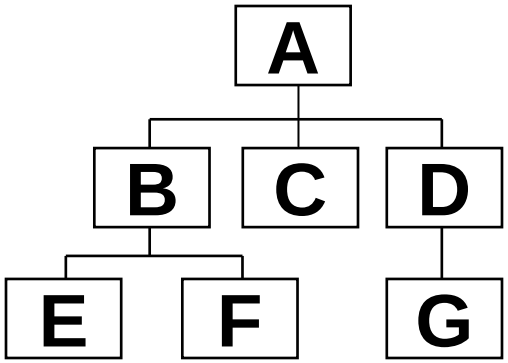
Prueba del módulo A



Prueba del módulo B

Se prueban los otros módulos de manera similar: C, D, E, F y G

No se hacen integraciones parciales



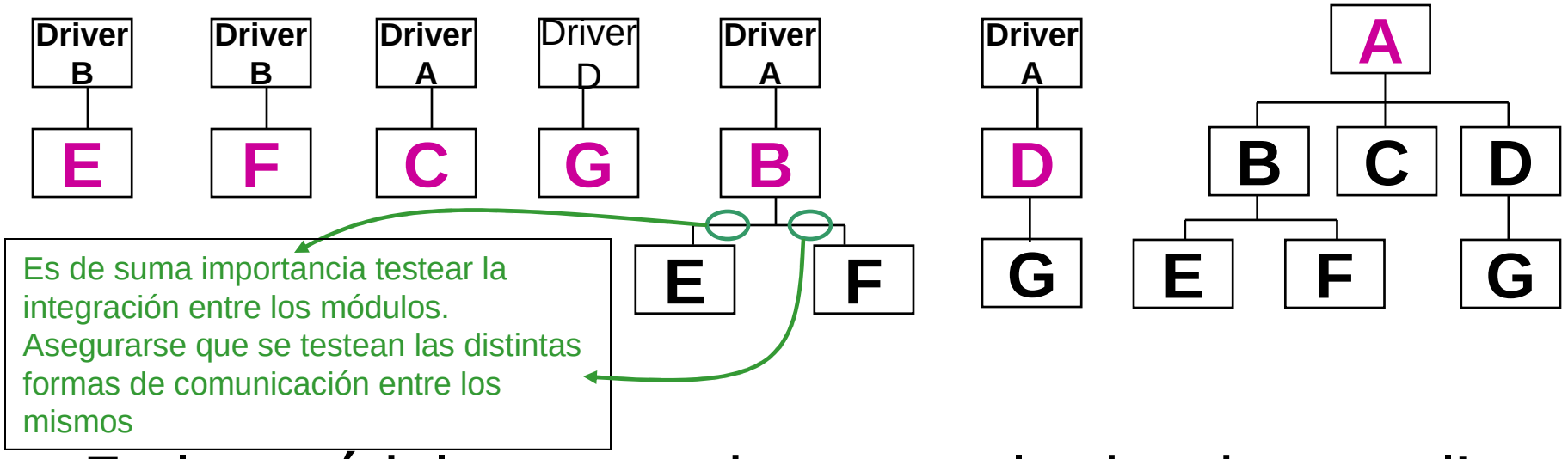
Después se integran todos los módulos a la vez y se prueba el funcionamiento en conjunto

Big-Bang

- Pros y contras
  - ✓ Existen buenas oportunidades para realizar actividades en paralelo (todos los módulos pueden ser probados a la vez)
  - ✗ Se necesitan stubs y drivers para cada uno de los módulos a ser testeados ya que cada uno se testea de forma individual
  - ✗ Es difícil ver cuál es el origen de la falla ya que integré todos los módulos a la vez
  - ✗ No se puede empezar la integración con pocos módulos
  - ✗ Los defectos entre las interfaces de los módulos no se pueden distinguir fácilmente de otros defectos
  - ✗ No es recomendable para sistemas grandes

Bottom-Up

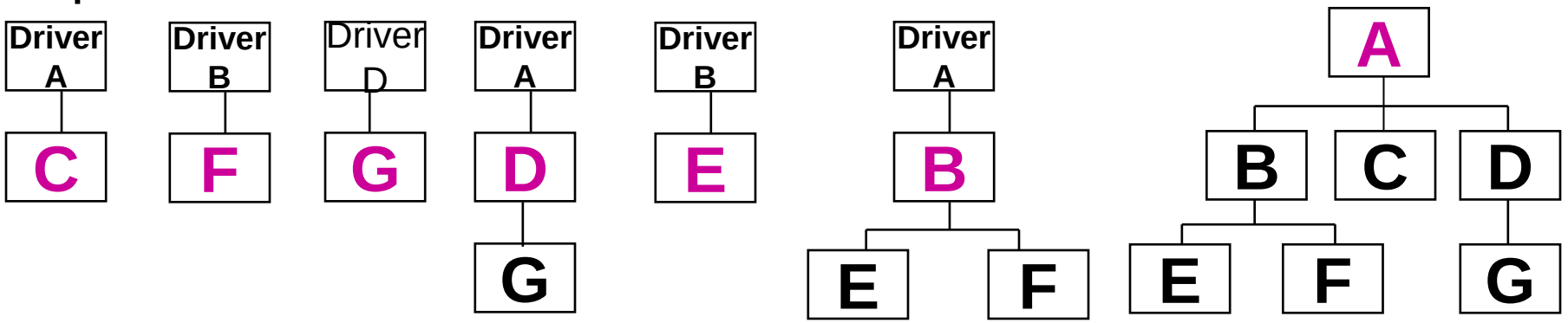
- Comienza por los módulos que no requieren de ningún otro para ejecutar
- Se sigue hacia arriba según la jerarquía "usa"
- Requiere de *Drivers* pero no de *Stubs*



- En los módulos marcados con color bordo se aplican métodos de prueba unitaria

Bottom-Up

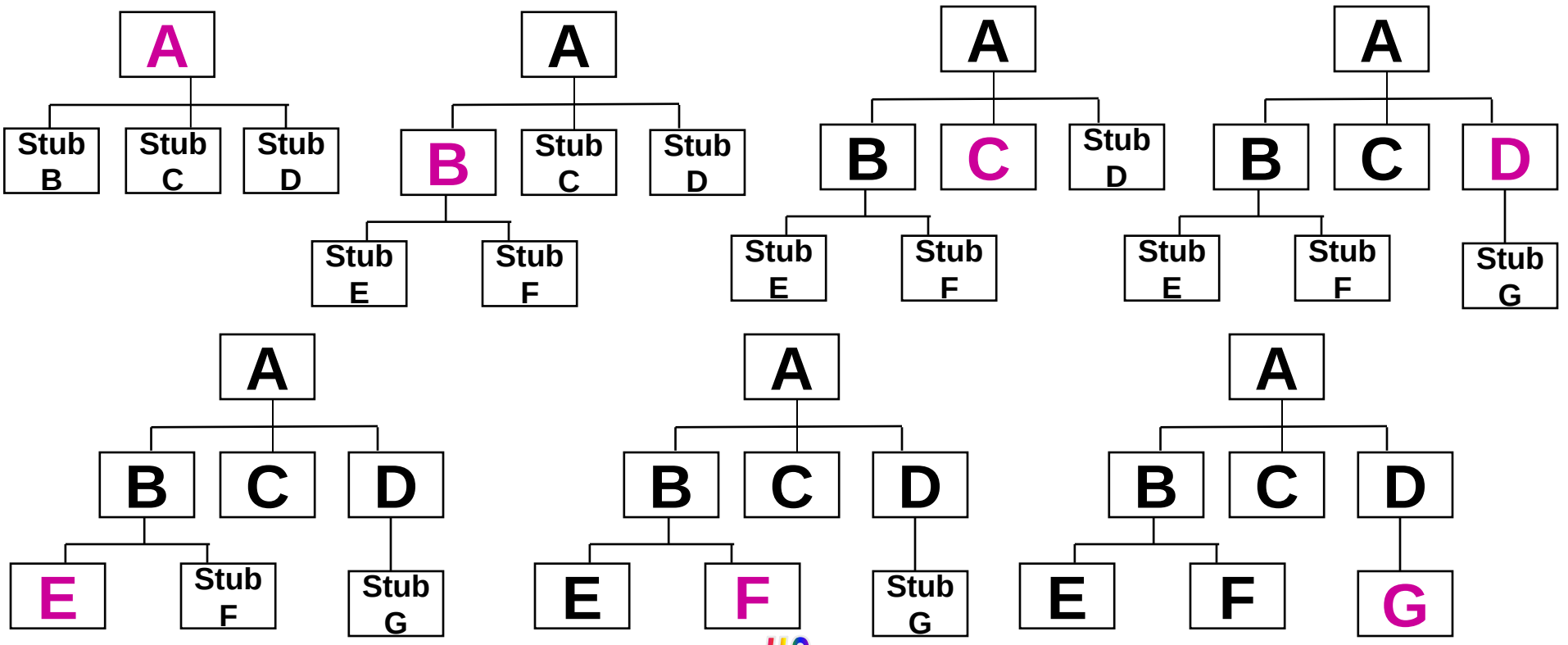
- La regla general indica que para probar un módulo todos los de "abajo" del módulo a probar deben estar probados
- Supongamos que C, F y D son módulos críticos (módulo complicado, con un algoritmo nuevo o con posibilidad alta de contener errores, etc) entonces es bueno probarlos lo antes posible



- Hay que considerar que hay que diseñar e implementar de forma tal que los módulos críticos estén disponible lo antes posible. Es importante la planificación

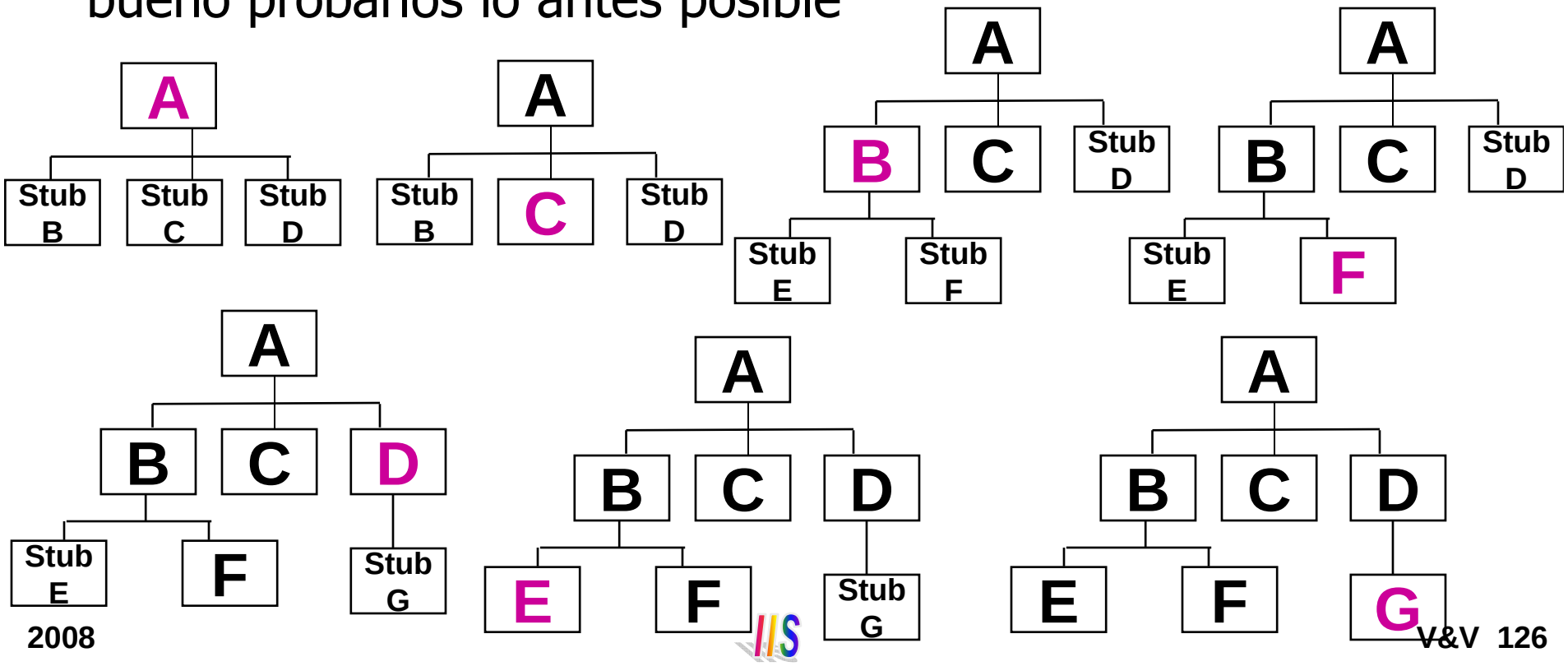
Top-Down

- Comienza por el módulo superior de la jerarquía usa
- Se sigue hacia abajo según la jerarquía "usa"
- Requiere de *Stubs* pero no de *Drivers*



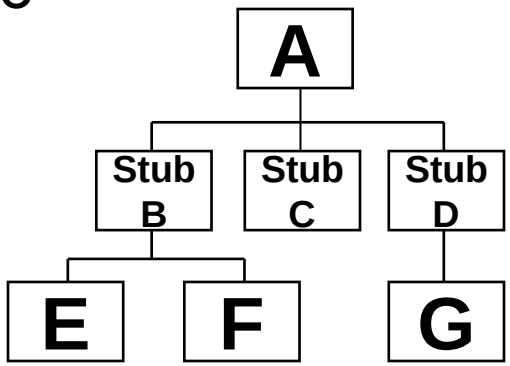
Top-Down

- La regla general indica que para probar un módulo todos los de "arriba" (respecto al módulo a probar) deben estar probados
- Supongamos que C, F y D son módulos críticos entonces es bueno probarlos lo antes posible



Otras Técnicas

- Sandwich
  - Usa Top-Down y Bottom-Up. Busca probar los módulos más críticos primero



- Por Disponibilidad
  - Se integran los módulos a medida de que estos están disponibles

Comparación de las Estrategias

- No incremental respecto a incremental
  - Requiere mayor trabajo. Se deben codificar mas Stubs y Drivers que en las pruebas incrementales
  - Se detectan más tardíamente los errores de interfaces entre los módulos ya que estos se integran al final
  - Se detectan más tardíamente las suposiciones incorrectas respecto a los módulos ya que estos se integran al final
  - Es más difícil encontrar la falta que provocó la falla. En la prueba incremental es muy factible que el defecto esté asociado con el módulo recientemente integrado (en sí mismo) o en interfaces con los otros módulos
  - Los módulos se prueban menos. En la incremental vuelvo a probar indirectamente a los módulos ya probados.

Comparación de las Estrategias

- Top-down (focalizando en OO)
  - ✓ Resulta fácil la representación de casos de prueba reales ya que los módulos superiores tienden a ser GUI
  - ✓ Permite hacer demostraciones tempranas del producto
  - ✗ Los stubs resultan muy complejos
  - ✗ Las condiciones de prueba pueden ser imposibles o muy difíciles de crear (pensar en casos de prueba inválidos)
- Bottom-Up (focalizando en OO)
  - ✓ Las condiciones de las pruebas son más fáciles de crear
  - ✓ Al no usar stubs se simplifican las pruebas
  - ✗ El programa como entidad no existe hasta no ser agregado el último módulo

Builds en Microsoft

- Iteración: diseño, construcción, prueba

(clientes involucrados en proceso de prueba)

- Equipos de tres a ocho desarrolladores
- Diferentes equipos responsables por distintas características (features)
- Autonomía relativa de cada equipo sobre la especificación de las características
- Descomposición en características

Enfoque Sincronizar y Estabilizar

Hito 1: características más críticas y componentes compartidos
Diseño, codificación, prototipación / Verificación de Usabilidad
Builds diarios / Integración de Características
Eliminar faltas severas

Hito 2: características deseables
Diseño, codificación, prototipación / Verificación de Usabilidad
Builds diarios / Integración de Características

Hito 3: características menos críticas
Diseño, codificación, prototipación / Verificación de Usabilidad
Builds diarios / Integración de Características Completas
Liberación a Producción

Pruebas de Sistemas OO

- Temario
  - Características
  - Algunos problemas
  - "Object-Oriented programs and testing" – Perry y Kaiser
  - Estrategia N+

Características

- Normalmente son sistemas más complicados de probar (testear)
- Características que lo diferencian de otros sistemas
  - Encapsulación de la información
  - Las clases tienen estado y comportamiento. El estado afecta al comportamiento y el comportamiento afecta al estado. Esto no es común en lenguajes procedurales
  - Las unidades son más pequeñas (clases, objetos)
  - Los métodos son más pequeños
  - Un sistema OO tiene muchas más integraciones que uno procedural
  - Herencia, polimorfismo y dynamic binding

Algunos Problemas

- Debido a la encapsulación de la información
  - Si tengo que testear un método multX (multiplica al atributo x por el pasado en el método) de una clase A y ese método cambia el valor del atributo x de la clase A, ¿Cómo sé si lo cambió correctamente? ¿Es válido confiar en el método getX de la clase A?

```
Class A {
   int x;
   public A(int px) {
      x = px
   }
   public void multX(int y) {
      x = (x * y) - 1;
   }
   public int getX(){
      return x + 1;
   }
}
```

```
Class PruebaMetX {
   main {
      a = new A(5);
      a.multX(2);
      int res = a.getX();
      if res = 10 pruebaOK
      else        pruebaFail
   }
}
```

En este caso la prueba pasa (resultado pruebaOK), sin embargo el valor de x en el objeto a es 9 y este es un valor incorrecto

Algunos Problemas

- Debido a la encapsulación de la información
  - Si la clase está debidamente testeada se creía (en los primeros tiempos de la OO) que podría ser usada en cualquier otro sistema comportándose de manera correcta. Más adelante veremos que esto no es como se pensaba.
  - En definitiva, una gran bondad de la OO que es la reutilización no se pierde, **sin embargo, hay que volver a testear la integración (esto es debido al cambio de contexto)**

Algunos Problemas

- Debido al estado y comportamiento de los objetos
  - Cuando un objeto recibe un mensaje no siempre actúa de la misma manera sino que depende del estado en que este se encuentre (el estado es el valor de todos los atributos del objeto; incluyendo otros objetos)
  - Cuando un objeto termina de ejecutar el método, puede ser que su estado haya cambiado
  - Esto trae aparejado que los métodos convencionales de testing no pueden ser aplicados tal cual. Existen métodos que usan diagramas de estado de clases para realizar pruebas de objetos (vamos a ver uno más adelante, estrategia N+)

Algunos Problemas

- Unidades más pequeñas, métodos más pequeños y mayor integración que lenguajes procedurales
  - Si bien los objetos manejan estados y esto hace distintas las pruebas respecto a las tradicionales, como los métodos son más pequeños es más fácil realizar pruebas de caja blanca en OO que en lenguajes procedurales
  - Por otro lado la cantidad de objetos que interactúan entre sí es mucho más grande que los módulos que interactúan en un lenguaje procedural
  - Se considera que la prueba de unidad es más sencilla en sistemas OO pero las pruebas de integración son más extensas y tienden a ser más complejas

Algunos Problemas

- Debido a herencia, polimorfismo y dynamic binding
  - Cuando una superclase cambia un método hay que testear la subclase ya que puede haber introducido inconsistencias en está última. Esto es sabido desde "siempre"
  - Se creía que cuando una clase heredaba de otra que ya estaba adecuadamente testeada, los métodos heredados no debían ser testeados. Mostraremos más adelante que este concepto esta equivocado

OO Programs and Testing
E. Perry y G. Kaiser

- "Tiran abajo" algunos mitos sobre las pruebas de sistemas Orientados a Objetos
- Para hacerlo se basan en los axiomas de J. Weyuker: "Axiomatizing Software Test Data Adequacy"
- Definición
  - Un programa está *adecuadamente testeado* si a sido cubierto de acuerdo al criterio seleccionado (el criterio puede ser tanto estructural como funcional)

OO Programs and Testing
E. Perry y G. Kaiser

- Axiomas de J. Weyuker
  - *Applicability*
    - Para cada programa existe un test set adecuado
  - *Non-Exhaustive Applicability*
    - Para un programa P existe un test set T tal que P es adecuadamente testeado por T, y T no es un test exhaustivo
  - *Monotonicity*
    - Si T es adecuado para P y T es un subconjunto de T' entonces T' es adecuado para P
  - *Inadequate empty set*
    - El conjunto vacío no es un test set adecuado para ningún programa

OO Programs and Testing
E. Perry y G. Kaiser

- Axiomas de J. Weyuker
  - *Renaming*
    - Sea P un renombre de Q entonces T es adecuado para P si y solo si T es adecuado para Q
    - Un programa P es un renombre de Q si P es idéntico a Q excepto que todas las instancias de un identificador x de Q ha sido reemplazado en P por un identificador y, donde y no aparece en Q. O si hay un conjunto de estos renombres
  - *Complexity*
    - Para todo n > 0, hay un programa P, tal que P es adecuadamente testeado por un test set de tamaño n pero no por un test set de tamaño n – 1
  - *Statement coverage*
    - Si T es adecuado para P entonces T causa que toda sentencia ejecutable de P sea ejecutada

OO Programs and Testing
E. Perry y G. Kaiser

- Axiomas de J. Weyuker
  - *Antiextensionality*
    - Si dos programas computan la misma función, un test set adecuado para uno no es necesariamente adecuado para el otro
  - *General Multiple Change*
    - Cuando dos programas son sintácticamente similares (tienen la misma forma) usualmente requieren test set diferentes
    - Existen programas P y Q que tienen la misma forma y un test set T que es adecuado para P, pero T no es adecuado para Q
    - Dos programas son de la misma forma si uno puede ser transformado en el otro aplicando las siguientes reglas
      - Reemplazar el operador r1 por el r2
      - Reemplazar la constante c1 por la c2

OO Programs and Testing
E. Perry y G. Kaiser

- Axiomas de J. Weyuker
  - *Antidecomposition*
    - Testear un componente en el contexto de un programa puede ser adecuado respecto a ese programa pero no necesariamente respecto a otros usos de esa componente
    - Se testea adecuadamente el programa P con el test set T pero el test set T' que es la entrada a Q desde P no es adecuado para Q
  - *Anticomposition*
    - Testear adecuadamente cada componente de forma aislada no necesariamente testea adecuadamente todo el programa. La integración de dos componentes resulta en interacciones que no pueden surgir en aislamiento

OO Programs and Testing
E. Perry y G. Kaiser

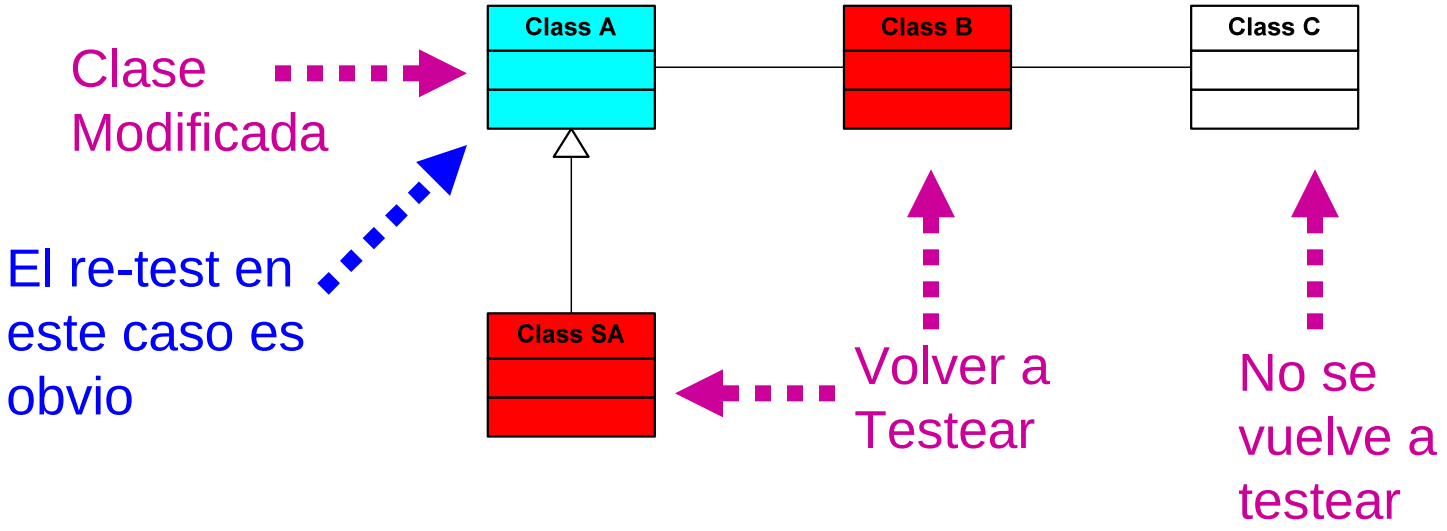
- Axiomas de J. Weyuker
  - Supongamos que P tiene p caminos y que Q tiene q caminos con q<p. Puedo tener un test set T de cardinalidad p que cubra todos los caminos de P y las entradas de las llamadas a Q cubran todos los caminos de Q. Sin embargo los cantidad de caminos de la composición PQ puede ser tan larga como p x q. Como T es de cardinalidad p no sería adecuado para PQ. (el criterio que se esta considerando es el de cubrimiento de caminos)

OO Programs and Testing
E. Perry y G. Kaiser

- Encapsulación en clases
- Mito:
  - Si mantenemos las interfaces de una clase sin cambiar, y cambiamos la implementación interna manteniendo la misma funcionalidad, no hay que volver a testear las clases que usan la clase modificada
- Realidad:
  - El axioma *anticomposition* nos recuerda la necesidad de volver a testear también todas las unidades dependientes porque un programa que fue testeado adecuadamente de forma aislada puede no estar adecuadamente testeado en combinación. Esto significa que además del testeo de unidad que hay que realizar en la clase modificada también hay que realizar testeo de integración

OO Programs and Testing
E. Perry y G. Kaiser

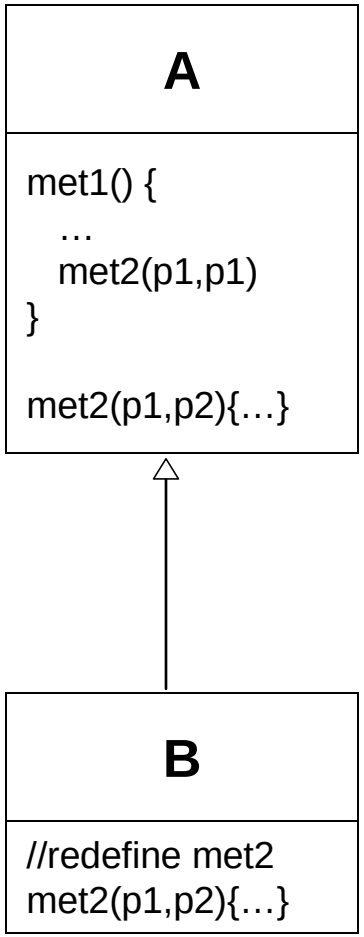
- Encapsulación en clases
  - Cuando modifico una clase se debe volver a testear esa clase y todas las que se relacionan explícitamente con esta



OO Programs and Testing
E. Perry y G. Kaiser

- Subclases nuevas o modificaciones de subclases
- Mito:
  - No se deben testear los métodos heredados de la superclase ya que fueron testeados en esta
- Realidad:
  - El axioma *antidecomposition* nos dice lo contrario. El uso de subclases agrega esta inesperada forma de dependencia porque provee un nuevo contexto para las componentes heredadas.

OO Programs and Testing
E. Perry y G. Kaiser



El método met2 se redefine en la clase B. Esto genera un nuevo contexto y es por esto que se debe volver a testear met1 para la clase B, debido a que met1 llama a met2

OO Programs and Testing
E. Perry y G. Kaiser

- Cuando la subclase es una extensión pura de la superclase entonces no es necesario volver a testear.
- Definición de extensión pura
  - Las nuevas variables creadas por la subclase así como los nuevos métodos no interactúan en ninguna dirección con las variables heredadas ni con los métodos heredados

OO Programs and Testing
E. Perry y G. Kaiser

- Sobre-escritura de métodos
- Mito:
  - Al sobre-escribir un método heredado se puede testear adecuadamente la subclase con un test set adecuado para la superclase
- Realidad:
  - Aunque muy probablemente los dos métodos (el de la superclase y el sobre-escrito en la subclase) computen una función semánticamente similar un test set adecuado para uno no tiene porque ser adecuado para el otro. Esto se deduce a partir del axioma *antiextensionality*

OO Programs and Testing
E. Perry y G. Kaiser

- Conclusión más general
  - Se revoca la intuición que antes se tenía de que la encapsulación y la herencia iban a reducir los problemas del testing

Testeo de Unidad en OO

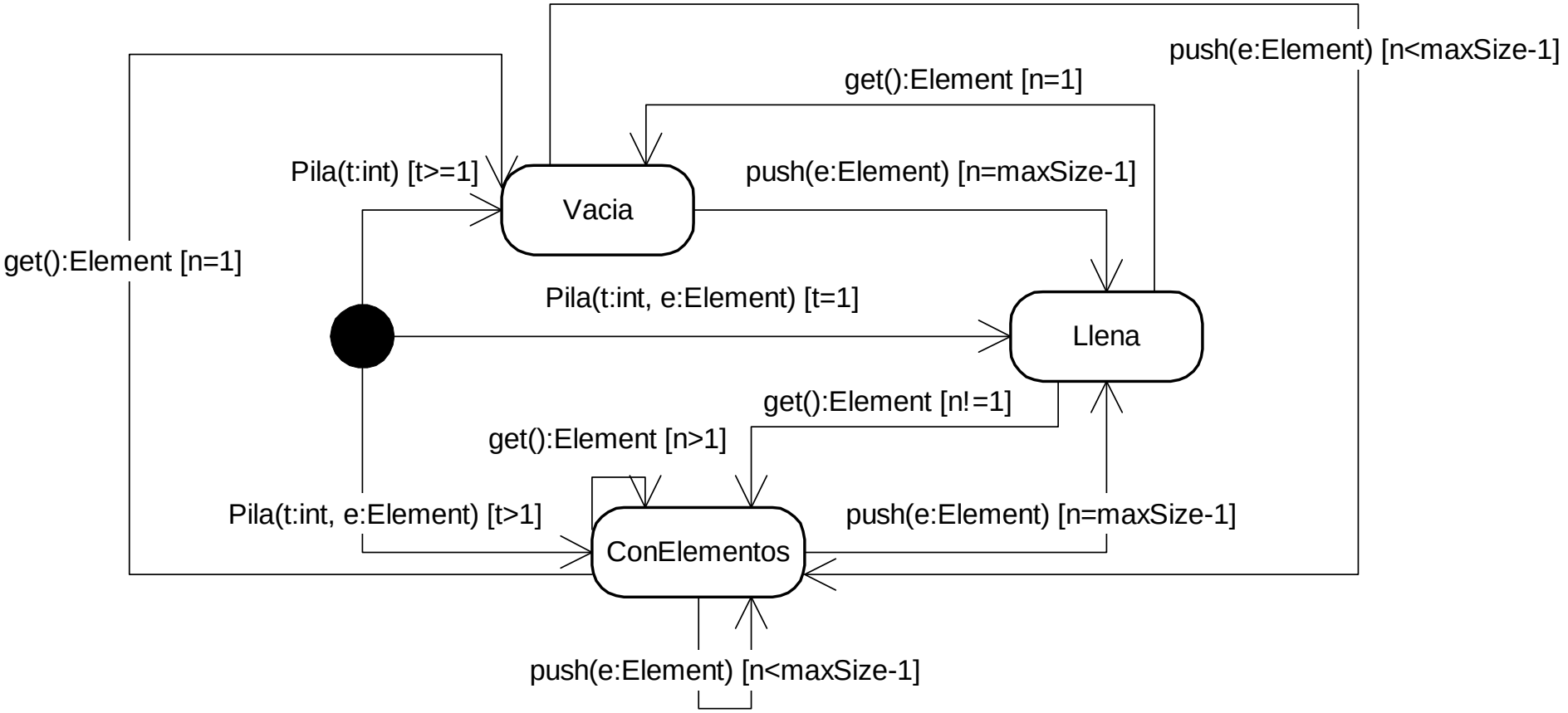
- Es un acuerdo casi universal que el **objeto** es la **unidad** básica para el diseño de casos de prueba
- Otras **unidades** para testear son agregaciones de clases: class cluster (normalmente patterns de diseño) y pequeños subsistemas
- El testing de sistemas orientados a objetos difiere en algunos aspectos del testing de sistemas estructurados

Estrategia N+

- Como mencionamos hay clases que son estado-dependientes
  - Ejemplo: Lista o Pila
    - Estados: Vacía, ConElementos, Llena
  - Métodos de la clase Pila
    - Pila(t:int) / pre: t>1 ; post: maxSize=t@pre y n=0
    - Pila(t:int,e:Elemento) / pre: t>1 ; post: maxSize=t@pre y e pertenece a la pila como último elemento insertado y n=1
    - push(e:Element) / pre: n!=maxSize ; post: n=n@pre+1 y e pertenece a la pila como último elemento insertado
    - get():Element / pre: n>0 ; post: n=n@pre-1 y el último elemento insertado no está más en la pila y result=al último elemento insertado
  - Estos métodos los sabemos testear (caja negra o blanca)

Estrategia N+

- Diagrama de estado de la clase Pila



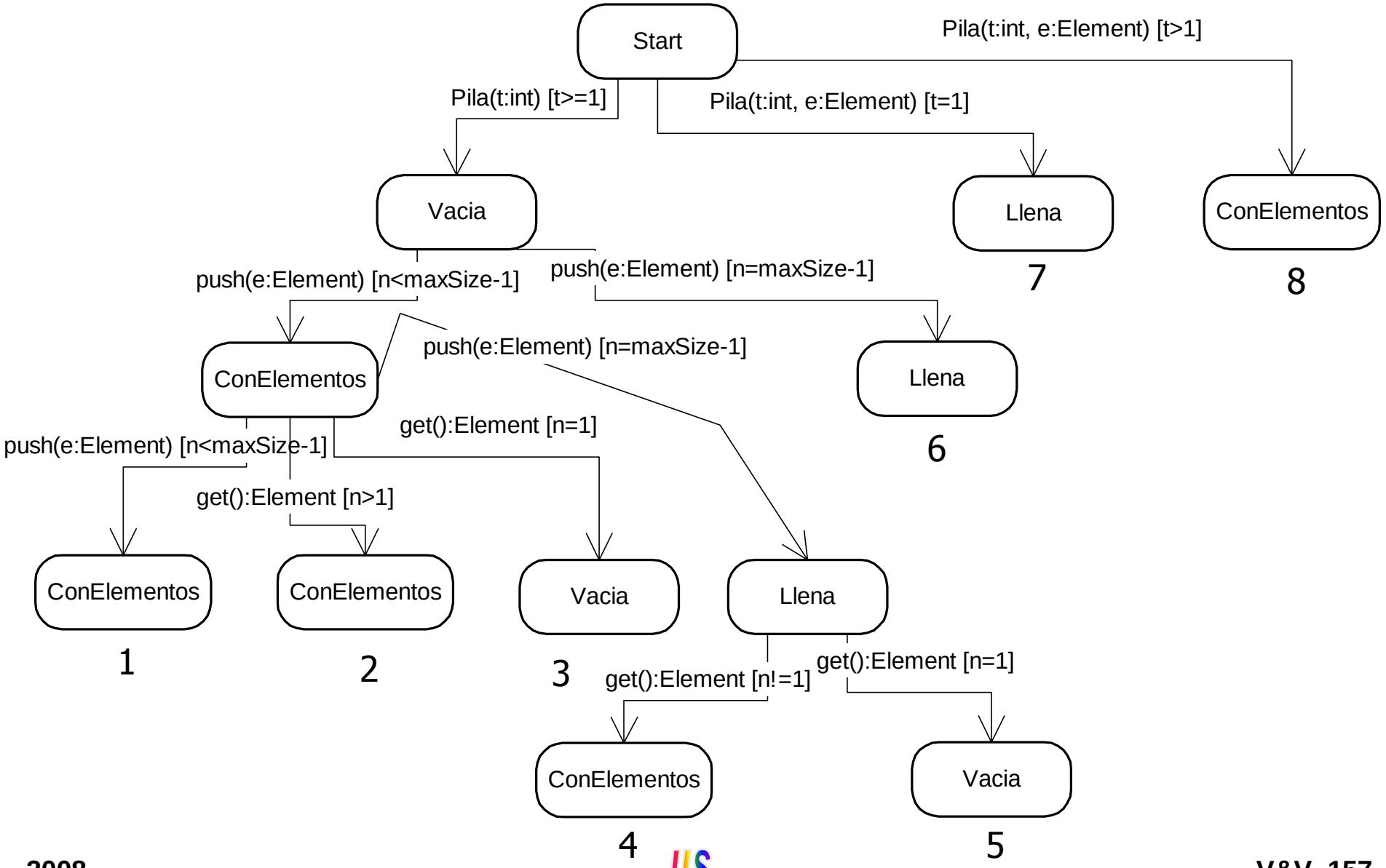
Estrategia N+

- La estrategia N+ es una estrategia propuesta por Binder y se divide en dos partes
  - All round-trip path
  - Sneak paths
- All round-trip path
  - Los casos de prueba deben cubrir:
    - Todos los caminos simples desde el estado inicial al final
    - y todos los caminos que empiezan y terminan en un mismo estado
- Sneak paths
  - Los casos de prueba deben cubrir:
    - Todos los mensajes no esperados en cierto estado (ejemplo: un push en el estado lleno)

Estrategia N+

- All round-trip path
  - Primero construimos el "árbol de transición" haciendo una recorrida en profundidad (también puede ser en amplitud) del diagrama de estados
  - La recorrida de un camino se detiene siempre que el estado al que se llega ya está presente en el árbol
  - En el caso de guardas con varios conectores se agregan transiciones. No vamos a profundizar en este tema
  - Luego para cada camino desde el nodo inicial hasta el final se realiza un caso de prueba (esto asegura cumplir con el criterio All round-trip path)
    - TODOS LOS CASOS **JUNTOS** CUBREN ALL ROUND TRIP PATH

Estrategia N+



Estrategia N+

- Un caso de prueba:

```
Pila p = new Pila(2);
Chequear poscondiciones del método Pila
Chequear estado de la clase = Vacia
Element e = new Element();
p.push(e);
Chequear poscondiciones del método push
Chequear estado de la clase = ConElementos
p.push(e);
Chequear poscondiciones del método push
Chequear estado de la clase = Llena
Element z = p.get();
Chequear poscondiciones del método get
Chequear estado de la clase = ConElementos
```

Estrategia N+

- Cada estado debe estar definido en función de restricciones respecto a los miembros de la clase
- A partir de estas restricciones se puede chequear en que estado está la clase
- Otra forma es proveer métodos isEstate() que devuelven true si la clase se encuentra en el estado y false en caso contrario. Estos métodos chequean las restricciones

Estrategia N+

- Sneak paths
  - Pongo a la clase en un estado y le mando mensajes no válidos
  - Ejemplos:
    - La clase está en el estado Llena y le mando mensaje push(e)
    - La clase está en el estado Vacía y le mando un mensaje get()
  - Estos test complementan los realizados por All round-trip path

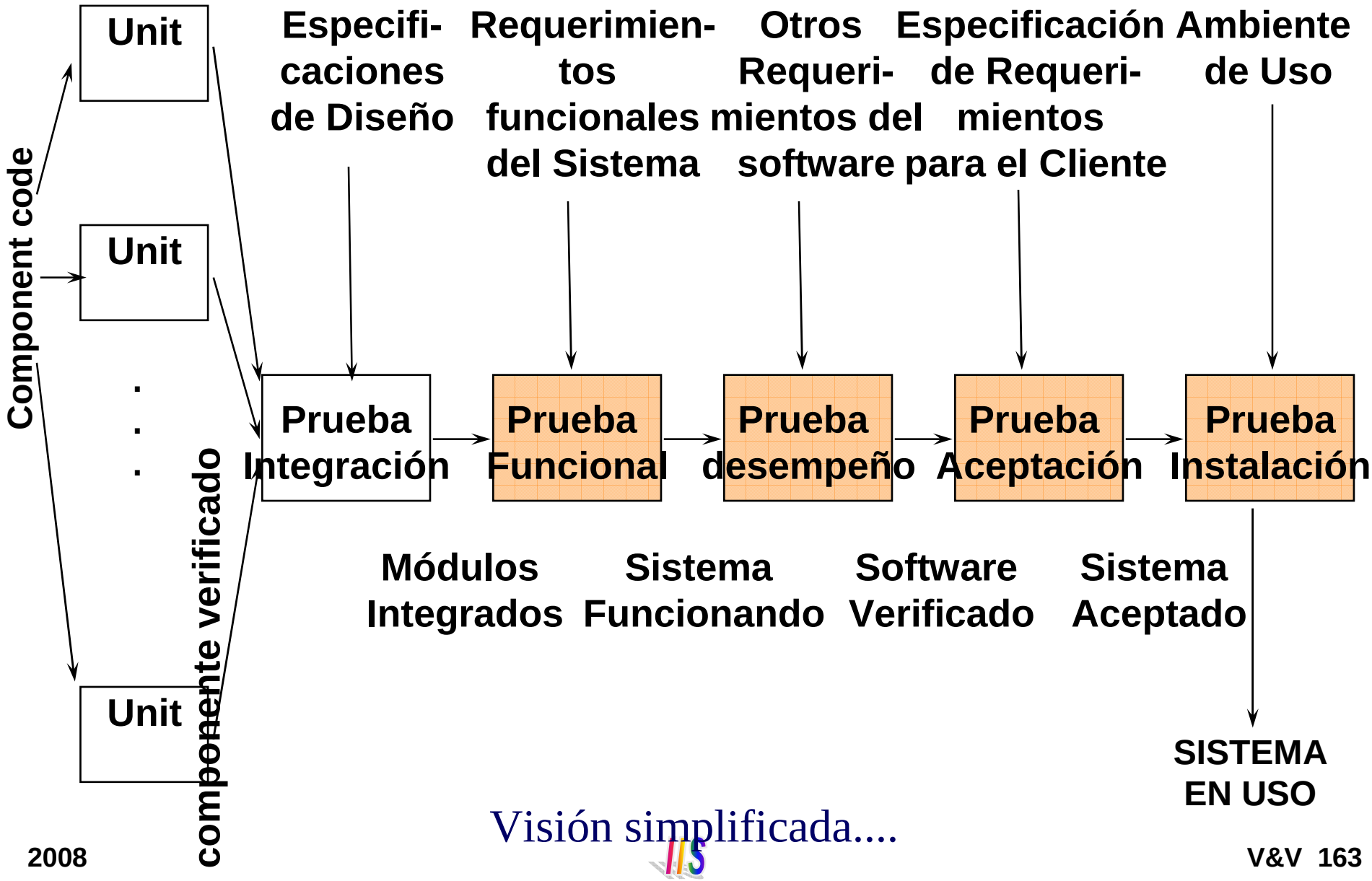
Tipos de Testing OO

- Propuesta muy común:
  - Testing de métodos individuales
    - Dada la definición funcional de un método se testea según técnicas que ya vimos
  - Testing intraclase
    - Se busca testear la relación entre los métodos de una clase. Por ejemplo, estrategia N+. Existen otras formas
  - Testing interclases
    - Similar al test de integración solo que como ya se comento es más complejo por la cantidad de interacciones
  - Testing funcional
  - Etc...
- Los dos primeros se "corresponden" al testing unitario

Pruebas del Sistema

- Temario
  - Prueba Funcional
    - A partir de casos de uso
  - Prueba de Desempeño
  - Prueba de Aceptación
  - Prueba de Instalación

Proceso de V&V



Visión simplificada....

Las Distintas Pruebas del Sist.

- Prueba de función
  - Verifica que el sistema integrado realiza sus funciones especificadas en los requerimientos
- Prueba de desempeño
  - Verifica los requerimientos no funcionales del sistema
- Prueba de aceptación
  - Se realiza junto con el cliente. Busca asegurar que el sistema es el que el cliente quiere
- Prueba de instalación
  - Se realizan las pruebas en el ambiente objetivo

Prueba de Función

- La idea es probar las distintas funcionalidades del sistema
  - Se prueba cada funcionalidad de forma individual
    - Esto puede ser testeo de casos de uso
  - Se prueban distintas combinaciones de funcionalidades
    - Ciclos de vida de entidades (alta cliente, modificación del cliente, baja del cliente)
    - Procesos de la organización (pedido de compra, gestión de la compra, envió y actualización de stock)
    - Esto se puede ver como testeo de ciclos de casos de usos
- El enfoque es más bien de caja negra
- Está prueba está basada en los requerimientos funcionales del sistema

Prueba de Función – Use Case

- Se hace la prueba funcional a partir de los casos de uso
- Se identifican los distintos escenarios posibles y se usan como base para las condiciones de prueba
- Se completan las condiciones en función de los tipos de datos de entrada y de salida
- A partir de las condiciones se crean los Casos de Prueba

Caso de Uso: Retiro

Precondición: el cliente ingresó una tarjeta válida e ingresó nro. de PIN

Flujo Principal:

1. El CA despliega las distintas alternativas disponibles,el Cliente elige Retiro
2. El CA pide cuenta y monto y el Cliente los elige (o ingresa)
3. CA envía Id. Tarjeta, PIN, cuenta y monto
4. SC (Servicio de Cajeros) contesta: Continuar (OK) o No Continuar
5. CA dispensa el dinero
6. CA devuelve la tarjeta
7. CA imprime el recibo

4A. Importe inválido

Si el monto indicado por el cliente no puede obtenerse a partir de los billetes de que dispone el CA, este despliega un mensaje de advertencia y le solicita que ingrese nuevamente el importe. No hay límite de repeticiones.

4B. No hay suficiente saldo en la cuenta.

4B1. CA despliega mensaje al Cliente y devuelve la tarjeta (no se imprime recibo)

Caso de Uso: Retiro

Instancias de un Casos de Uso

1. Flujo Principal
2. Flujo Principal- Importe inválido (n)
3. Flujo Principal- No hay suficiente saldo en cuenta

Condiciones de Prueba

Condiciones	Tipo de Resultado	Caso
1. Normal	Retiro exitoso	
2. Normal-Importe inválido (n)	Retiro exitoso	
3. Normal- sin saldo	No exitoso	

Condiciones de Prueba

Condiciones	Tipo de Resultado	Caso
1. Normal	Retiro exitoso	
1.1 Normal Caja de Ahorros		
1.2 Normal Cuenta Corriente		
2. Normal-Importe inválido (n)	Retiro exitoso	
3. Normal- sin saldo	No exitoso	

Casos de Prueba

\Caso	A1	A2	A3		A4
Entradas					
Tarjeta	13131	13131	13131		23232
PIN	9001	9001	9001		9002
Cuenta	CA128	CC321	CC321		CA228
Monto	100	500	12	1200	100
Salidas			Importe inválido		Sin saldo
Dispensa	$100	$500		$1200	
Dev.tarjeta	Sí	Sí		Sí	Sí
Recibo	13131...	13131...		13131...	
Condics.	1.1	1.2		2	3

Prueba de Desempeño

- Lo esencial es definir
  - Procedimientos de prueba
    - Ejemplo: Tiempo de respuesta
      - Simular la carga, tomar los tiempos de tal manera, número de casos a probar, etc
  - Criterios de aceptación
    - Ejemplo
      - El cliente lo valida si en el 90% de los casos de prueba en el ambiente de producción se cumple con los requerimientos de tiempo de respuesta
  - Características del ambiente
    - Definir las características del ambiente de producción ya que estas hacen grandes diferencias en los requerimientos no funcionales

Prueba de Desempeño

- Prueba de estrés (esfuerzo)
- Prueba de volumen
- Prueba de facilidad de uso
- Prueba de seguridad
- Prueba de rendimiento
- Prueba de configuración
- Prueba de compatibilidad
- Prueba de facilidad de instalación
- Prueba de recuperación
- Prueba de confiabilidad
- Prueba de documentación

Prueba de Desempeño

- Prueba de estrés (esfuerzo)
  - Esta prueba implica someter al sistema a grandes cargas o esfuerzos considerables.
  - No confundir con las pruebas de volumen. Un esfuerzo grande es un pico de volúmenes de datos (normalmente por encima de sus límites) en un *corto período de tiempo*
  - No solo volúmenes de datos sino también de usuarios, dispositivos, etc
  - Analogía con dactilógrafo
    - Volumen: Ver si puede escribir un documento enorme
    - Esfuerzo: Ver si puede escribir a razón de 50 palabras x minuto
  - Ejemplo
    - Un sistema de conmutación telefónica es sometido a esfuerzo generando un número grande de llamadas telefónicas

Prueba de Desempeño

- Prueba de volumen
  - Esta prueba implica someter al sistema a grandes volúmenes de datos
  - El propósito es mostrar que el sistema no puede manejar el volumen de datos especificado
  - Ejemplos
    - A un compilador lo ponemos a compilar un programa ***absurdamente*** grande
    - Un simulador de circuito electrónico puede recibir un circuito diseñado con miles de componentes

Prueba de Desempeño

- Prueba de facilidad de uso
  - Trata de encontrar problemas en la facilidad de uso
  - Algunas consideraciones a tener en cuenta
    - ¿Ha sido ajustada cada interfaz de usuario a la inteligencia y nivel educacional del usuario final y a las presiones del ambiente sobré el ejercidas?
    - ¿Son las salidas del programa significativas, no-abusivas y desprovistas de la "jerga de las computadoras"?
    - ¿Son directos los diagnósticos de error (mensajes de error) o requieren de un doctor en ciencias de la computación?
  - Myers presenta más consideraciones. Lo que importa es darse cuenta lo necesario de este tipo de pruebas. Si no se realizan el sistema puede ser inutilizable.
  - Algunas se pueden realizar durante el prototipado

Prueba de Desempeño

- Prueba seguridad
  - La idea es tratar de generar casos de prueba que burlen los controles de seguridad del sistema
  - Ejemplo
    - Diseñar casos de prueba para vencer el mecanismo de protección de la memoria de un sistema operativo
  - Una forma de encontrar casos de prueba es estudiar problemas conocidos de seguridad en sistemas similares. Luego mostrar la existencia de problemas parecidos en el sistema bajo test.
  - Existen listas de descripciones de fallas de seguridad en diversos tipos sistemas

Prueba de Desempeño

- Prueba de rendimiento
  - Generar casos de prueba para mostrar que el sistema no cumple con sus especificaciones de rendimiento
  - Casos
    - Tiempos de respuesta en sistemas interactivos
    - Tiempo máximo en ejecución de alguna tarea
  - Normalmente esto está especificado bajo ciertas condiciones de carga y de configuración del sistema
    - Ejemplo
      - El proceso de facturación no debe durar más de una hora en las siguientes condiciones:
        - » Se ejecuta en el ambiente descrito en el anexo 1
        - » Se ejecuta para 10.000 empleados

Prueba de Desempeño

- Prueba de configuración
  - Se prueba el sistema en distintas configuraciones de hardware y software
  - Como mínimo las pruebas deben ser realizadas con las configuraciones mínima y máxima posibles
- Prueba de compatibilidad
  - Son necesarias cuando el sistema bajo test tiene interfaces con otros sistemas
  - Se trata de determinar que las funciones con la interfaz no se realizan según especifican los requerimientos

Prueba de Desempeño

- Prueba de facilidad de instalación
  - Se prueban las distintas formas de instalar el sistema
- Prueba de recuperación
  - Se intenta probar que no se cumplen los requerimientos de recuperación
  - Para esto se simulan o crean fallas y luego se testea la recuperación del sistema

Prueba de Desempeño

- Prueba de confiabilidad
  - El propósito de toda prueba es aumentar la confiabilidad del sistema
  - Este tipo de prueba es aquella que se realiza cuando existen requerimientos específicos de confiabilidad del sistema →Se deben diseñar casos de prueba especiales
  - Esto no siempre es fácil
    - Ejemplo: El sistema Bell TSPS dice que tiene un tiempo de detención de 2 horas en 40 años.
      - No se conoce forma de probar este enunciado en meses o en unos pocos años
  - De otros casos se puede estimar la validez con modelos matemáticos
    - Ejemplo: El sistema tiene un valor medio de tiempo entre fallas de 20 horas

Prueba de Desempeño

- Prueba de documentación
  - La documentación del usuario debe ser objeto de inspección controlando su exactitud y claridad (entre otros)
  - Además todos los ejemplos que aparezcan en la misma deben ser codificados como casos de prueba. Es importantísimo que al ejecutar estos casos de prueba los resultados sean los esperados
    - Un sistema uruguayo tiene en su manual de usuario un ejemplo que no funciona en el sistema

Prueba de Desempeño

- Se mostraron algunos de los tipos de prueba pero no se dio ningún método de cómo estas se realizan
- Los métodos y los tipos de prueba dependen mucho del sistema que se esté diseñando
- Existen herramientas que automatizan algún tipo de estas pruebas. Por ejemplo las pruebas de esfuerzo

Prueba de Aceptación e Instalación

- Prueba de aceptación
  - El cliente realiza estas pruebas para ver si el sistema funciona de acuerdo a sus requerimientos y necesidades

- Prueba de instalación
  - Su mayor propósito es encontrar errores de instalación
  - Es de mayor importancia si la prueba de aceptación no se realizó en el ambiente de instalación

Otras Pruebas

- Desarrollo para múltiples clientes
  - Prueba Alfa
    - Realizada por el equipo de desarrollo sobre el producto final
  - Prueba Beta
    - Realizada por clientes elegidos sobre el producto final
    - Se podría decir que Windows está siempre en Beta testing
- Desarrollo de un sistema que sustituye a otro
  - Pruebas en paralelo
    - Se deja funcionando el sistema "viejo" mientras se empieza a usar al nuevo
    - Se hacen comparaciones de resultados y se intenta ver que el sistema nuevo funciona adecuadamente

Otras Pruebas

- Pruebas de regresión
  - Después de que se realizan modificaciones se verifica que lo que ya estaba probado sigue funcionando
  - ¿Conviene tener automatizadas las pruebas así las pruebas de regresión se pueden hacer con mucha mayor rapidez?
- Pruebas piloto
  - Se pone a funcionar el sistema en producción de forma localizada
  - Menor impacto de las fallas

Herramientas

- Temario
  - Distintos tipos de herramientas
  - Clasificación de testingfaqs.org

Distintos Tipos de Herramientas

- Cada vez son más utilizadas. Existen procesos en los cuales su uso es obligatorio (XP)

- Herramientas para verificación automatizada (centradas en el código)
  - Estáticas (no se ejecuta el programa)
    - Chequean sintaxis
    - Generan grafos de flujo de control y pueden detectar problemas de estructura
  - Dinámicas (se ejecuta el programa)
    - Rastreo de la ejecución
    - Examinar el contenido de la memoria o instancias de datos

Distintos Tipos de Herramientas

- Herramientas para ejecución de pruebas
  - Automatización
    - Elaboración del plan de prueba
    - Ejecución
  - Captura y reproducción
    - Digitaciones, clicks y movimientos del mouse
    - Se guardan los datos y los resultados esperados
    - Luego de detectar un defecto y corregirlo son usadas para repetir las pruebas
  - Generación de drivers y de stubs
    - Asisten en la generación automática de drivers y stubs

Distintos Tipos de Herramientas

- Otras
  - Evaluadoras de cobertura
    - Instrumentación automática (modificación) del código para registrar la cobertura (por ejemplo: Hansel)
  - Generadores de casos de prueba
    - A partir del código fuente
    - Generan casos de prueba para asegurar determinada cobertura
    - También a partir de especificaciones
  - Ambientes integrados
    - Ofrecen conjunto de facilidades
    - Integran los tipos de herramientas mencionados más algunos otros tipos no vistos en el curso

Clasificación

www.testingfaqs.org/tools.htm

- Test Design Tools
  - Herramientas que ayudan a decidir cuales tests se necesitan ejecutar (Ej: McCabe Test)
- GUI Test Drivers
  - Herramientas que automatizan la ejecución de tests para productos que usan interfaces gráficas
- Load and Performance Tools
  - Herramientas que se especializan en poner una carga alta en el sistema
- Unit Test Tools
  - Herramientas que soportan el testing unitario

Clasificación

www.testingfaqs.org/tools.htm

- Test Implementation Tools
  - Generan stubs
  - Generan assertions para hacer las faltas mas obvias
- Test Evaluation Tools
  - Ayudan a evaluar que tan buenos son los test set
  - Por ejemplo code coverage
- Static Analysis Tools
  - Herramientas que analizan programas sin correrlos
- Defect Tracking Tools
  - Herramientas que ayudan a manejar una base de datos de reportes de errores

Clasificación

www.testingfaqs.org/tools.htm

- Test Managers
  - Ayudan a gerenciar grandes cantidades de test set

JUnit

- Artículo a leer: "Infected: Programmers Love Writing Tests"
- JUnit es una herramienta de testing unitario para Java
- Para cada clase creamos otra que es la que va a testear el correcto funcionamiento de esta

JUnit

- El famoso ejemplo del triángulo

```
public class Triangulo {

   private int lado1 = 0;  private int lado2 = 0;  private int lado3 = 0;

   public Triangulo(int parLado1, int parLado2, int parLado3) throws Exception {
      if ( ((parLado1 + parLado2) <= parLado3) ||
           ((parLado2 + parLado3) <= parLado1)  ||
           ((parLado1 + parLado3) <= parLado2) )
           throw (new Exception());
      lado1 = parLado1;       lado2 = parLado2;       lado3 = parLado3;
   }
   //por motivos didacticos se usa el if y no se hace el return directamente
   public boolean isEscaleno() {
      if ( (lado1 != lado2) && (lado1 != lado3) && (lado2 != lado3) )  return true;
      return false;
   }

   public boolean isEquilatero() {
      if ( (lado1 == lado2) && (lado2 == lado3) )   return true;
      return false;
   }

   public boolean isIsosceles() { //si es equilatero también es isosceles
      if ( (lado1 == lado2) || (lado2 == lado3) || (lado1 == lado3) )  return true;
      return false;
   }
}
```

JUnit

```
public class TrianguloTest extends junit.framework.TestCase {
   private Triangulo tEscaleno;
   private Triangulo tNoEscaleno;

   protected void setUp() {
      try {
         tEscaleno = new Triangulo(2,3,4);
         tNoEscaleno = new Triangulo(2,2,3);
      }
      catch (Exception e) {
      }
   }

   public static Test suite() {
      return new TestSuite(TrianguloTest.class);
   }

public void testNoTriangulo() {
      try {  Triangulo tri = new Triangulo(1,1,0);  }
      catch (Exception e){    Assert.assertTrue(true);   return; }
      Assert.assertTrue(false);
   }

   public void testTrianguloEscaleno() {
      boolean resultado = tEscaleno.isEscaleno();   Assert.assertTrue(resultado);
      resultado = tNoEscaleno.isEscaleno();    Assert.assertTrue(!resultado);

   }
}
```

Heredo de la clase TestCase de JUnit

Al ejecutar el test se llama automáticamente al método setUp

Hace que se ejecuten todos los test de esta clase

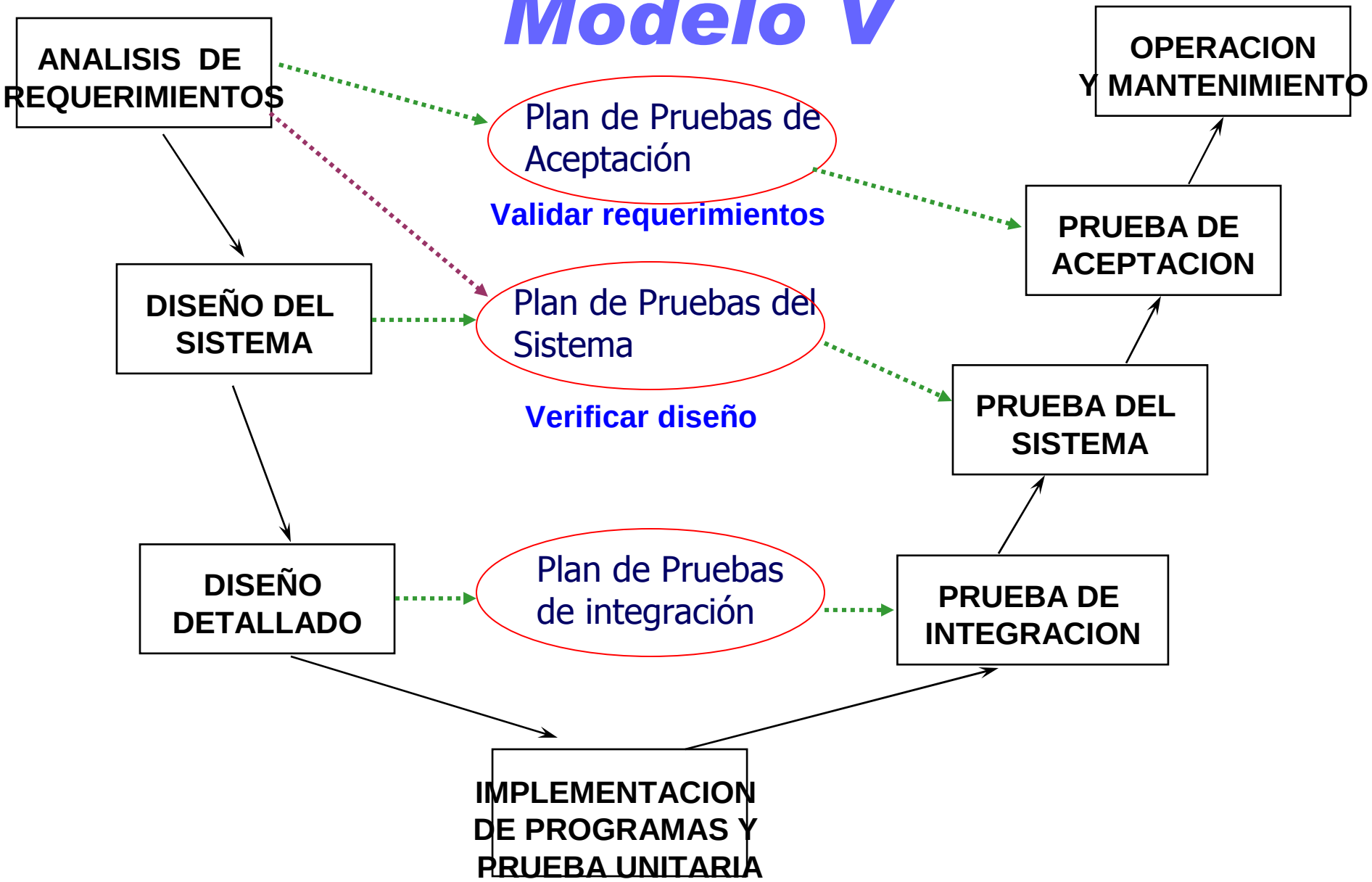
Testea la creación de un triángulo que no es válido

Testea por triángulo escaleno con un caso escaleno y el otro no

Planificación de V&V

- Temario
  - Modelo V
  - Generalidades
  - Planes de prueba

Modelo V


ANALISIS DE REQUERIMIENTOS
DISEÑO DEL SISTEMA
DISEÑO DETALLADO
IMPLEMENTACION DE PROGRAMAS Y PRUEBA UNITARIA
PRUEBA DE INTEGRACION
PRUEBA DEL SISTEMA
PRUEBA DE ACEPTACION
OPERACION Y MANTENIMIENTO
Plan de Pruebas de Aceptación
Validar requerimientos
Plan de Pruebas del Sistema
Verificar diseño
Plan de Pruebas de integración

Generalidades

El mayor error cometido en la planificación de un proceso de prueba

Suponer al momento de realizar el cronograma, que no se encontrarán fallas

Los resultados de esta equivocación son obvios

- Se subestima la cantidad de personal a asignar
- Se subestima el tiempo de calendario a asignar
- Entregamos el sistema fuera de fecha o ni siquiera entregamos

Generalidades

- La V&V es un proceso caro → se requiere llevar una planificación cuidadosa para obtener el máximo provecho de las revisiones y las pruebas para controlar los costos del proceso de V&V
- El proceso de planificación de V&V
  - Pone en la balanza los enfoques estático y dinámico para la verificación y la validación
  - Utiliza estándares y procedimientos para las revisiones y las pruebas de software
  - Establece listas de verificación para las inspecciones
  - Define el plan de pruebas de software
- A sistemas más críticos mayor verificación estática

Planes de Prueba (Sommerville)

- Comprende aplicar los estándares para el proceso de prueba más que a describir las pruebas del producto
- Permite que el personal técnico obtenga una visión global de las pruebas del sistema y ubique su propio trabajo en ese contexto
- También proveen información al personal responsable de asegurar que los recursos de hardware y software apropiados estén disponibles para el equipo de pruebas

Planes de Prueba (Sommerville)

- Componentes principales del plan
  - Proceso de prueba
    - Descripción principal de las fases de prueba
  - Requerimientos de rastreo o seguimiento
    - Se planifican pruebas de tal forma que se puedan testear individualmente todos los requerimientos
  - Elementos probados
    - Se especifican los productos del proceso de software a probar
  - Calendarización del las pruebas
    - Calendarización de todas las pruebas y asignación de recursos. Esto se vincula con el calendario general del proyecto

Planes de Prueba (Sommerville)

- Componentes principales del plan (2)
  - Procedimiento de registro de las pruebas
    - Los resultados de la ejecución de las pruebas se deben grabar sistemáticamente. Debe ser posible auditar los procesos de prueba para verificar que se han llevado a cabo correctamente
  - Requerimientos de hardware y software
    - Esta sección señala el hardware y software requerido y la utilización estimada del hardware
  - Restricciones
    - Se anticipan las restricciones que afectan al proceso de prueba, como por ejemplo la escasez de personal
- Como otros planes, el plan de pruebas no es estático y debe revisarse regularmente

Planes de Prueba (Pfleeger)

- Se deben planear cada uno de los siguientes pasos de prueba
  - Determinación de los objetivos de la prueba
  - Diseño de los casos de prueba
  - Preparación escrita de los casos de prueba
  - Verificación de los casos de prueba
  - Ejecución de las pruebas
  - Evaluación de los resultados

Planes de Prueba (Pfleeger)

- Componentes principales del plan
  - Objetivos del plan de prueba
    - Organiza cada tipo de prueba (unitaria, de integración, etc). Por lo tanto el plan de prueba es una serie de planes de prueba; uno por cada tipo de prueba
  - Cómo se ejecutarán las pruebas
    - Tipos de prueba, quienes las ejecutan, etc
  - Criterios de terminación
    - Elegir criterios realistas de terminación de las pruebas. Ej: criterios de cobertura
  - Métodos a utilizar
    - Descripción de los métodos a usar para cada tipo de prueba (unitaria, etc). Por ejemplo, recorridas en unitarias. Describe el uso de herramientas

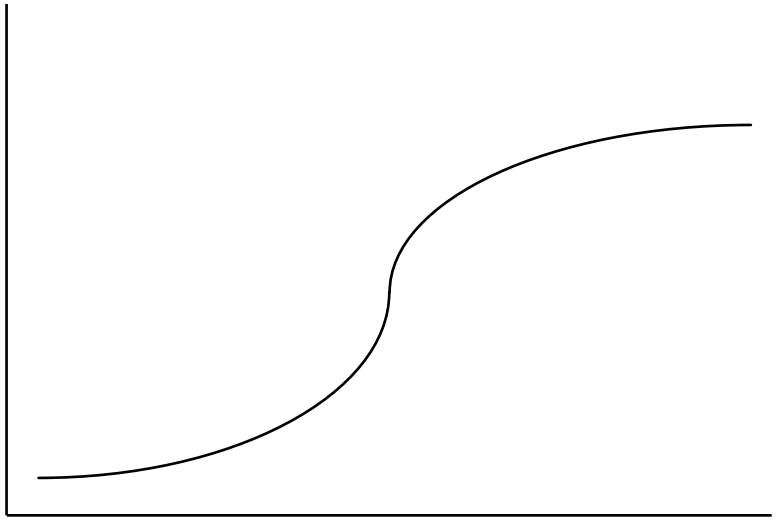
Planes de Prueba (Pfleeger)

- Componentes principales del plan (2)
  - Generación de casos de prueba y recopilación
    - Explica como se generan los casos de prueba y como se recopilarán las salidas.
    - Descripción de herramientas automáticas de casos de prueba, de seguimiento de defectos, etc

Terminación de la Prueba

- Temario
  - Más faltas... más por encontrar
  - Criterios de terminación
  - Estimar cantidad de defectos

Más Faltas... más por Encontrar



Estudio de Myers en 1979

- Va en contra de nuestra intuición
- Hace difícil saber cuando detener la prueba
- Sería bueno estimar el número de defectos remanente
  - Ayuda a saber cuando detener las pruebas
  - Ayuda a determinar el grado de confianza en el producto

Criterios de Terminación

- ¿Es la última falta detectada la última que quedaba?
- No es razonable esperar descubrir todas las faltas de un sistema
  - En vista de este dilema y que la economía muchas veces es quien dictamina que la prueba debe darse por terminada, surgen las alternativas de terminar la prueba de forma puramente arbitraria o de usar algún criterio útil para definir la terminación de la prueba

Criterios de Terminación

- Criterios usados pero contraproducentes
  - Terminar la prueba cuando
    1. el tiempo establecido para la misma ha expirado
    2. Todos los casos de prueba se han ejecutado sin detectar fallas (es decir, cuando todos los casos de prueba son infructuosos)
  - El primer criterio es inútil puesto que puede ser satisfecho sin haber hecho absolutamente nada. Este criterio no mide la calidad de las pruebas realizadas
  - El segundo criterio es inútil porque es independiente de la calidad de los casos de prueba. Es también contraproducente porque subconscientemente se tienden a crear casos de prueba con baja probabilidad de descubrir errores

Categorías de Criterios (Myers)

1. Por criterios del diseño de casos de prueba
   - Basa la terminación de las pruebas en el uso de métodos específicos de casos de prueba (ejemplo: cubrimiento de arcos)
2. Detención basada en la cantidad de defectos
   - Ejemplos
     - Detectar 3 defectos por módulo y 70 en la prueba funcional
     - Tener detectado el 90% de los defectos del sistema
3. Basada en la cantidad de fallas detectadas por unidad de tiempo, durante las pruebas

Por Criterios del Diseño de Casos de Prueba

- Se definen criterios de terminación según el cumplimiento de un criterio de cubrimiento y que todos esos casos de prueba se ejecuten sin provocar fallas
- Ejemplo:
  - Termino las pruebas cuando ejecute todas las trayectorias independientes del programa y todos los test den igual al resultado esperado
  - Esto se puede hacer con ayuda de herramientas que indican que cobertura tengo del código con los casos de prueba ejecutados

Por Criterios del Diseño de Casos de Prueba

- Pros y contras
  - ✓ Pueden ser apropiados para las pruebas unitarias
  - ✗ No son muy útiles en la prueba del sistema
  - ✗ Se fijan los métodos de diseño de casos de prueba pero no se da una meta respecto a las faltas del sistema

Detención Basada en la Cantidad de Defectos

- Termino la prueba basado en la cantidad de defectos → ¿cómo determino el número de defectos a ser detectado para dar por terminada la prueba?
  - Necesito estimar el número total de defectos. De esta manera puedo saber cuando parar la prueba
  - Ejemplo de criterio
    - Detener la prueba cuando haya detectado el 90% de los defectos
    - Para saber cuando detecte el 90% de los defectos uso la estimación del total de defectos
- A continuación se presentan formas de estimar la cantidad total de defectos

Estimar Cantidad de Defectos

- Siembra de defectos (Fault Seeding)
- Pruebas independientes

Siembra de Defectos

- Sirve para estimar el número de defectos
  - Se agregan defectos intencionalmente en un módulo
  - Se hacen pruebas para detectar defectos en ese módulo
  - Se usa la siguiente relación (creer o reventar)

$$\frac{\text{Defectos sembrados detectados}}{\text{Total de defectos sembrados}} = \frac{\text{Defectos no sembrados detectados}}{\text{Total de defectos no sembrados}}$$

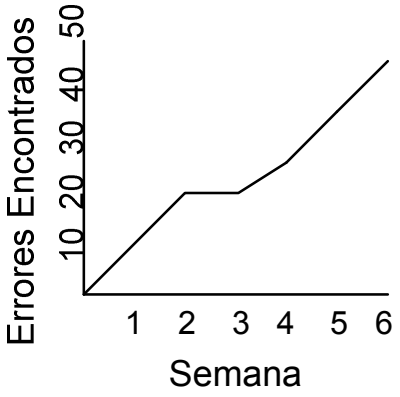
  - Despejando se obtiene una estimación del total de defectos no sembrados
  - ✓ Es un enfoque simple y útil
  - ✗ Asume que los defectos sembrados son representativos de los defectos reales (tipo y complejidad)

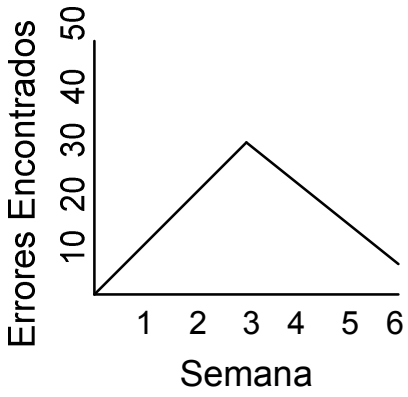
Pruebas Independientes

- Pongo dos grupos de prueba a probar el mismo programa
- Defectos encontrados por Grupo 1 = x
- Defectos encontrados por Grupo 2 = y
- Defectos en común = q entonces q<=x & q<=y
- Defectos totales = n (número a estimar)
- Efectividad = cantDefectosEncontrados / n
- Efectividad Grupo1 = E1 = x/n E2 = y/n
- Consideremos el Grupo 1 y tomemos que la tasa de efectividad es igual para cualquier parte del programa
  - El Grupo 1 encontró q de los y defectos detectados por el grupo 2 → E1 = q / y
- Entonces se deriva la siguiente formula n = (x*y)/q

Detención Basada en Fallas por Unidad de Tiempo

- Se registra el número de fallas que se detectan durante la prueba por unidad de tiempo. Luego se grafican estos valores
- Examinando la forma de la curva se puede determinar cuando detener las pruebas





- Si tengo la gráfica de la izquierda y estoy usando este método de detención debería continuar las pruebas
- La gráfica de la derecha podría estar indicando que es un buen momento para parar las pruebas. De todos modos hay que garantizar que la bajada en la detección de fallas no se deba a factores como:
  - Menos cantidad de pruebas ejecutadas
  - Agotamiento de los casos de prueba preparados

Una Combinación

- Para la fase de prueba de sistema la mezcla de las dos últimas categorías puede resultar positiva

Se pararán las pruebas cuando se detecte un número predeterminado de defectos o cuando haya transcurrido el tiempo fijado para ello, eligiendo la que ocurra más tarde, siempre que un análisis del gráfico del número de defectos detectados por unidad de tiempo en función del tiempo indique que la prueba se ha tornado improductiva