

Aseguramiento de la Calidad del Software

Universidad Autónoma de Coahuila
Facultad de Sistemas
Calidad y Pruebas del Software

Carlos Nassif Trejo García

“Cliente interno”

Definiciones

- Calidad de diseño: Características que los diseñadores especifican para el producto final
- Calidad de conformidad: Grado en la cual se siguen las especificaciones de diseño
- Calidad de control: Inspecciones, revisiones y pruebas
- Aseguramiento de Calidad (QA): Auditoria y reportes de procedimientos

Costos

Prevención

- Plan de calidad
- Revisiones técnicas formales
- Equipo de prueba
- Entrenamiento

Apreciación:

- Inspecciones
- Calibración y mantenimiento de equipo
- Pruebas

Falla

- Re trabajo, reparación, Análisis de fallas

Fallas externas

- Resolución de queja, devolución, Help line, garantías

Incluye

Proceso de ACS

Tareas específicas

Métodos y herramientas

Control de desarrollo y cambios

Cumplimiento de estándares

Medición y reportes

Elementos

Estándares

Revisiones y auditorias

Pruebas

Colección y análisis

Administración del cambio

Educación

Administración de los proveedores

Administración de la seguridad

Seguridad

Administración de riesgos

Estandares



IEEE

*Advancing Technology
for Humanity*





III Revisiones y auditorias



Pruebas



Colección y análisis de los errores



Administración del cambio



Educación



Administración de los proveedores



Paquetes contenidos en un caja

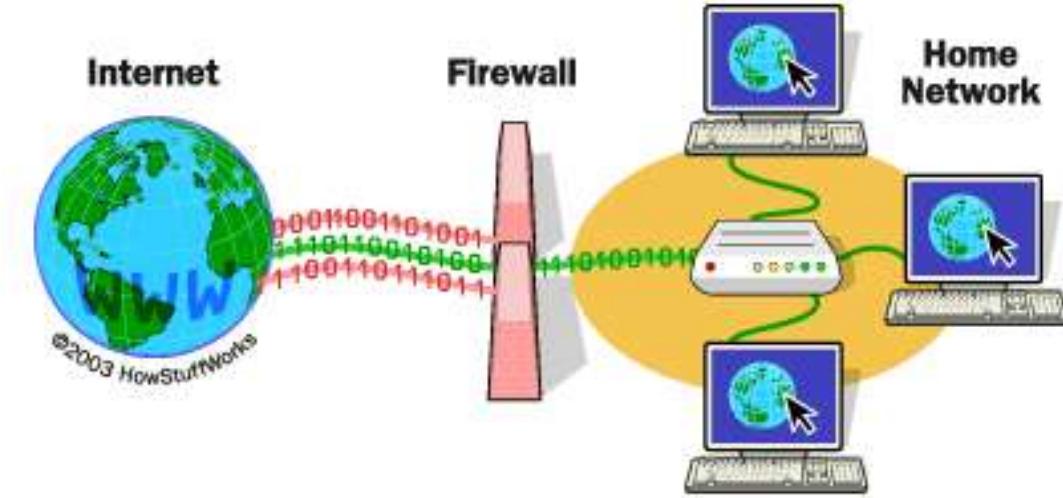


Shell personalizado



Software contratado

Administración de la seguridad



Seguridad



EFECTO DE FALLAS



DISMINUIR RIESGOS

Administración de riesgos



Asegura que se lleve acabo la administración de riesgos



Planes de contingencia

Técnicas

Ingenieros de software

- Trabajo técnico

Grupo de ACS

- Planear
- Supervisar
- Registrar
- Analizar
- Reportar

Tareas del ACS

Preparar el plan de ACS para un proyecto

Participa en el desarrollo de la descripción del software del proyecto

Revisar actividades de los ingenieros de software

Audita productos de trabajo

Maneja desviaciones de trabajo

Registra y reporta fallas de cumplimiento

Meta	Atributo	Métrica
Calidad de los requerimientos	Ambigüedad	Número de modificadores ambiguos (por ejemplo, muchos, grande, amigable, etc.)
	Compleitud	Número de TBA y TBD
	Comprendibilidad	Número de secciones y subsecciones
	Volatilidad	Número de cambios por requerimiento
		Tiempo (por actividad) cuando se solicita un cambio
	Trazabilidad	Número de requerimientos no trazables hasta el diseño o código
	Claridad del modelo	Número de modelos UML
Calidad del diseño		Número de páginas descriptivas por modelo
	Integridad arquitectónica	Existencia del modelo arquitectónico
	Compleitud de componentes	Número de componentes que se siguen hasta el modelo arquitectónico
		Complejidad del diseño del procedimiento
Calidad del código	Complejidad de la interfaz	Número promedio de pasos para llegar a una función o contenido normal
		Distribución apropiada
	Patrones	Número de patrones utilizados
Eficacia del control de calidad	Complejidad	Complejidad ciclomática
	Facilidad de mantenimiento	Factores de diseño (capítulo 8)
	Comprendibilidad	Porcentaje de comentarios internos
		Convenciones variables de nomenclatura
	Reusabilidad	Porcentaje de componentes reutilizados
	Documentación	Índice de legibilidad
	Asignación de recursos	Porcentaje de personal por hora y por actividad
	Tasa de finalización	Tiempo de terminación real versus lo planeado
	Eficacia de la revisión	Ver medición de la revisión (capítulo 14)
	Eficacia de las pruebas	Número de errores de importancia crítica encontrados
		Esfuerzo requerido para corregir un error
		Origen del error

Aseguramiento estadístico

1. Se recaba y clasifica la información acerca de errores y defectos del software.
2. Se hace un intento por rastrear cada error y defecto hasta sus primeras causas (por ejemplo, no conformidad con las especificaciones, error de diseño, violación de los estándares, mala comunicación con el cliente, etc.).
3. Con el uso del Principio de Pareto (80 por ciento de los defectos se debe a 20 por ciento de todas las causas posibles), se identifica 20 por ciento de las causas de errores y defectos (*las pocas vitales*).
4. Una vez identificadas las pocas causas vitales, se corrigen los problemas que han dado origen a los errores y defectos.

Seis Sigma

- Definir los requerimientos y metas
- Determinar el desempeño actual de calidad (Medir)
- Analizar métricas de los defectos y sus causas

Mejorar

- Eliminar causas originales
- Controlar el proceso

Desarrollar:

- Diseñar el proceso
- Verificar

Confiabilidad

- “Probabilidad que tiene un programa de computo de operar sin fallas en un ambiente específico por un tiempo específico”
- Tiempo medio entre fallas = tiempo medio para la falla + tiempo medio para la reparación
- Fallas en el tiempo:
 - 1 FET = 1 falla cada mil millones de horas de operación

Disponibilidad

- Probabilidad de que un programa opere de acuerdo con los requerimientos en un momento determinado de tiempo
- Disponibilidad =
$$(\text{Tiempo medio para la falla} / (\text{tiempo medio para la falla} + \text{tiempo medio para la recuperación})) * 100\%$$

Plan de ACS

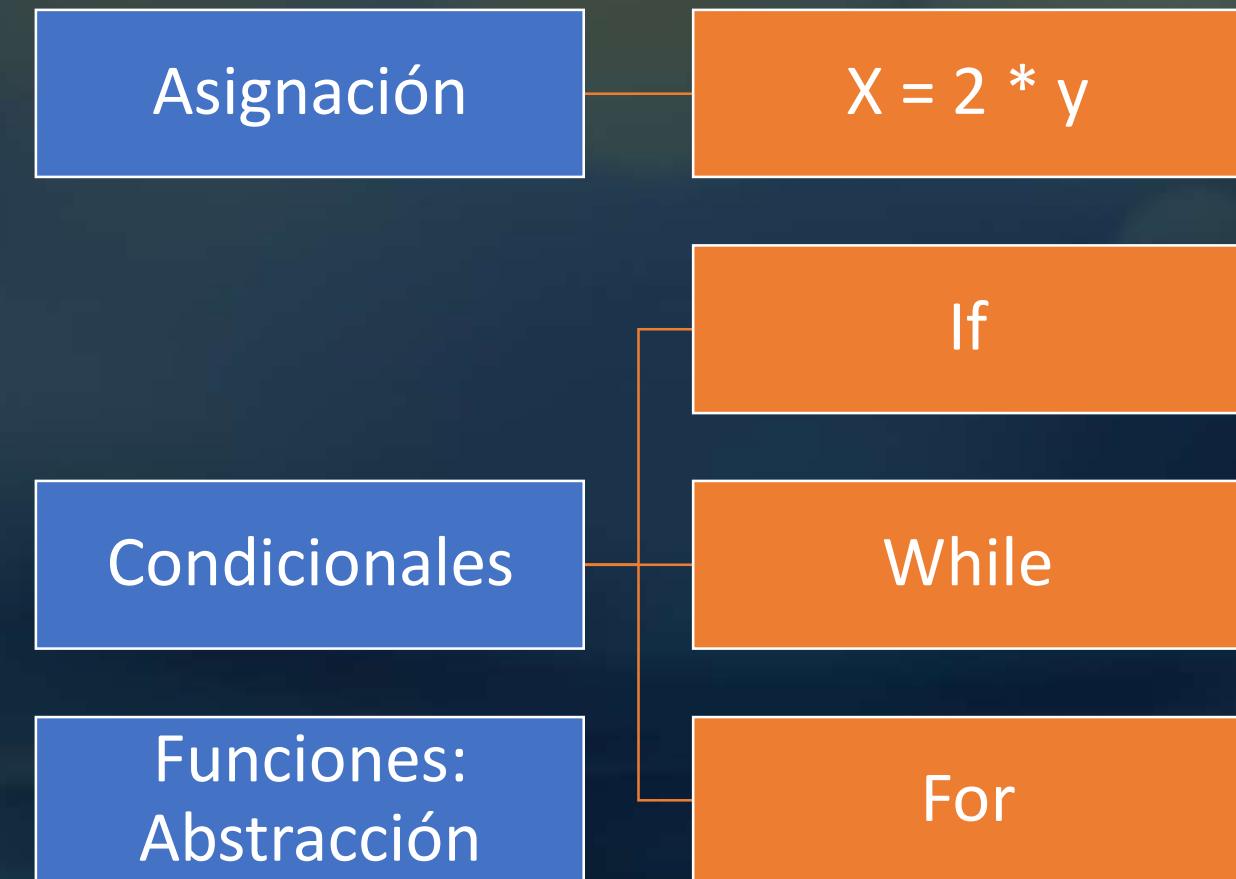
IEEE, estructura:

- Propósito y alcance del plan
- Descripción de todos los productos
- Normas y prácticas aplicables
- Acciones y tareas del ACS
- Herramientas y apoyos al ACS
- Procedimientos para la configuración
- Métodos para mantener los registros
- Roles y responsabilidades

Control Flow Testing

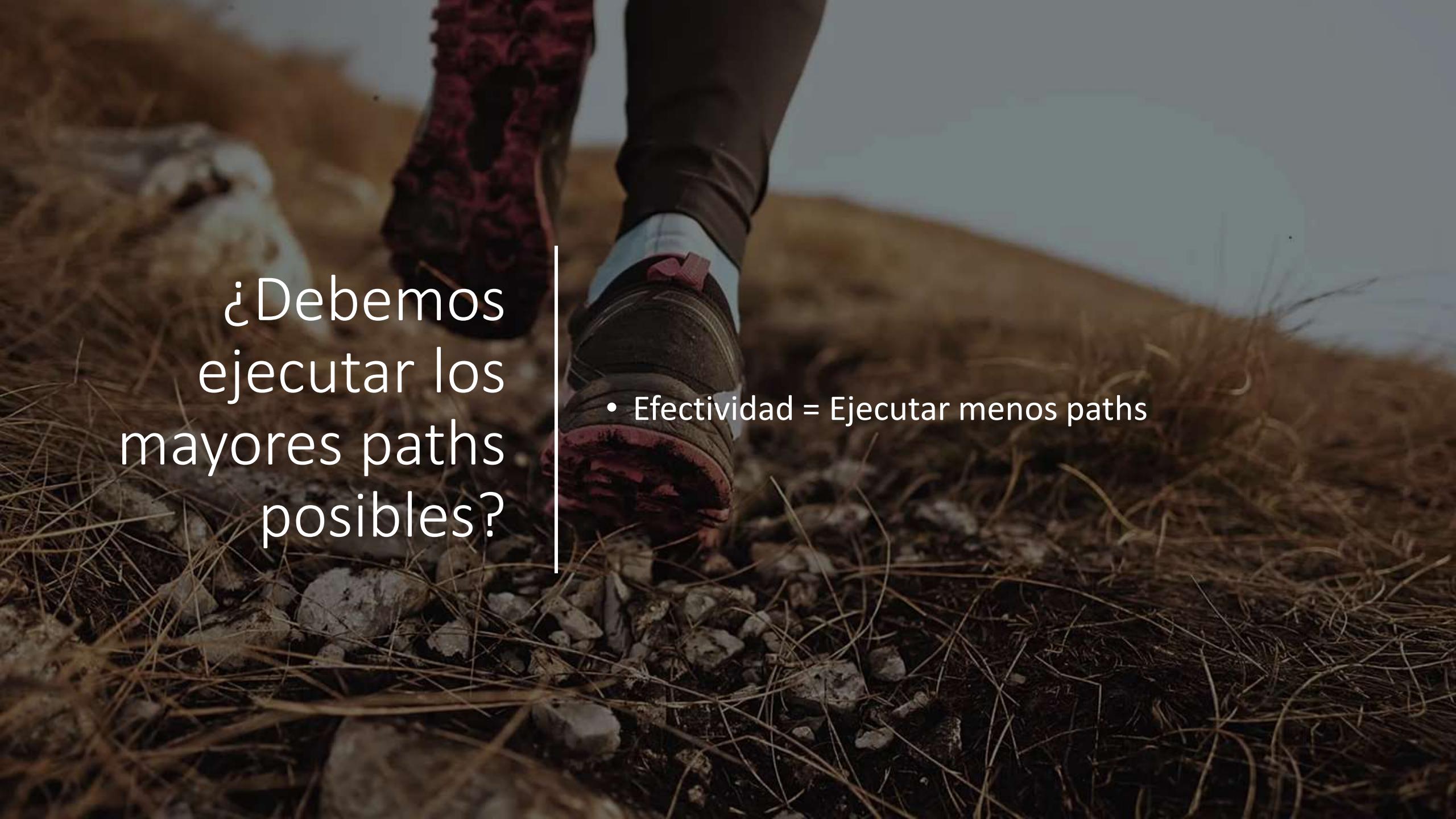
Universidad Autónoma de Coahuila
Facultad de Sistemas
Carlos Nassif Trejo García

Contexto



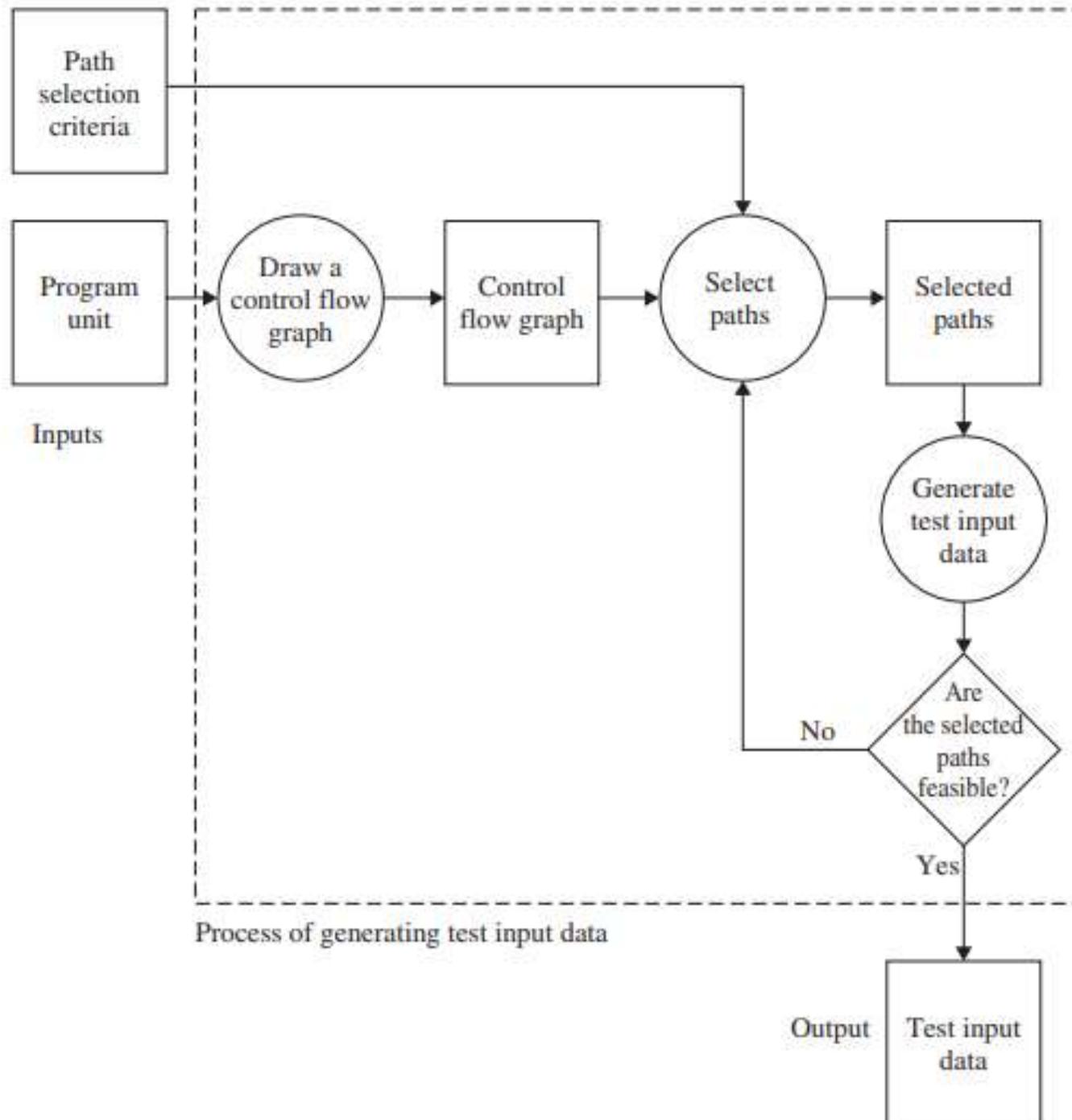
Program path

- Punto de entrada
- Punto de salida
- PP:
 - Ejecución de una secuencia de instrucciones desde la entrada hasta la salida
 - Entrada y salida esperada
 - Unas entradas en específico pueden ejecutar un path en específico



¿Debemos
ejecutar los
mayores paths
posibles?

- Efectividad = Ejecutar menos paths



Criterios

Cada declaración es ejecutado al menos un vez

Para cada condicional, obtener un verdadero y uno falso

Símbolos

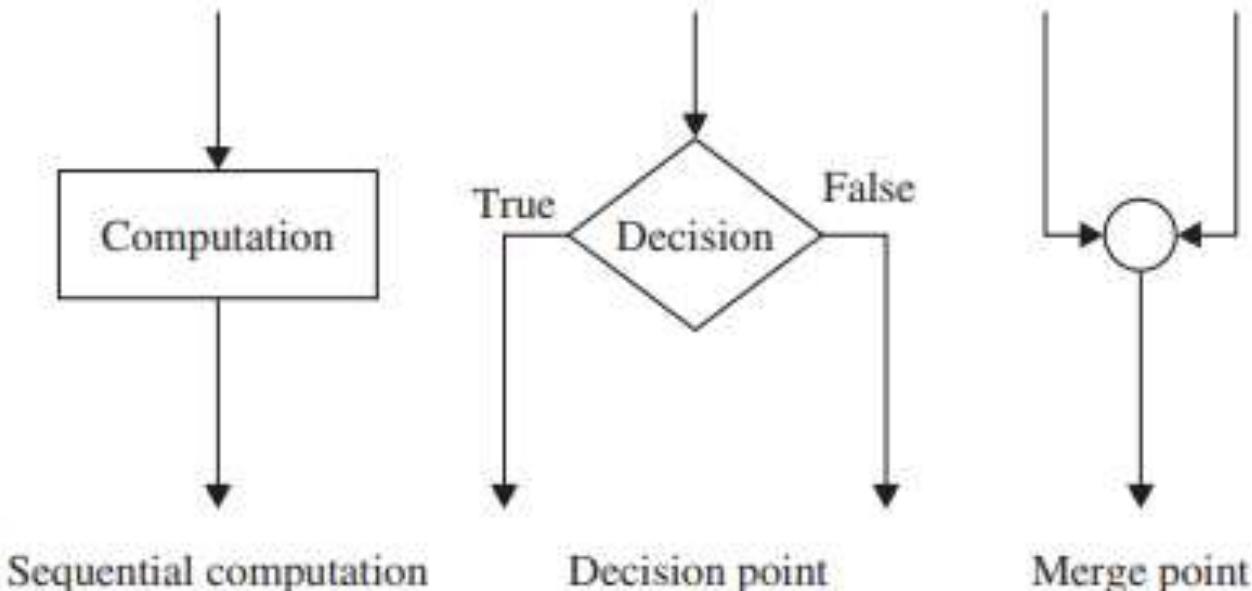


Figure 4.2 Symbols in a CFG.

```

FILE *fptr1, *fptr2, *fptr3; /* These are global variables. */

int openfiles(){
/*
    This function tries to open files "file1", "file2"
    "file3" for read access, and returns the number o
    successfully opened. The file pointers of the ope
    are put in the global variables.
*/
    int i = 0;
    if(
        ((( fptr1 = fopen("file1", "r")) != NULL) && (
            && (
                ((( fptr2 = fopen("file2", "r")) != NULL) && (
                    && (
                        ((( fptr3 = fopen("file3", "r")) != NULL) && (
                            );
                    return(i);
                }

```

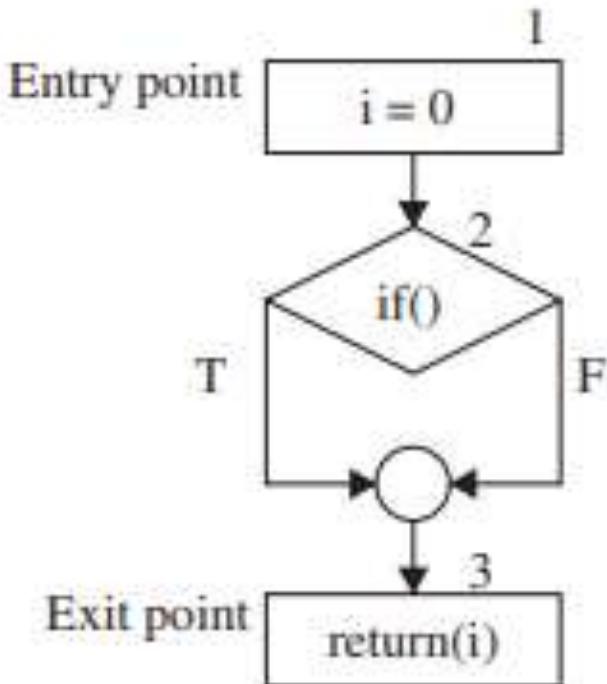
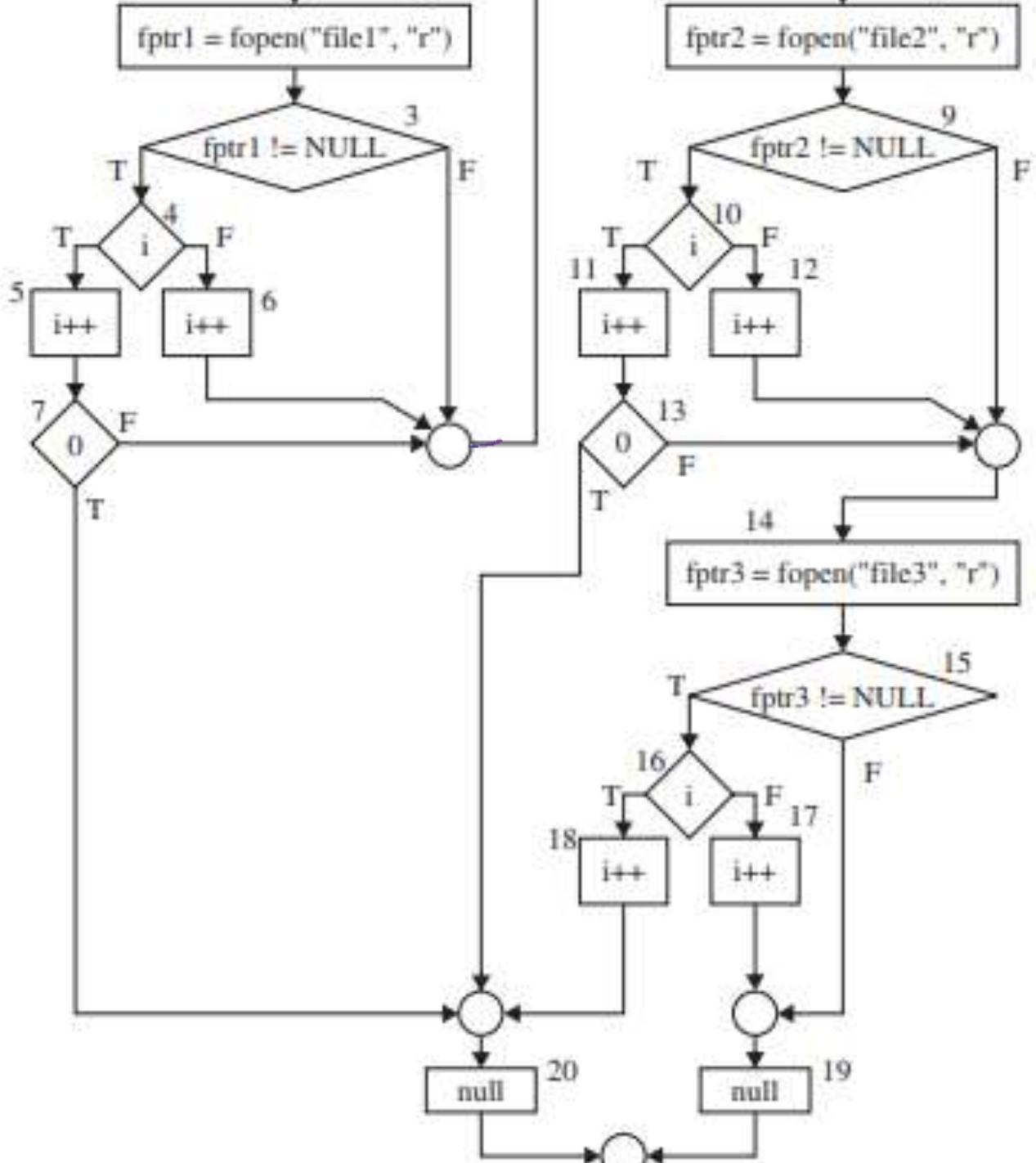


Figure 4.3 Function to open three files.



Selección de paths

- Cada declaración es ejecutado al menos una vez
- No usar entradas que activen el mismo path repetidamente
- Conocer las características de lo que ya fue probado y lo que no

Todos los paths

- Statement Coverage
- Branch Coverage
- Predicate Coverage

Todos los paths

TABLE 4.2 Input Domain of openfiles()

Existence of file1	Existence of file2	Existence of file3
No	No	No
No	No	Yes
No	Yes	No
No	Yes	Yes
Yes	No	No
Yes	No	Yes
Yes	Yes	No
Yes	Yes	Yes

TABLE 4.3 Inputs and Paths in openfiles()

Input	Path
<No, No, No >	1-2-3(F)-8-9(F)-14-15(F)-19-21
< Yes, No, No >	1-2-3(T)-4(F)-6-8-9(F)-14-15(F)-19-21
< Yes, Yes, Yes >	1-2-3(T)-4(F)-6-8-9(T)-10(T)-11-13(F)-14-15(T)-16(T)-18-20-21

Statement Coverage

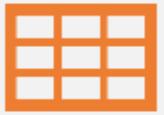
Cada declaración del código es ejecutado

Seleccionar paths pequeños

Seleccionar paths un poquito mas grandes

Seleccionar arbitrariamente paths complejos

Branch Coverage



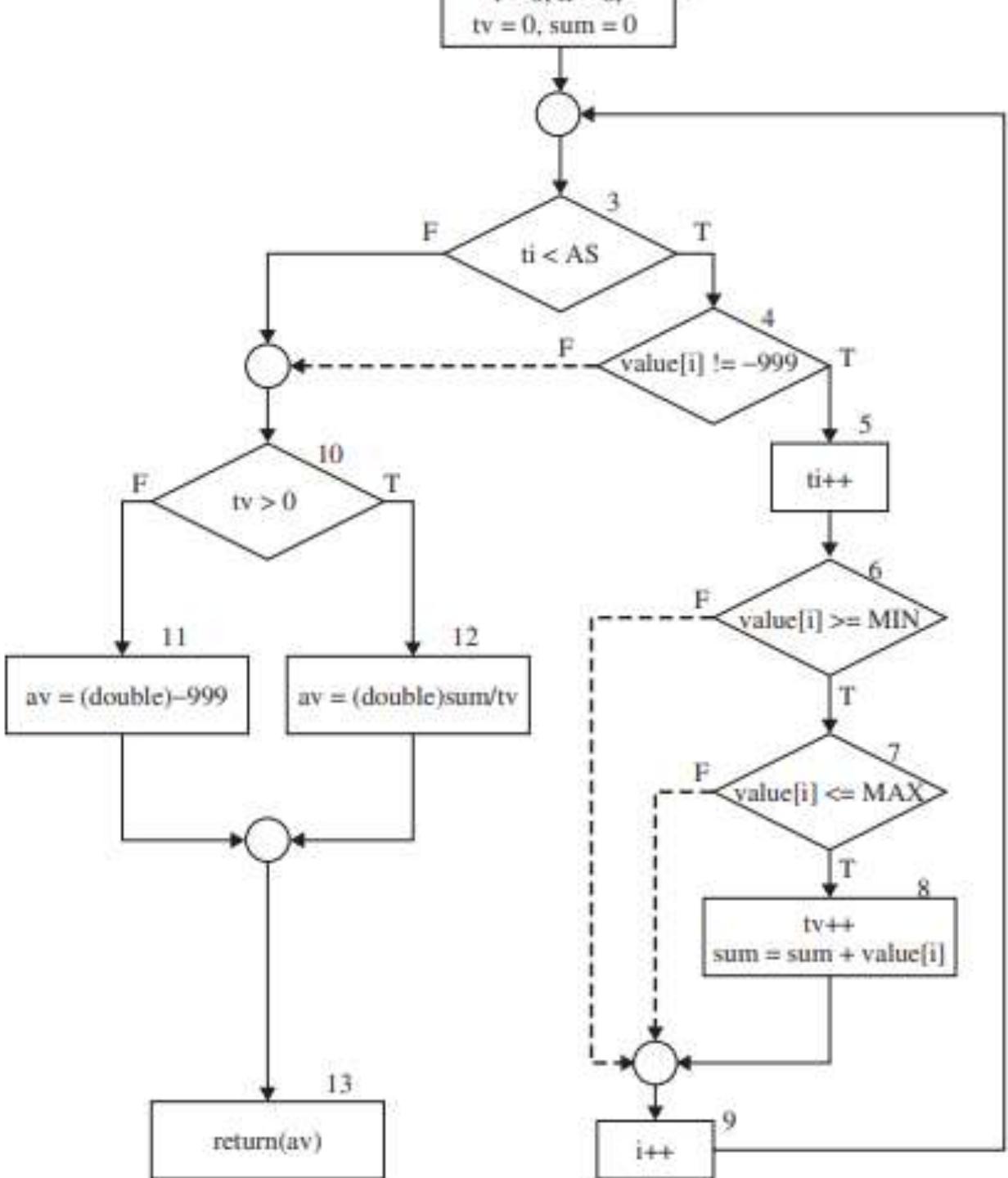
Rectángulos: 1



Rombo: 2

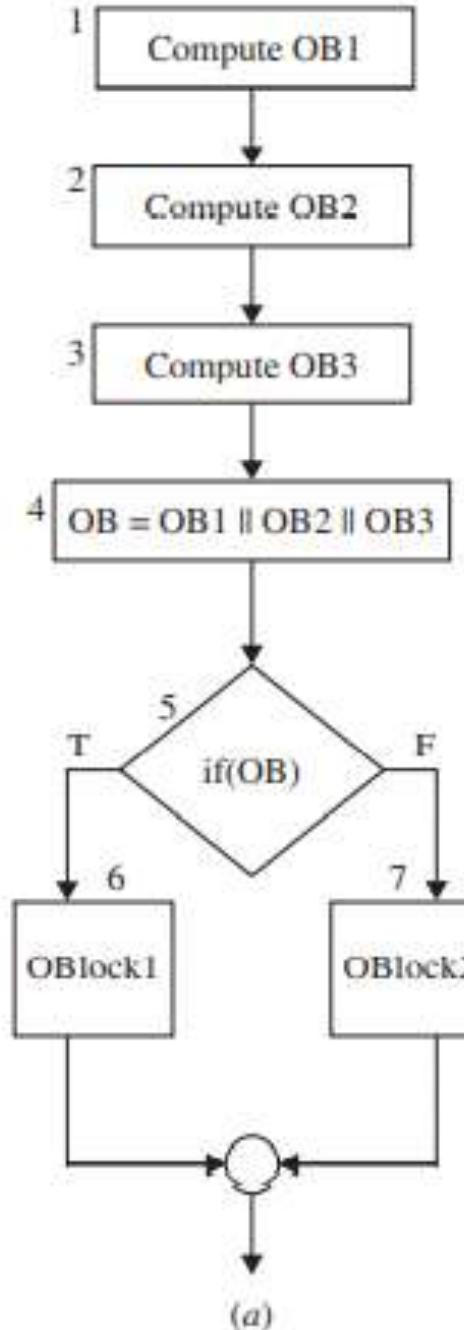


Salida: 0



Predicate Coverage

- También conocido como condition coverage
- Cada expresión booleana es evaluado como verdadero y como falso



Generando datos de entrada

Vector de entrada:

- Parámetros
- Variables y constantes globales
- Archivos
- Conexión internet
- Timers

Predicado: Función lógica evaluada en
un punto de decisión

- For (...)
- If (...)
- Booleano

Path Predicate

- Conjunto de predicados asociados a un path

```
1-2-3(T)-4(T)-5-6(T)-7(T)-8-9-3(F)-10(T)-12-13
```

Figure 4.10 Example of a path from Figure 4.7.

```
ti < AS ≡ True  
value[i] != -999 ≡ True  
value[i] >= MIN ≡ True  
value[i] <= MAX ≡ True  
ti < AS ≡ False  
tv > 0 ≡ True
```

Figure 4.11 Path predicate for path in Figure 4.10.

Interpretación de predicados

- Proceso de sustituir operaciones simbólicamente en un path determinado, para así poder expresar el predicado solamente en términos de vector de entrada y vector de constantes

```
public static int SymSub(int x1, int x2){  
    int y;  
    y = x2 + 7;  
    if (x1 + y >= 0)  
        return (x2 + y);  
    else return (x2 - y);  
}
```

Path Predicate Expression

No esta compuesto de variables locales, solo de vectores de entrada y de constantes

Es un conjunto de restricciones formado de los vectores mencionados

Valores de entrada de fuerza de ruta pueden ser generador al resolver el conjunto de restricciones

Si no se puede resolver el conjunto de restricciones, no existe valores de entrada que cause que el path se ejecute. i. e. path invalido

TABLE 4.7 Interpretation of Path Predicate of Path in Figure 4.10.

Node	Node Description	Interpreted Description
1	Input vector: $\langle \text{value}[], \text{AS}, \text{MIN}, \text{MAX} \rangle$	
2	$i = 0, t_i = 0,$ $tv = 0, \text{sum} = 0$	
3(T)	$t_i < \text{AS}$	$0 < \text{AS}$
4(T)	$\text{value}[i]! = -999$	$\text{value}[0]! = -999$
5	t_i++	$t_i = 0 + 1 = 1$
6(T)	$\text{value}[i] \geq \text{MIN}$	$\text{value}[0] \geq \text{MIN}$
7(T)	$\text{value}[i] \leq \text{MAX}$	$\text{value}[0] \leq \text{MAX}$
8	$tv++$ $\text{sum} = \text{sum} + \text{value}[i]$	$tv = 0 + 1 = 1$ $\text{sum} = 0 + \text{value}[0]$ $= \text{value}[0]$
9	$i++$	$i = 0 + 1 = 1$
3(F)	$t_i < \text{AS}$	$1 < \text{AS}$
10(T)	$tv > 0$	$1 > 0$
12	$av = (\text{double}) \text{sum}/tv$	$av = (\text{double}) \text{value}[0]/1$
13	$\text{return}(av)$	$\text{return}(\text{value}[0])$

Note: The bold entries in column 1 denote interpreted predicates.

0 < AS	\equiv	True	(1)
value[0] != -999	\equiv	True	(2)
value[0] >= MIN	\equiv	True	(3)
value[0] <= MAX	\equiv	True	(4)
1 < AS	\equiv	False	(5)
1 > 0	\equiv	True	(6)

Ejemplo de path invalido

1-2-3(T)-4(F)-10(T)-12-13.

Generar datos de entrada desde la Path Predicate Expression

Resolver el path predicate expression

0 < AS	\equiv True	(1)
value[0] != -999	\equiv True	(2)
value[0] >= MIN	\equiv True	(3)
value[0] <= MAX	\equiv True	(4)
1 < AS	\equiv False	(5)
1 > 0	\equiv True	(6)

Figure 4.13 Path predicate expression for path in Figure 4.10.

```
AS    = 1  
MIN   = 25  
MAX   = 35  
value[0] = 30
```



Possible
solución

Posibles errores

Generar datos de prueba para satisfacer el criterio de selección

Generar pruebas adicionales mas grandes

Inyectar errores

A close-up photograph of a clear plastic pipette dispensing a small amount of clear liquid onto a white surface. The surface features a repeating pattern of horizontal bands in various colors, including red, blue, green, and yellow, which is characteristic of a DNA gel electrophoresis analysis. The background is blurred.

Data Flow Testing

Universidad Autónoma de
Coahuila
Facultad de Sistemas

Carlos Nassif Trejo García

Idea general

Revisar el uso
de
apuntadores

Instanciar dos
veces

Motivaciones



Accesibilidad deseable de una variable (lectura / escritura)



Verificar la exactitud de un valor generado por una variables

Estático

Data Flow anomaly: Defectos potenciales de un programa

Dinámico

- Identificar paths de un programa desde el código fuente basado en una clase de criterios de data Flow testing

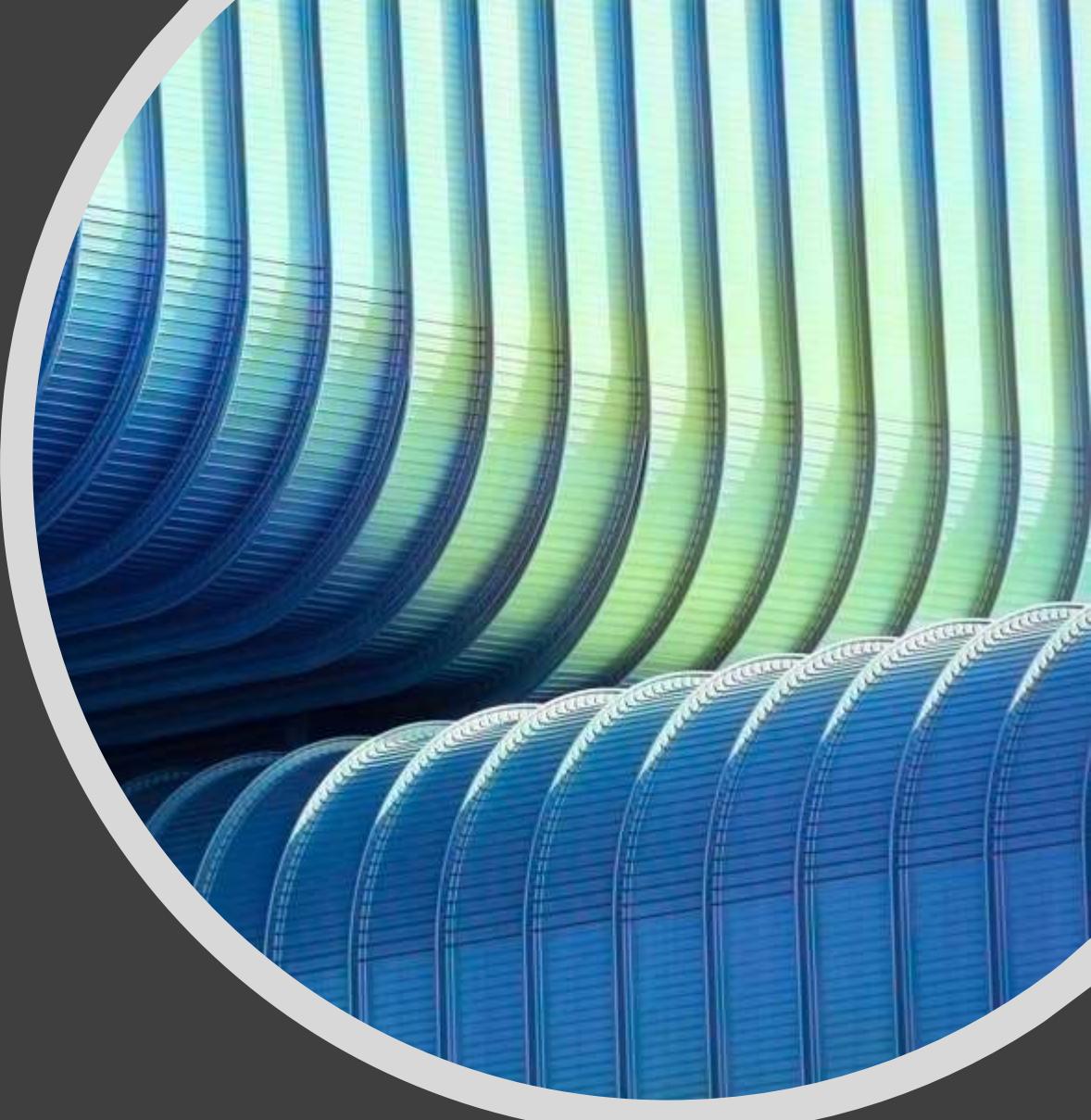
Control Flow Testing V. Data Flow Testing

Similitud

- Generar paths
- Generar casos de prueba a partir de esos paths

Diferencia

- CFT: Criterios son usados en una etapa previa
- DFT: Criterios son usados después



Anomalía

Desviación o forma anormal de hacer algo

Asignar dos valores a una variable sin usar la primera asignación

Usar una variable sin asignarla primero

Generar un valor y nunca utilizarlo



Tipo 1: Definido y después definido otra vez

- Primera computación es redundante
- Primera computación tiene un error
- Segunda computación tiene un error
- Computación faltante entre los dos

```
:  
x = f1(y)  
x = f2(z)  
:
```

Figure 5.1 Sequence of computations showing data flow anomaly.

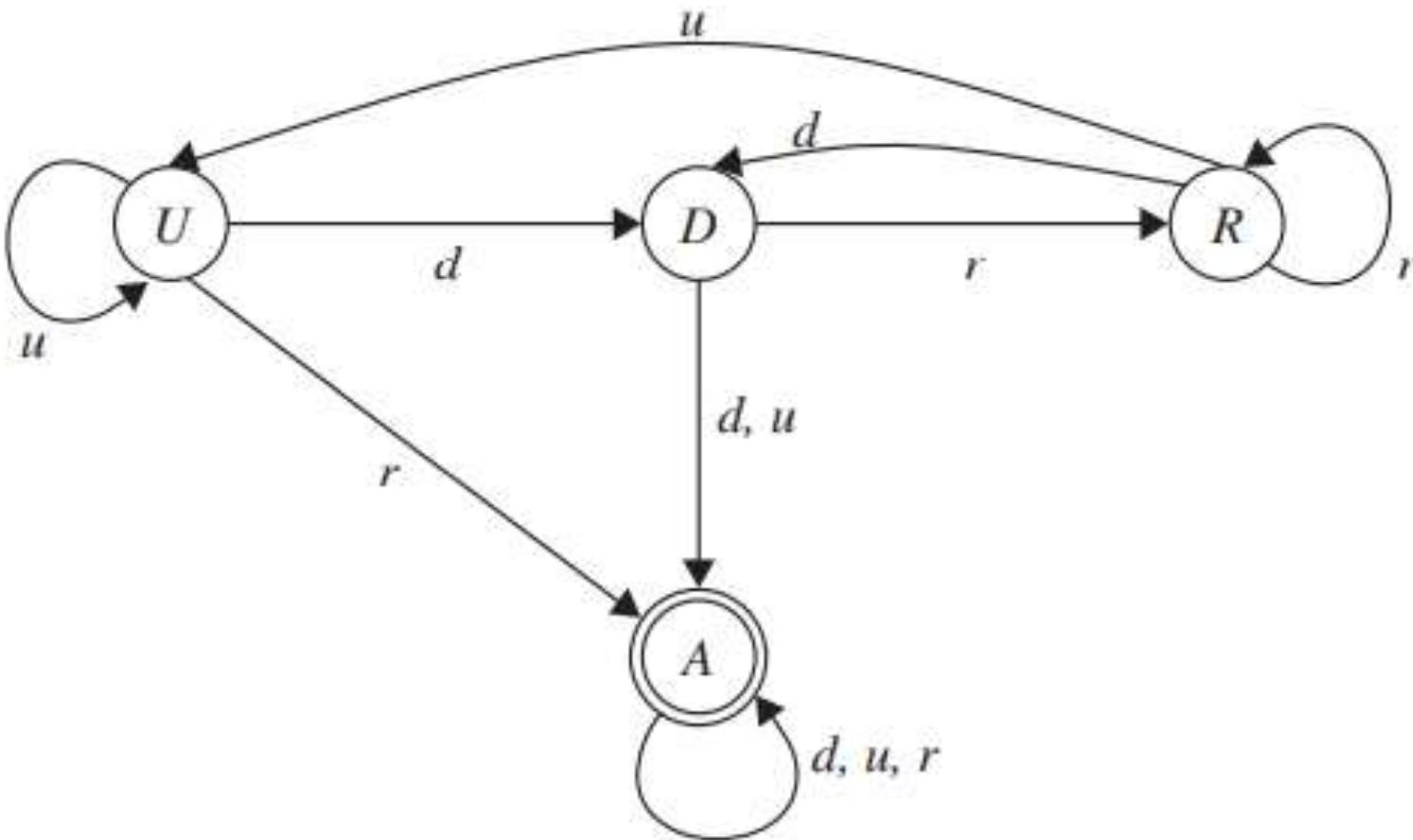
Tipo 2: Indefinido pero referenciado

```
def tipo2(x, y, z):  
    x = x + t + z  
    return x
```

Tipo 3: Definido pero no referenciado

- Definir una variable
- Indefinirla sin usarla

```
def tipo3(x):  
    y = x * 2  
    x = f(x, y)  
  
    return y
```



Legend:

States

U: Undefined

D: Defined but not referenced

R: Defined and referenced

A: Abnormal

Actions

d: Define

r: Reference

u: Undefine

Dinámico

- No podemos estar seguro que una variable fue asignado con el valor correcto si no hay casos de prueba que causan la ejecución de un path desde la asignación hasta la referencia de ella.

Program Path

- Seleccionar paths de entrada – salida con el objetivo de cubrir ciertas definiciones de datos y patrones de uso (data Flow testing criteria)
 1. Dibujar un grafo de data Flow
 2. Seleccionar 1 o más criterios de data Flow testing
 3. Identificar paths en el grafo que satisface los criterios
 4. Derivar expresiones de predicados
 5. Resolver las expresiones para obtener datos de entrada

Data Flow Graph

- Definición: Un valor es movido a una ubicación de memoria de una variable
- Indefinido o muerto: El valor y la ubicación se vuelven indeterminados
- Usado: El valor es obtenido desde la ubicación de memoria
 - Computacional: Un nuevo valor es producido
 - Predicado: Controla el Flow de la ejecución

```
int VarTypes(int x, int y){  
    int i;  
    int *iptr;  
    i = x;  
    iptr = malloc(sizeof(int));  
    *iptr = i + x;  
    if (*iptr > y)  
        return (x);  
    else {  
        iptr = malloc(sizeof(int));  
        *iptr = x + y;  
        return(*iptr);  
    }  
}
```

Figure 5.3 Definition and uses of variables.

Directed graph

- Secuencia de definiciones y c-uses con cada nodo del grafo
- Conjunto de p-uses asociados a cada arista
- El nodo de entrada tiene una definición de cada parámetro y cada variable global
- El nodo de salida produce indefiniciones de cada variable local

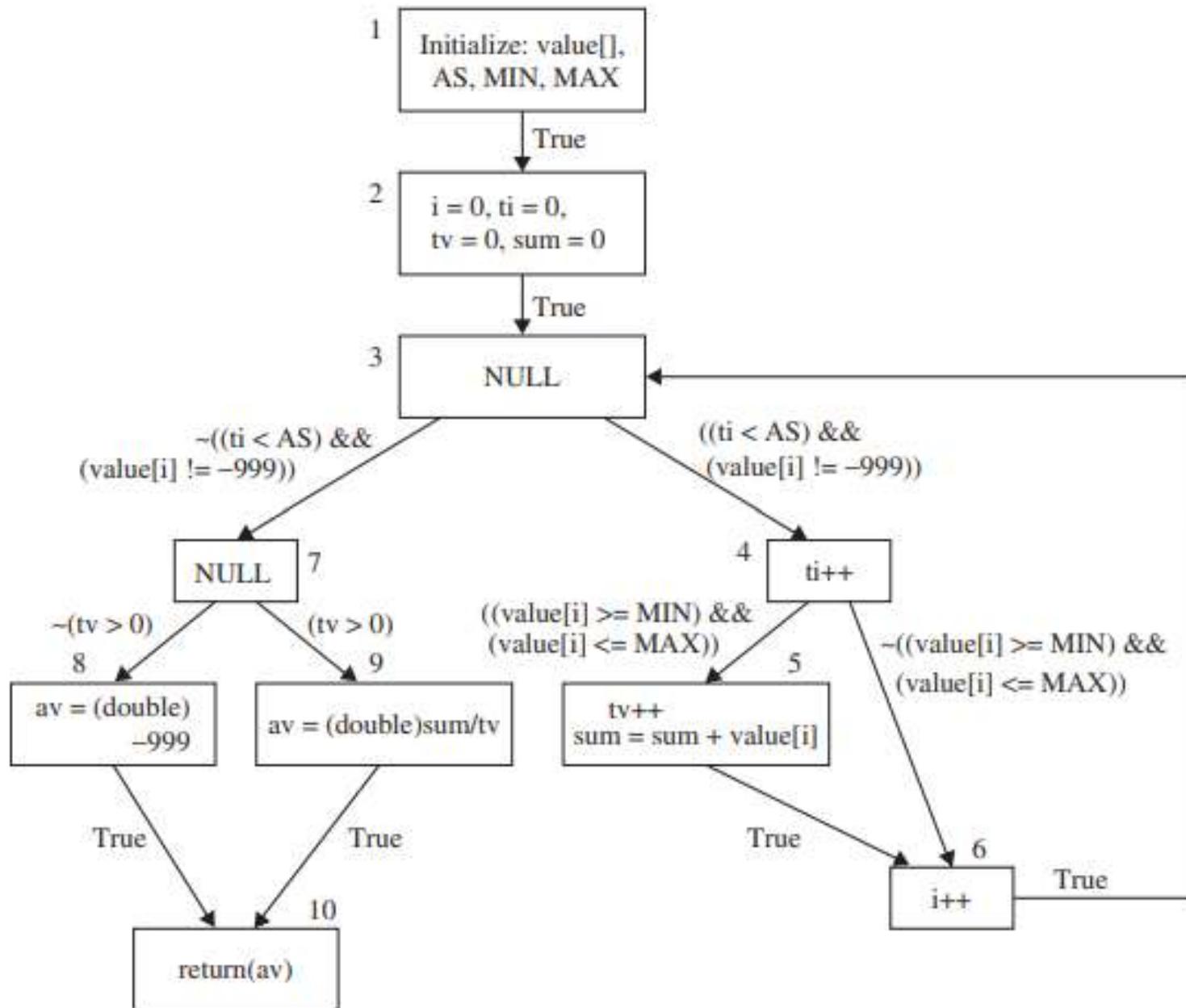


Figure 5.4 Data flow graph of `ReturnAverage()` example.

Términos del Data Flow

- Encontrar paths que contengan pares de definiciones y de uso de variables
- Global c-use: Un c-uso de una variable x en un nodo i , es denominado global c-use si x fue definido en un nodo anterior otro que el nodo i .
- Definition Use Path: Path donde nodo inicial defines una variable x , y el nodo final usas la variable X
- Definition Clear Path: Path donde el nodo inicial es el único que define una variable x
- Definición Global: Un nodo i tiene una definición global de una variable x si dicho nodo define x y existe un def-clear path con respecto al nodo i hacia algún nodo / arista (global c-use / p-use) que use x .

TABLE 5.1 Def() and c-use() Sets of Nodes in Figure 5.4

Nodes i	def(i)	c-use(i)
1	[value, AS, MIN, MAX]	[]
2	{i, ti, tv, sum}	[]
3	{}	{}
4	{ti}	{ti}
5	{tv, sum}	{tv, i, sum, value}
6	{i}	{i}
7	{}	{}
8	{av}	{}
9	{av}	{sum, tv}
10	{}	{av}

TABLE 5.2 Predicates and p-use() Set of Edges in Figure 5.4

Edges (i, j)	predicate(i, j)	p-use(i, j)
(1, 2)	True	{}
(2, 3)	True	{}
(3, 4)	$(ti < AS) \&\& (value[i] \neq -999)$	{i, ti, AS, value}
(4, 5)	$(value[i] \leq MIN) \&\& (value[i] \geq MAX)$	{i, MIN, MAX, value}
(4, 6)	$\neg((value[i] \leq MIN) \&\& (value[i] \geq MAX))$	{i, MIN, MAX, value}
(5, 6)	True	{}
(6, 3)	True	{}
(3, 7)	$\neg((ti < AS) \&\& (value[i] \neq -999))$	{i, ti, AS, value}
(7, 8)	$\neg(tv > 0)$	{tv}
(7, 9)	$(tv > 0)$	{tv}
(8, 10)	True	{}
(9, 10)	True	{}

Tipos de paths

- Simple path: Path en donde todos los nodos (excepto el inicial y el final), son distintos
- Loop-Free Path: Todos los nodos son distintos
- Complete path: Path desde la entrada hasta la salida

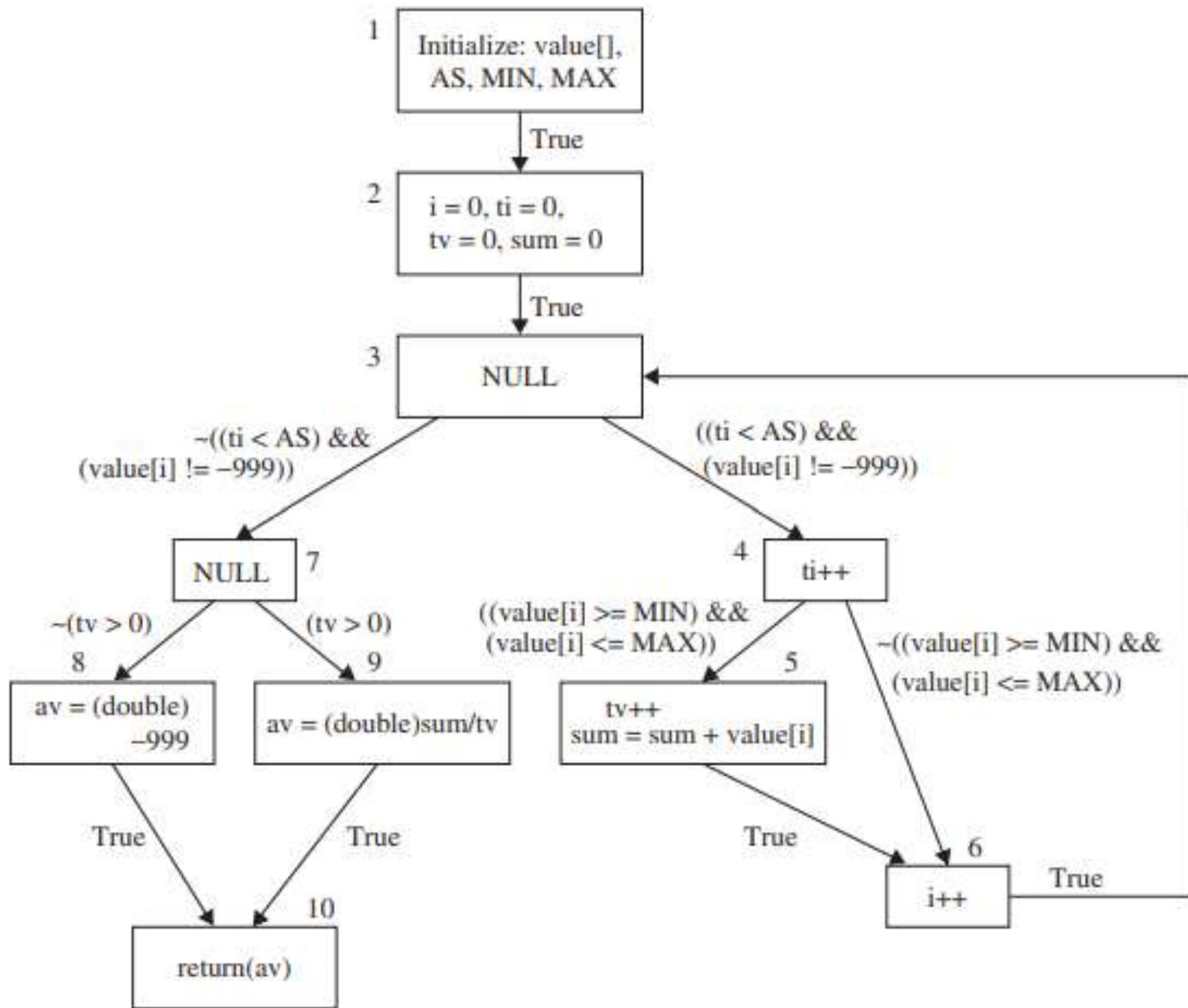
Data Flow Testing Criteria

- All Du-Path
- All-Uses
- All-C-Uses / Some-P-Uses
- All-P-Uses / Some-C-Uses
 - All-Defs
 - All-P-Uses

All-Defs

- Para cada variable x y para cada nodo i en donde existe una definición global de x , selecciona un path completo donde incluya una def-clear path del nodo i hacia
 - Nodo j que contenga un c-use global de x
 - Arista (j,k) teniendo un p-use de x

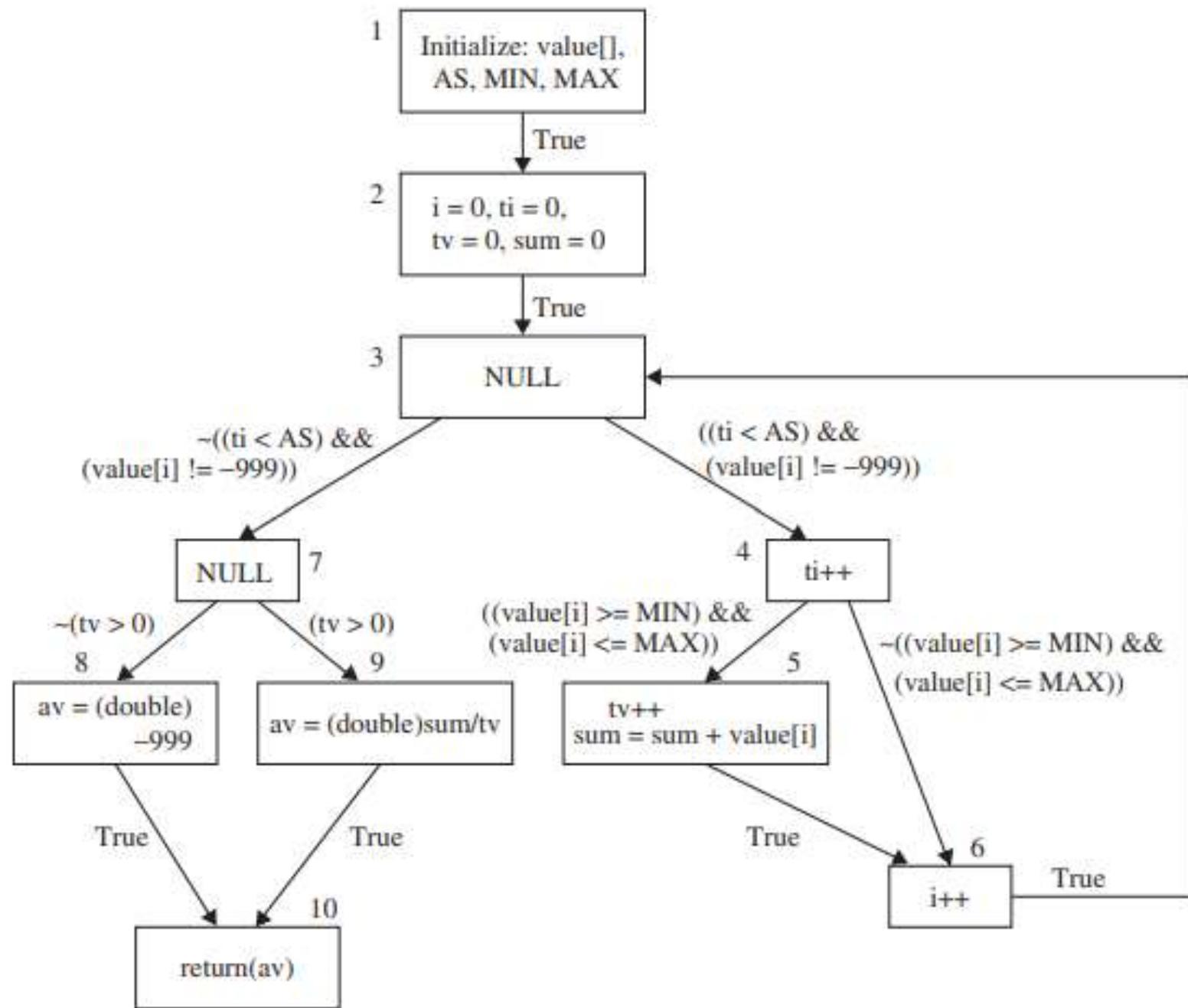
tv

Figure 5.4 Data flow graph of `ReturnAverage()` example.

All C-Uses

- Para cada variable x y para cada nodo i , dado que i contenga una definición global de x , selecciona path completos que incluya def-clear paths desde el nodo i hasta todos los nodos j dado que exista un uso global de x en j .

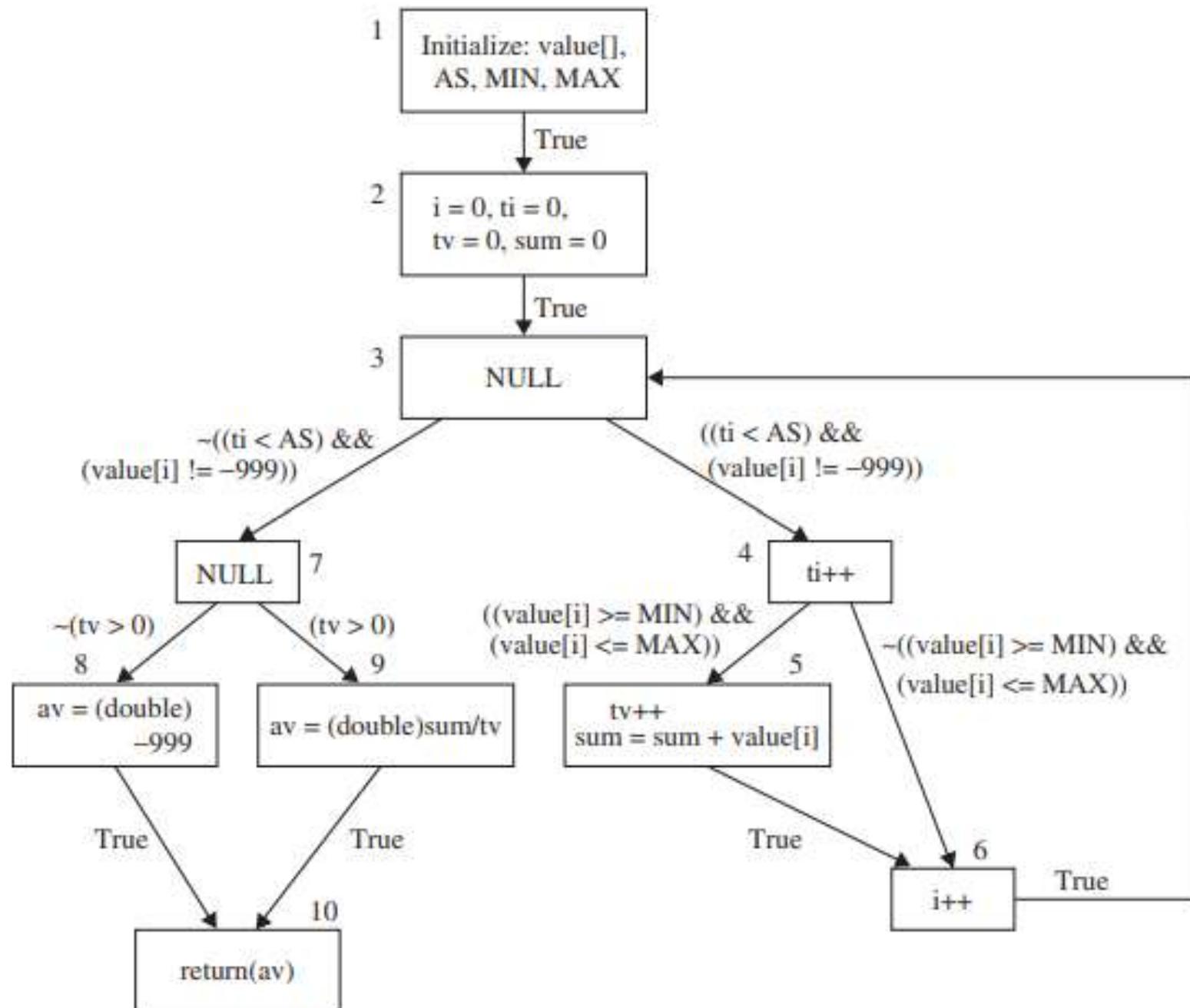
ti

Figure 5.4 Data flow graph of `ReturnAverage()` example.

All p-uses

- Para cada variable x y para cada nodo i dado que x contenga una definición global en i , selecciona paths completos que incluya def-clear paths desde el nodo i hacia las aristasas (j,k) dado que exista un p-use en dicha arista.

tv

Figure 5.4 Data flow graph of `ReturnAverage()` example.

All p-uses / Some c-uses

- X no tiene p-uses
- Para cada variable x y para cada nodo i dado que x contenga una definición global en i, selecciona paths completos que incluya def-clear paths desde el nodo i hacia algunos nodos j dado que exista un uso global c-use de x en j

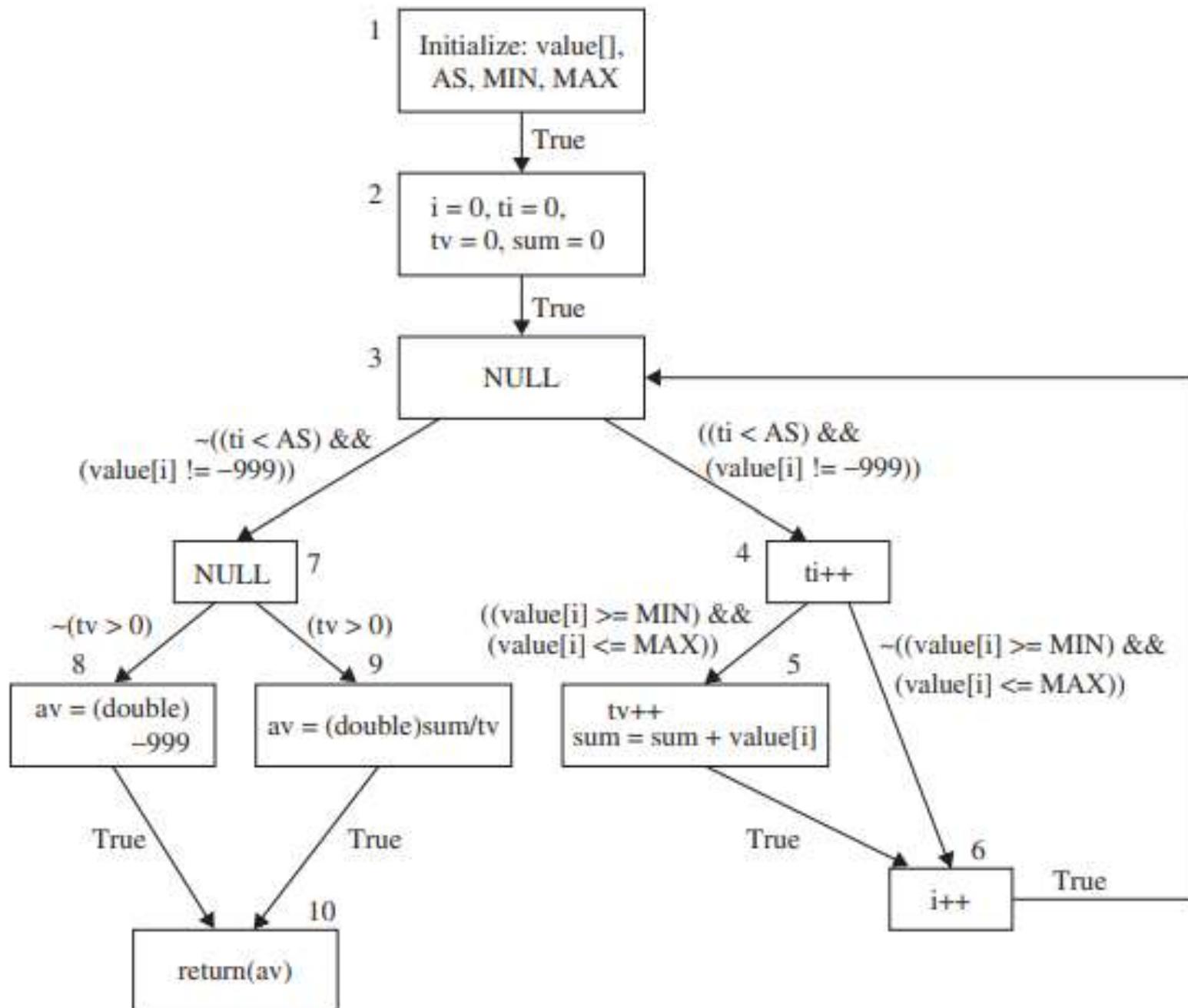
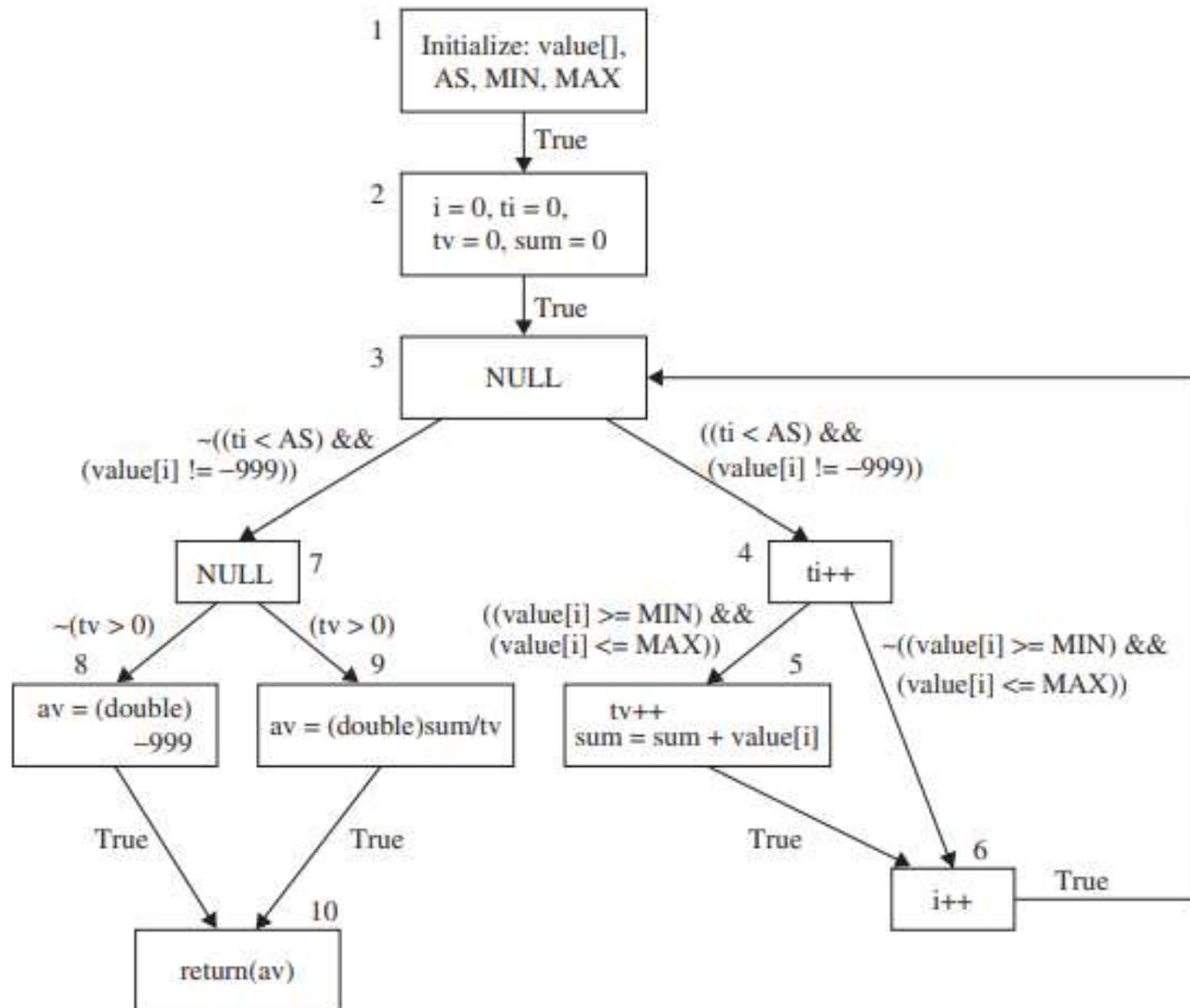


Figure 5.4 Data flow graph of `ReturnAverage()` example.

All-c-uses / Some-p-uses

- X no tiene c-uses
- Para cada variable x y para cada nodo i dado que x contenga una definición global en i, selecciona paths completos que incluya def-clear paths desde el nodo i hacia algunas aristas (j,k) dado que exista un p-use de x en (j,k)

AS

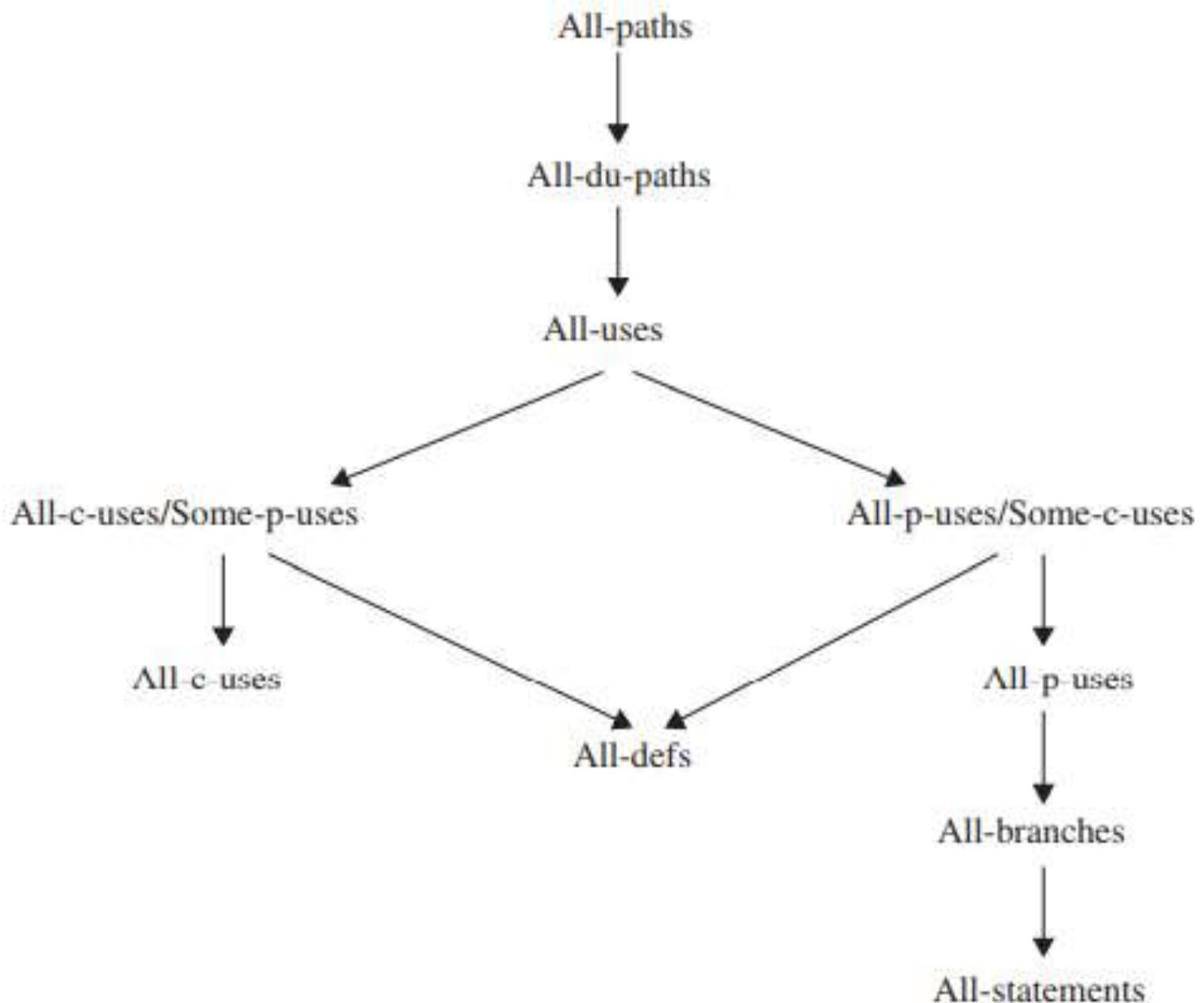
Figure 5.4 Data flow graph of `ReturnAverage()` example.

All uses

- Conjunto de all-p-uses y all-p-uses

All-du-paths

- Para cada variable x y para cada nodo i dado que x contenga una definición global en i , selecciona paths completos que incluya todos los du-paths desde el nodo i hasta
 - Todos los nodos j dado que existe un uso global c-use de x en j , y
 - Todas las aristas (j,k) dado que existe un p-use de x en (j,k)



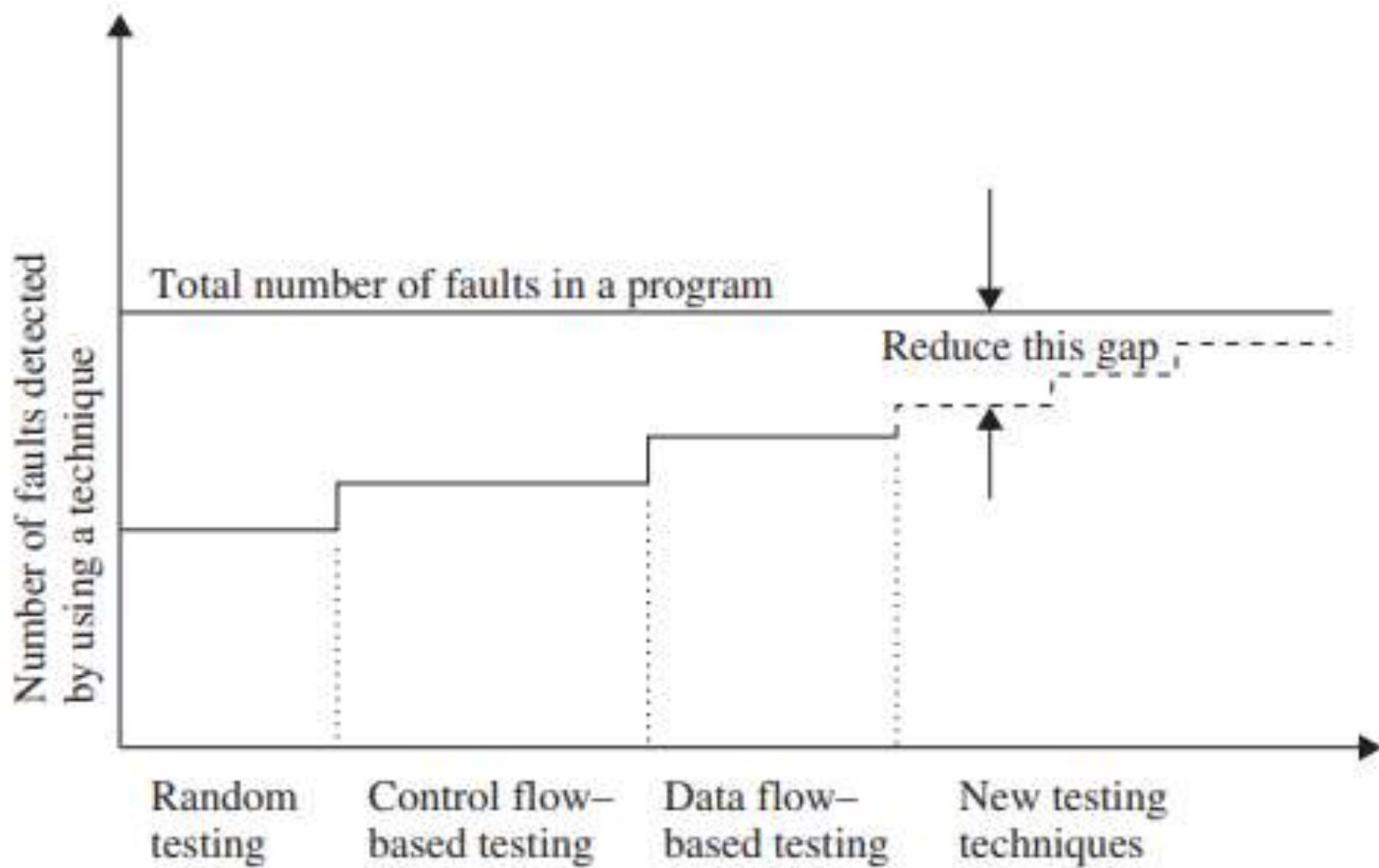


Figure 5.7 Limitation of different fault detection techniques.

Stubs

- Eliminar dependencias
- Asegurar ejecuciones
- Capturar errores
- Rapidez
- Reducir complejidades
- No esperar a que este creada la dependencia

Stubs



INYECIÓN



MOCK / MAGICMOCK



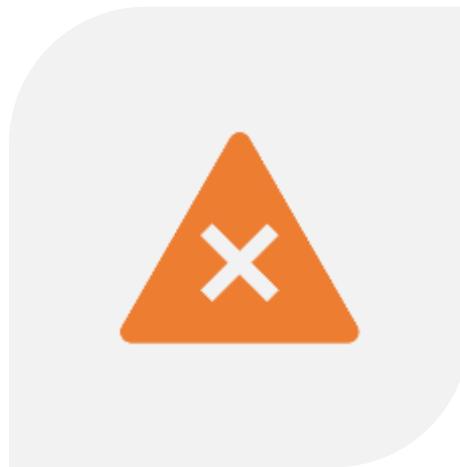
Debugging

Universidad Autónoma de Coahuila

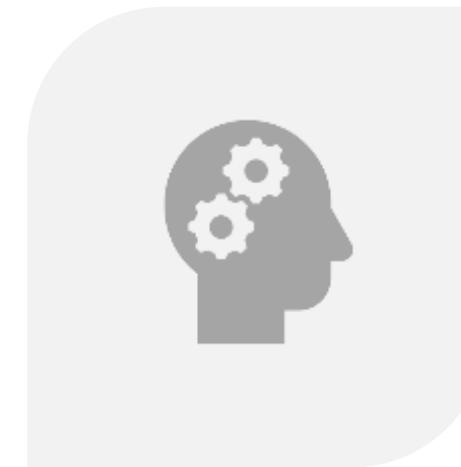
Facultad de Sistemas

Carlos Nassif Trejo García

¿Qué es?



DETERMINAR LA CAUSA DE
LA FALLA



MYERS PROPONE TRES
MÉTODOS

1. Fuerza bruta

Let the computer fix the error

Print statements en todos lados

Trazabilidad de como se ejecuta el código

Debugger dinamicos

Log / memory dump

2. Eliminación de causa

Inducción:

- Recolección de datos
- Datos organizados en base al comportamiento y síntomas
- Se estudia la relación para encontrar un patrón para aislar las causas
- Se crea una hipótesis de causa
- Se usan los datos para probar o rechazar la hipótesis

Deducción

- Listar posibles causas
- Pruebas que aprueben o rechacen cada causa
- Datos de prueba son utilizados para aislar el problema

3. Retrocediendo

Punto de partida en donde fallo

Retrocede la ejecución hasta el punto de donde ocurrió el error

Útil para código pequeño

Dificultades para depurar

- Síntoma y causa geográficamente remotos
- Síntoma puede desaparecer temporalmente
- No es causado por un error
- Causa humana
- Problemas de temporización que de procesamiento
- Difícil de reproducir condiciones de entrada
- Síntoma intermitente

Heurística

Síntomas

Hipótesis

Escenarios de prueba

Priorizar casos de prueba

Ejecutar casos de prueba

Corregir

Documentar

Reproducir los
síntomas

Guías y manuales para entender
los logs

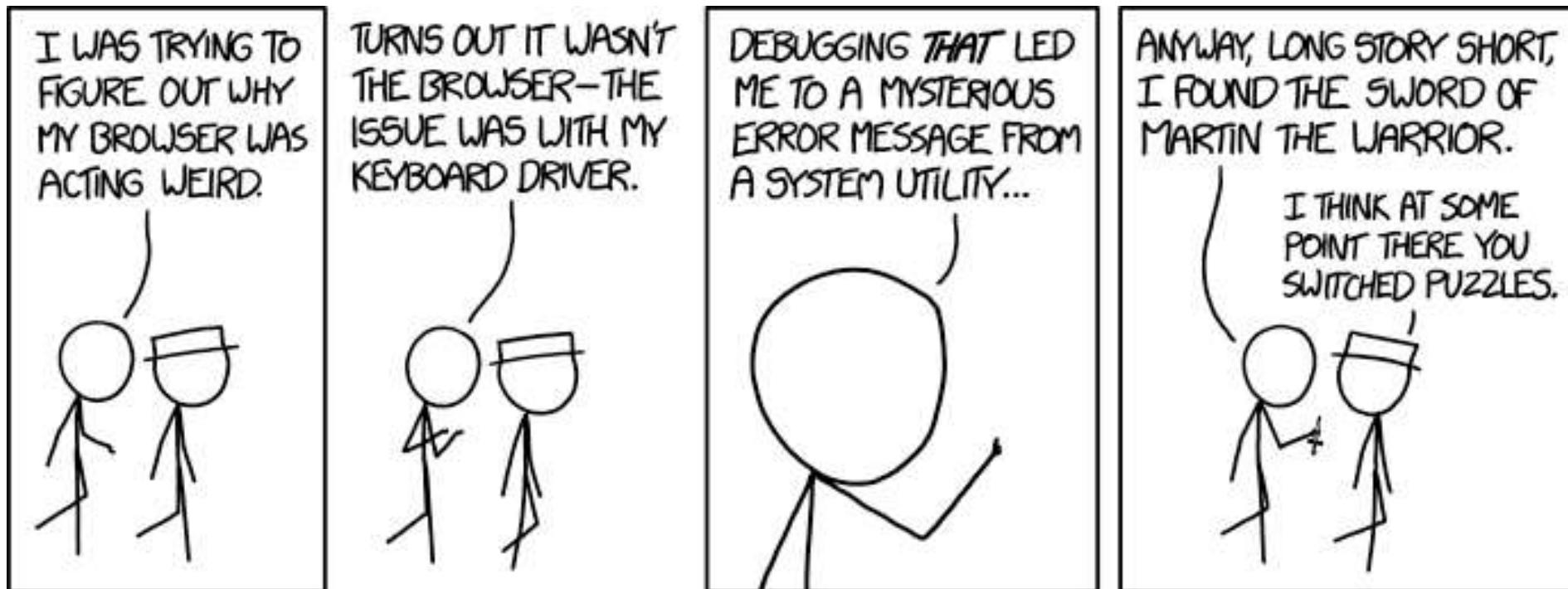
Código de diagnóstico

Análisis causal

- Causa inicial
- Iniciar acciones para eliminar el defecto

Hipótesis

- Formular hipótesis para la causa basado en el análisis causal



Escenarios de prueba

Crea un escenario de pruebas para cada hipótesis que prueben o desacrediten la hipótesis

No destructivos

Bajo costo

Estático:

- Revisión de código
- Documentación

Dinámico

- Ejecución

Priorizar

Prioriza la ejecución de cada caso de prueba.

Relacionada con el caso mas probable

Factor de costo

Ejecutar

Examinar el resultado para nuevas evidencias

Si resulta prometedor, se redefine los datos de prueba para intentar aislar el defecto

Corregir



ARREGLAR EL
CÓDIGO



REVISIÓN DE
CÓDIGO



VOLVER A EJECUTAR
LAS PRUEBAS

Checklist de corrección

¿La causa del error se reproduce en otra parte del programa?

¿Qué error se pudo haber introducido?

¿Cómo pudimos haber evitado este error?

Documentar

Comentarios en el código

Documentación del sistema

Cambios en las pruebas dinámicas

Base de datos de seguimiento de defectos

Herramientas

Auditor de
código

Bound checker /
Fugas de
memoria

Documentadores

Debuggers
interactivos

Coverage

Generadores de
datos

Simuladores

Test Driven Development

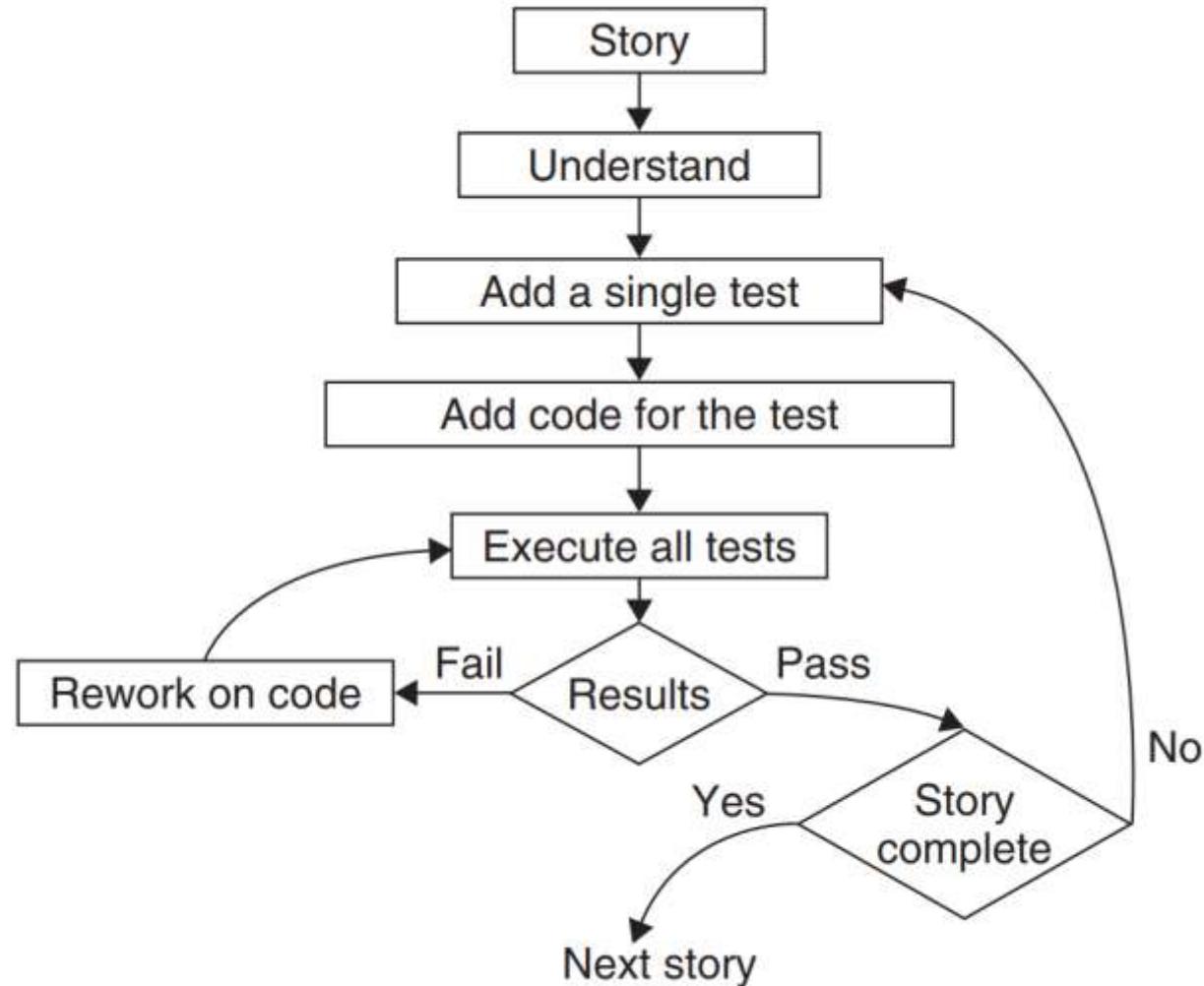


Figure 3.3 Test-first process in XP. (From ref. 24. © 2005 IEEE.)

Domain Testing

- Input Domain
- Program path
- Interest point
- Feasible / infeasible path

Computation Error

- Un par de entradas específicas causan que el programa se ejecute de manera correcta, en el path deseado, pero la salida es incorrecta

Domain Error

- Un par específico de entradas causan que el programa se ejecute de manera incorrecta, path indeseado.

- Dominio: Conjunto de valores de entrada por el cual el programa realiza la misma computación para cada miembro de ese conjunto.
- El programa debe realizar diferentes computaciones en dominios adjuntos
- El programa tiene un error de dominio si el programa realiza clasificaciones de entrada incorrectas

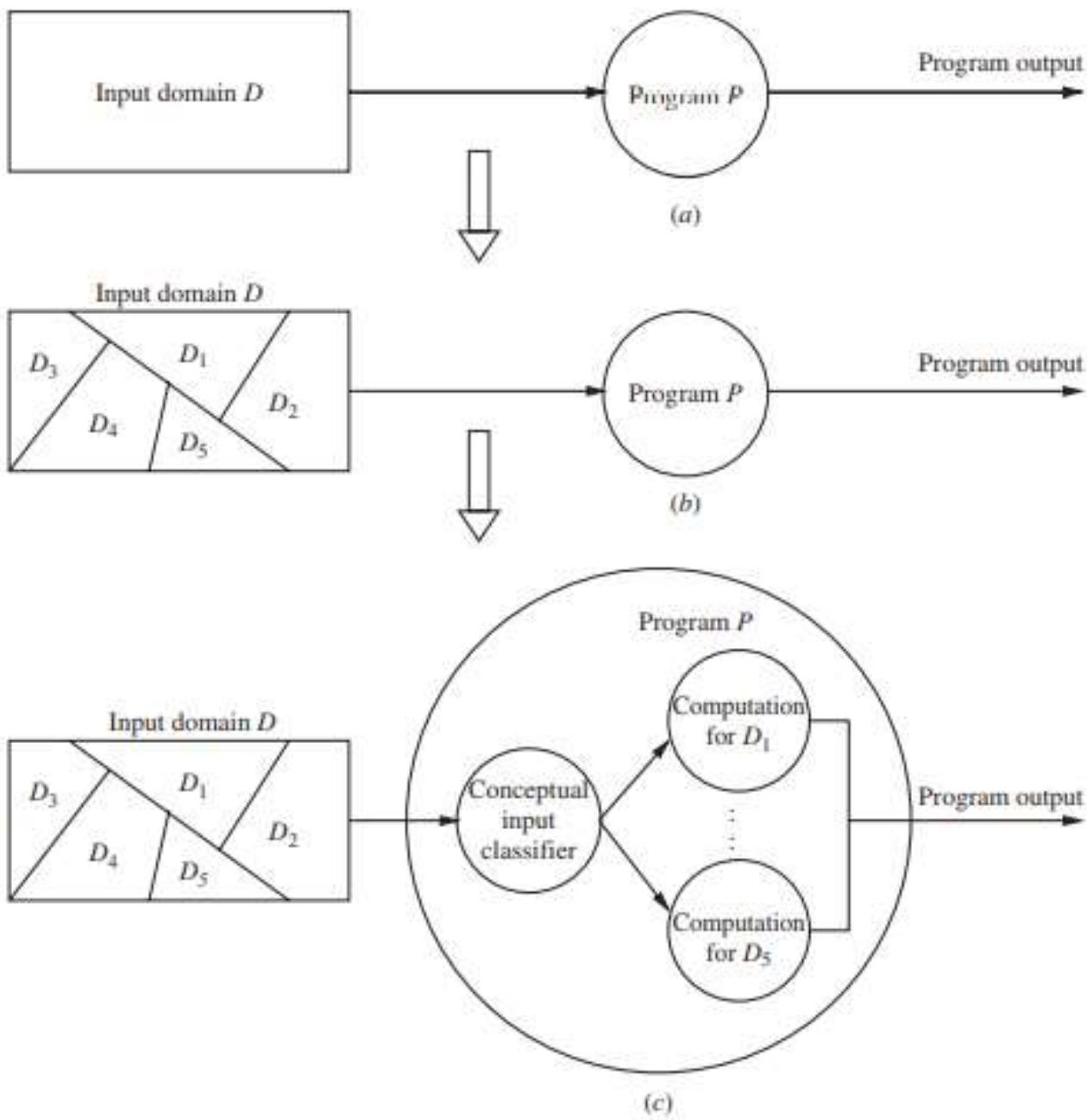


Figure 6.1 Illustration of the concept of program domains.

Conceptos

- Fuentes de dominio:
- Tipos de errores de dominio
- Seleccionar datos de prueba para revelar errores de dominio

Fuentes de dominios

- Dibujar un grafo del flujo de control desde el código fuente
- Encuentra todas las interpretaciones de los predicados
- Analiza los predicados para identificar dominios

```
int codedomain(int x, int y){  
    int c, d, k  
    c = x + y;  
    if (c > 5) d = c - x/2;  
    else         d = c + x/2;  
    if (d >= c + 2) k = x + d/2;  
    else             k = y + d/4;  
    return(k);  
}
```

Figure 6.2 A function to explain program domains.

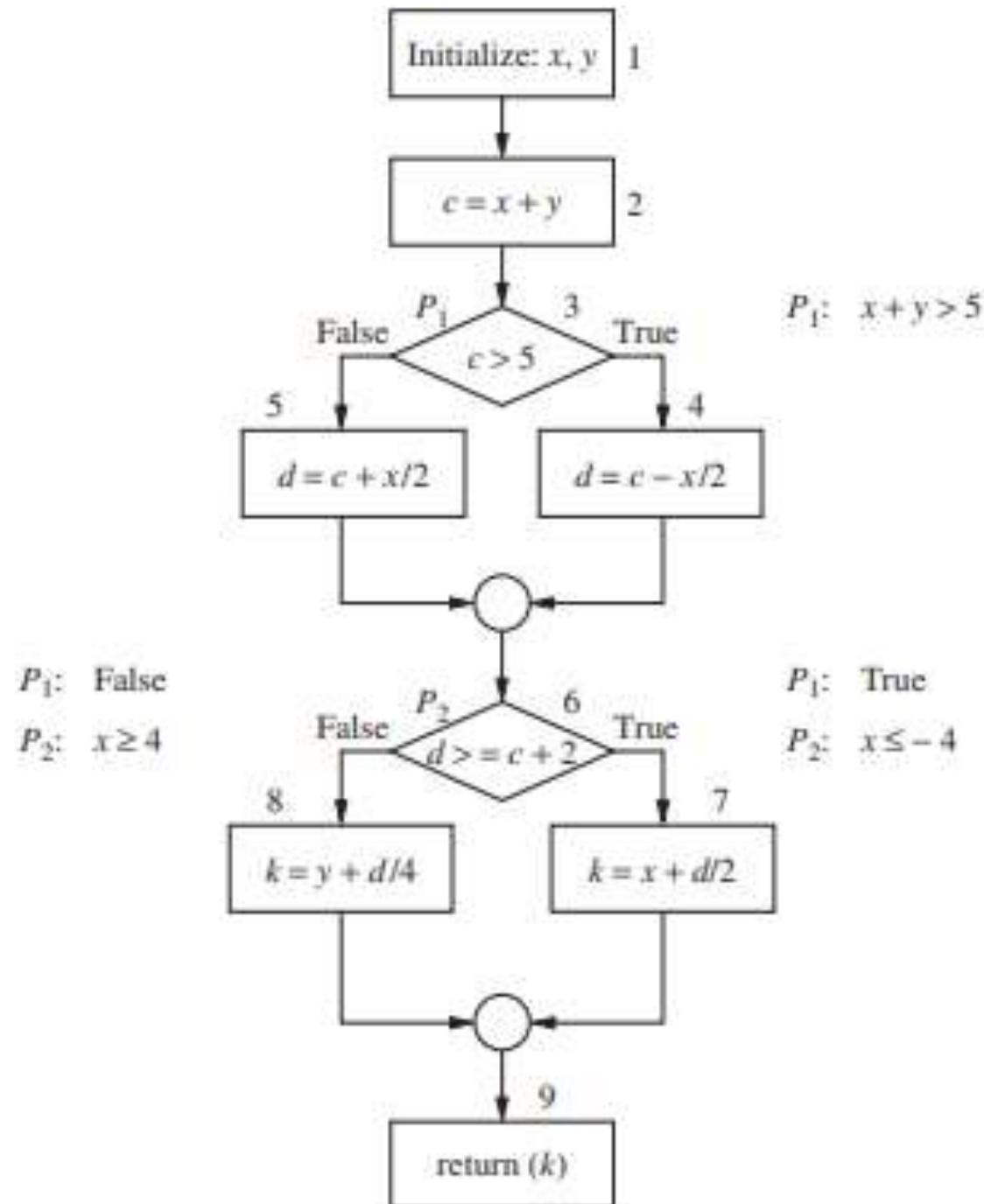
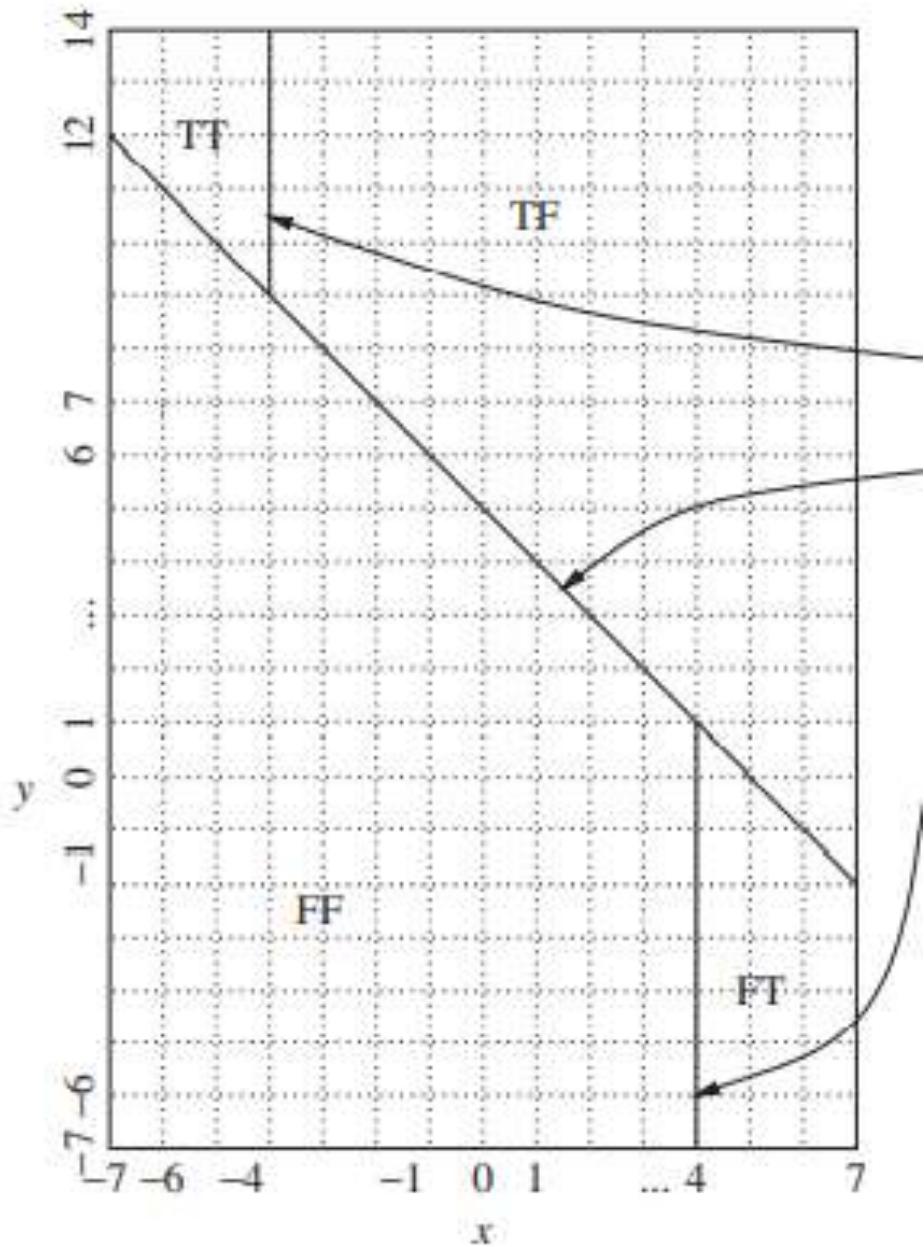


TABLE 6.1 Two Interpretations of Second if() Statement in Figure 6.2

Evaluation of P_1	Interpretation of P_2
True	$x \leq -4$
False	$x \geq 4$



P1:

$x + y > 5$

$\equiv True$

P2:

$x \leq -4$

$\equiv True$

Tipos de errores de dominio

- El dominio esta definido por un conjunto de restricciones denominado “desigualdades en los limites”
- Tipos de limite
 - Limite cerrado
 - Limite abierto
- Tipos de dominio
 - Dominio cerrado
 - Dominio abierto
- Punto extremo
- Dominio adyacente

Un path tendrá un error de dominio si:

- Formulación incorrecta en el predicado del path
- Causado por:
 - Una incorrecta especificación en el predicado
 - Una incorrecta asignación que afecta una variable usada en el predicado

Tipo 1: Error de cierre

- El límite es de tipo abierto cuando debería ser de tipo cerrado, o viceversa

Tipo 2: Error de límite desplazado

- El límite implementado está en forma paralela al límite planeado

Tipo 3: Error de límite inclinado

- Causado por tener valores equivocados en los coeficientes de las variables.

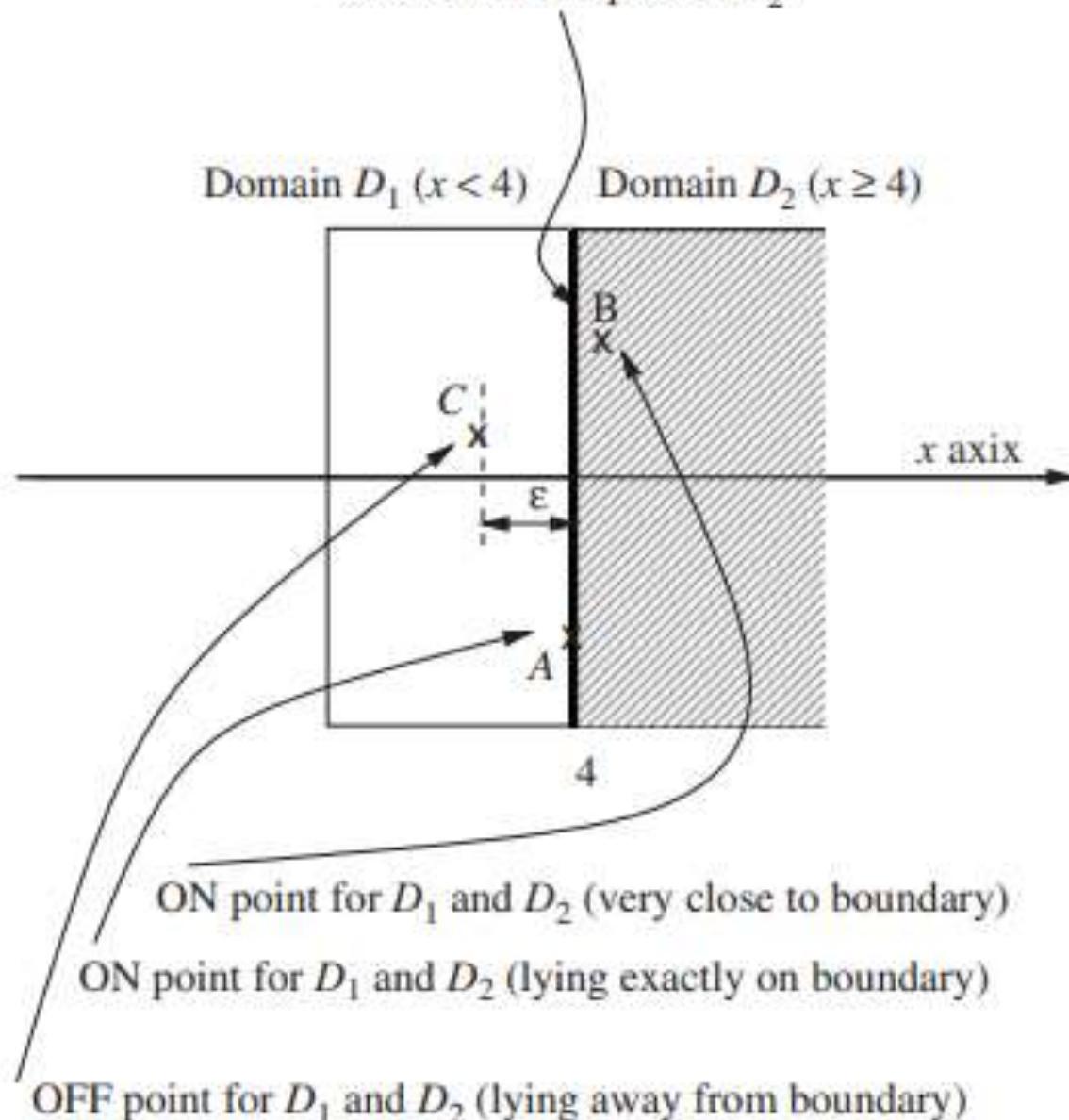
Conclusión

- Puntos de datos en o cerca de los límites son los más sensibles a errores de dominio.
- Tipos
 - En punto
 - Fuera de punto
- Fuera de punto
 - Dominio abierto: punto interno
 - Dominio cerrado: punto externo

Boundary:

Open with respect to D_1

Closed with respect to D_2



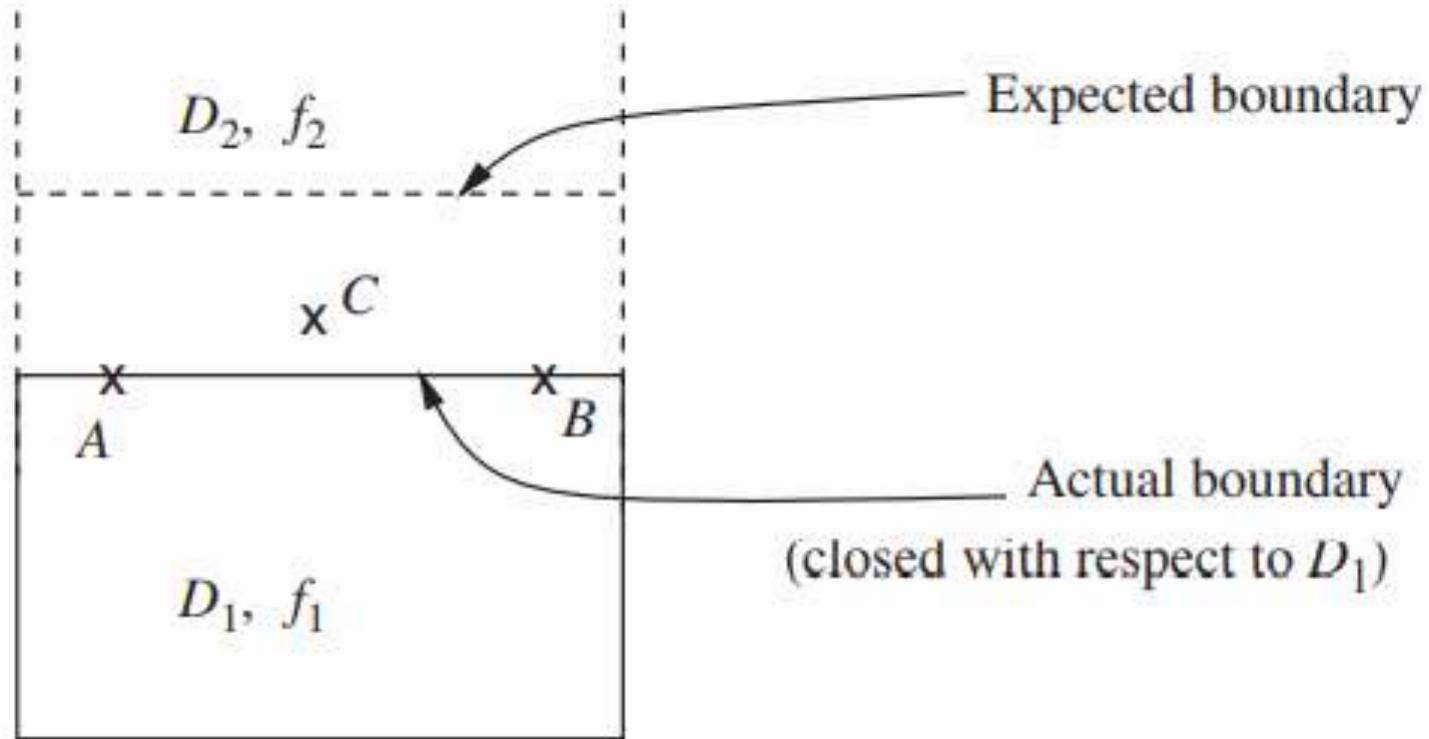
Suposiciones

- Un programa realiza diferentes computaciones un dominios adyacentes
- Predicados de limite son funciones lineares de variables de entrada

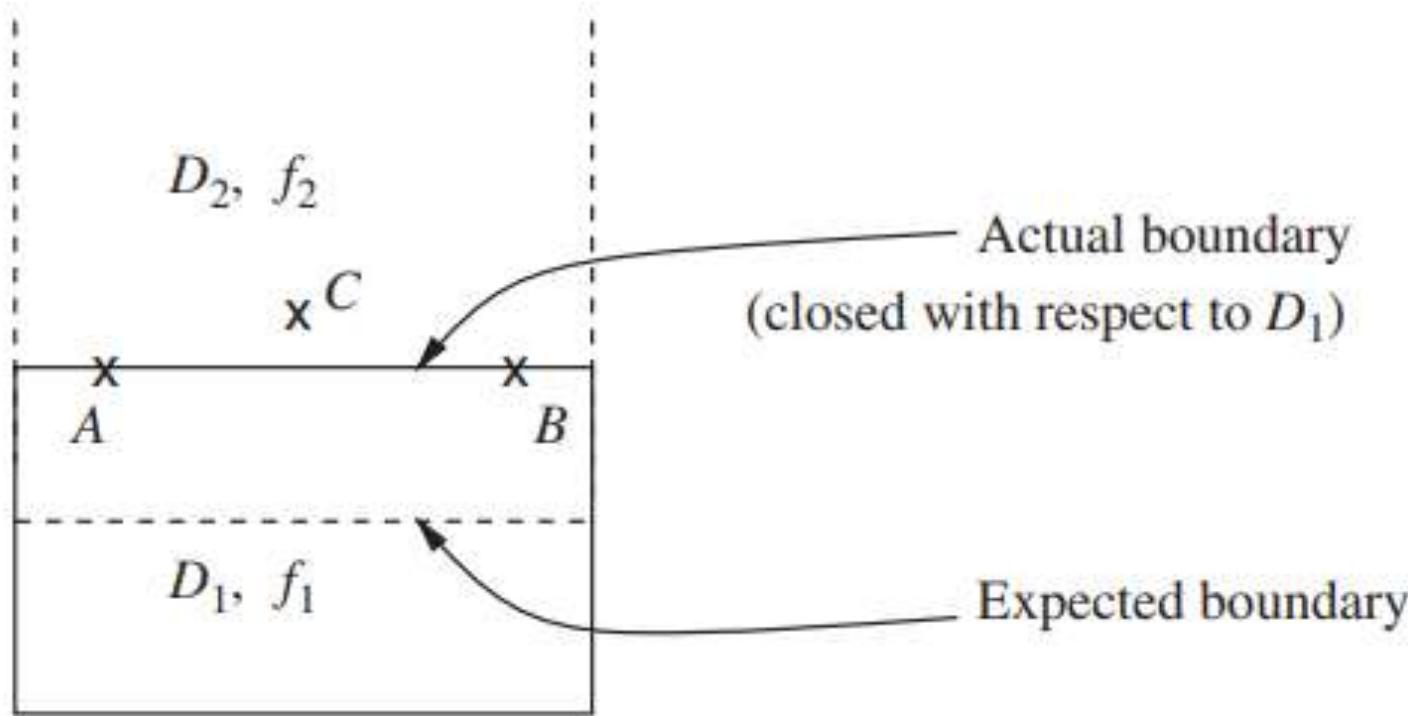
Criterios de selección

- Para cada dominio y para cada límite:
- Selecciona 3 puntos
 - A: ON
 - C: OFF
 - B: ON

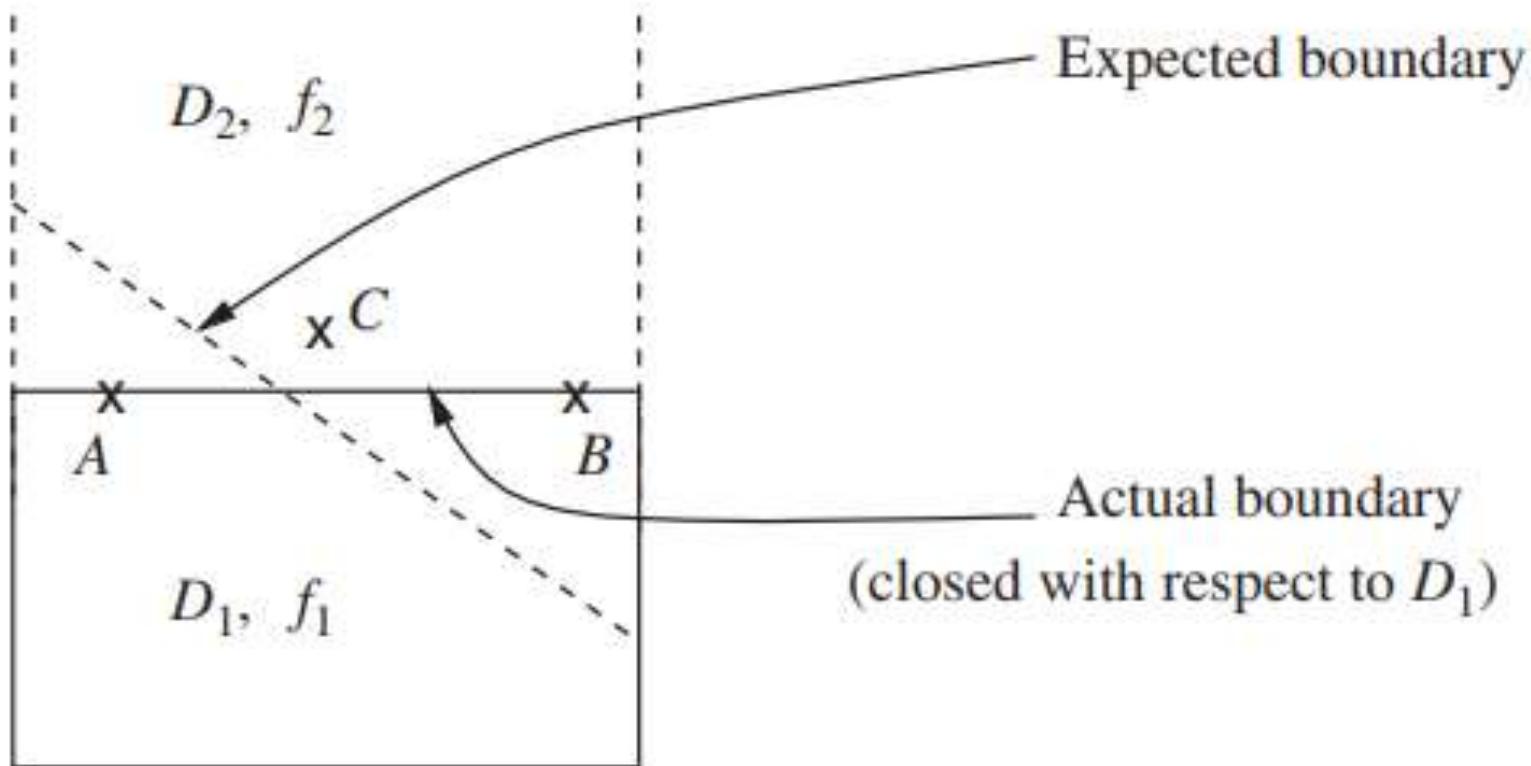
Disminución del dominio



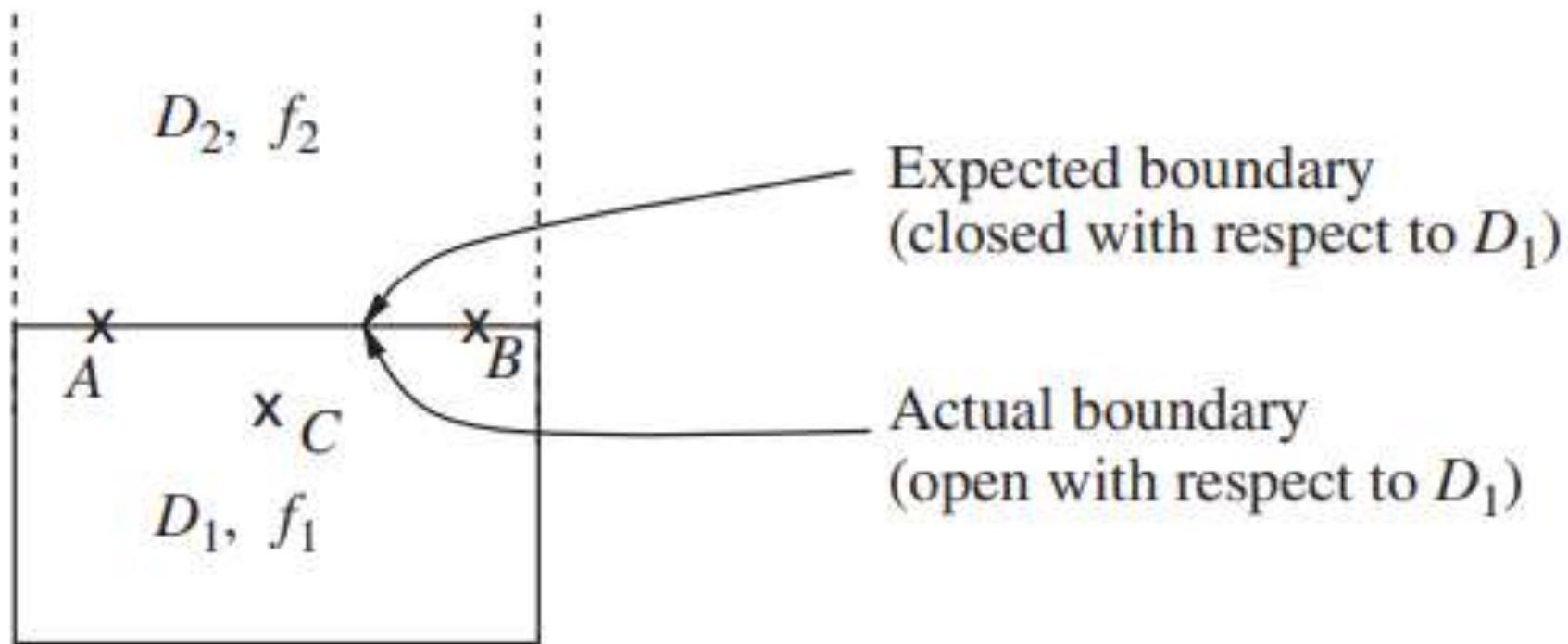
Aumento del dominio



Limite inclinado



Error de cierre



Estrategias de prueba

Universidad Autónoma de Coahuila

Facultad de Sistemas

Calidad y Pruebas de Software

Carlos Nassif Trejo García

Conlleva



Planificación



Diseño



Ejecución



Resultados

Prueba



PEQUEÑO A GRANDE



REVISIONES TÉCNICAS



DESARROLLADORES O
GRUPO
INDEPENDIENTE

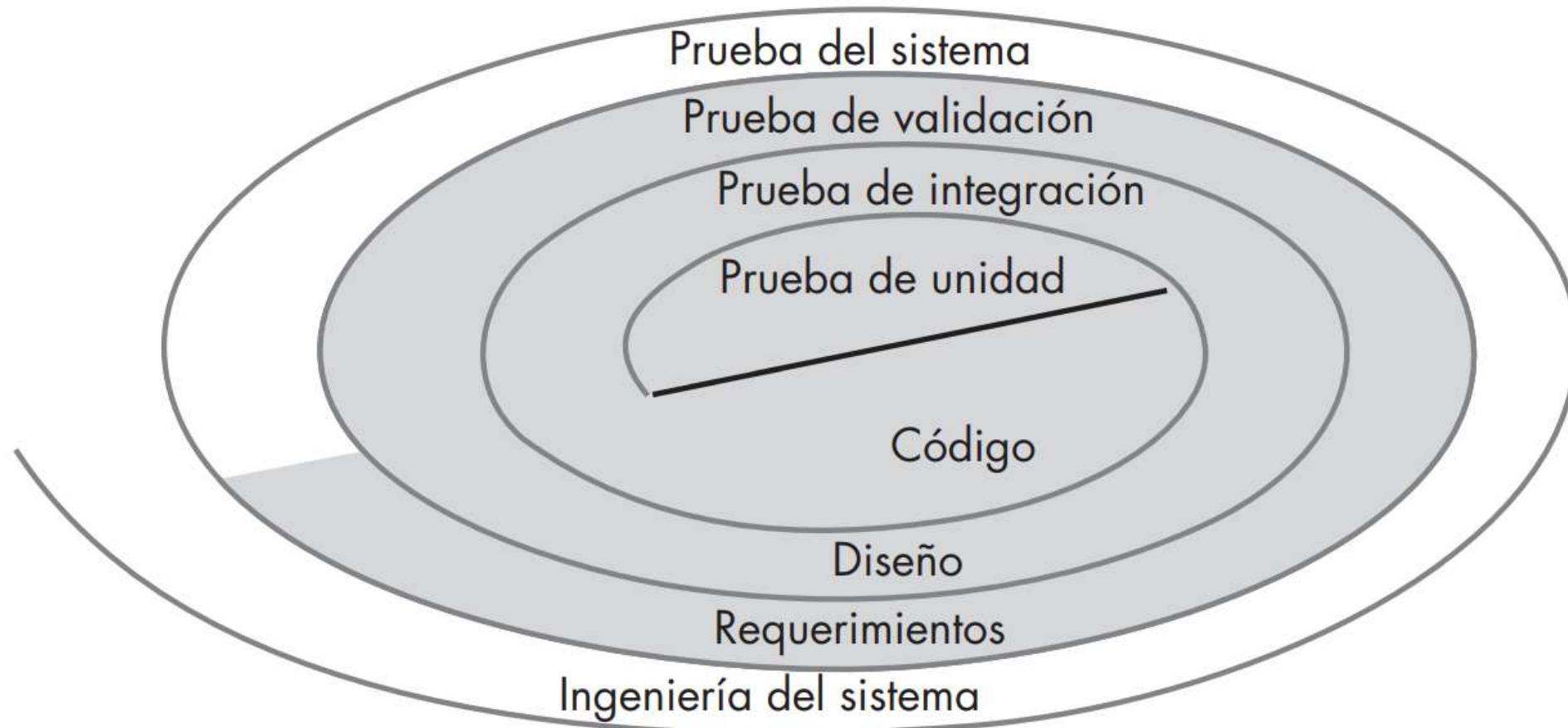


PRUEBA Y
DEPURACIÓN

Psicología



Estrategia



Aspectos a tomar en cuenta

Requerimientos cuantificables

Objetivos de las pruebas

Entender a los usuarios

Software robusto

Revisiones técnicas

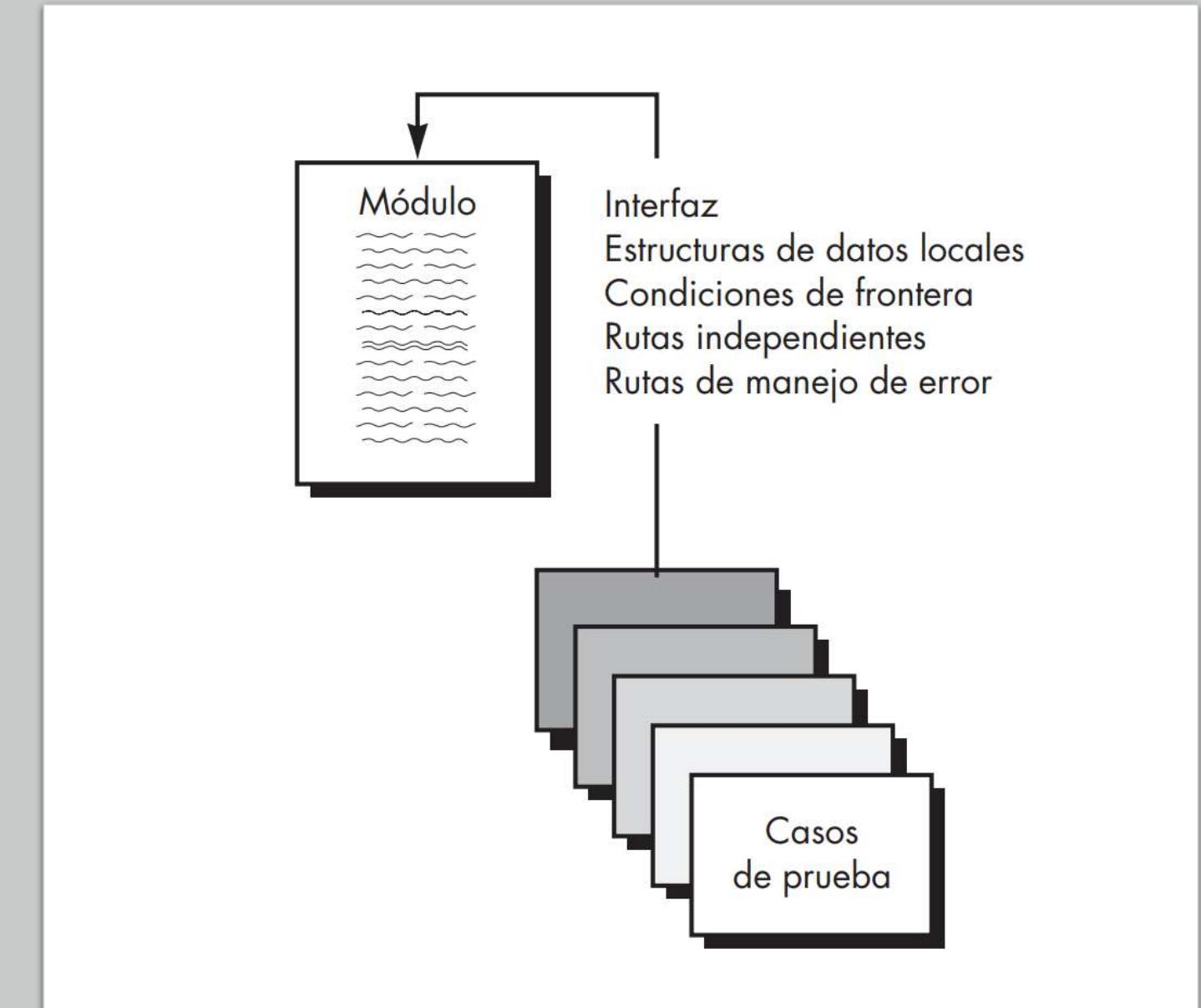
Mejora continua

¿Qué tanto
probar?



Pruebas de unidad

- Unidad mas pequeña
- Lógica interna



Manejar errores

Descripción de error

El sistema debe
manejar el error

Procesamiento
excepción-condición

Inexistente

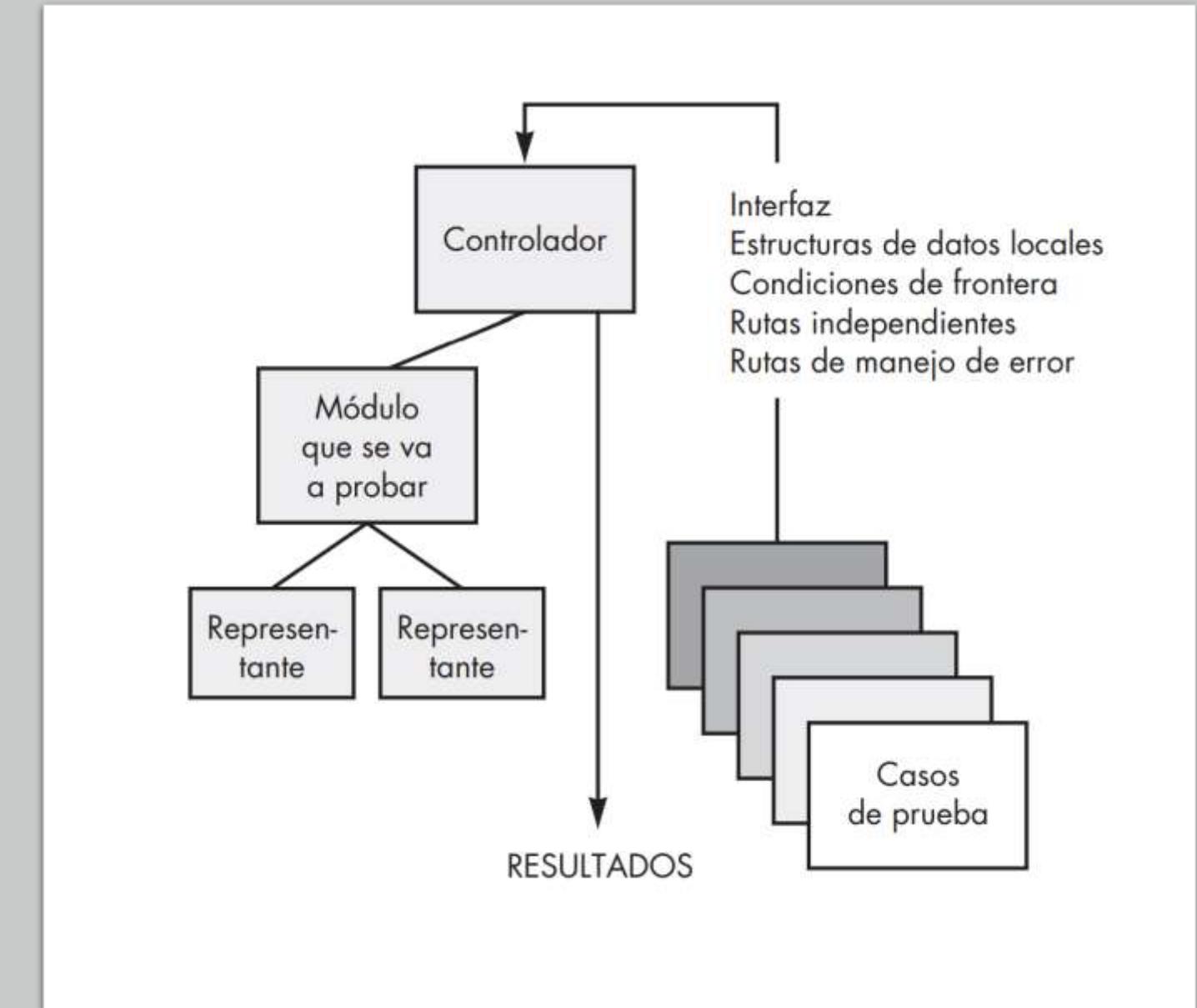
Erróneo

Causa

Auxiliar

Procedimientos

- Pruebas de unidad junto con el código
- Resultados esperados



Estático

Código no se ejecuta

Problemas potenciales

Inspección: Revisión grupal paso a paso donde cada uno es revisado a través de un criterio definido

Walkthrough: El autor explica al equipo la ejecución de su código usando escenarios definidos

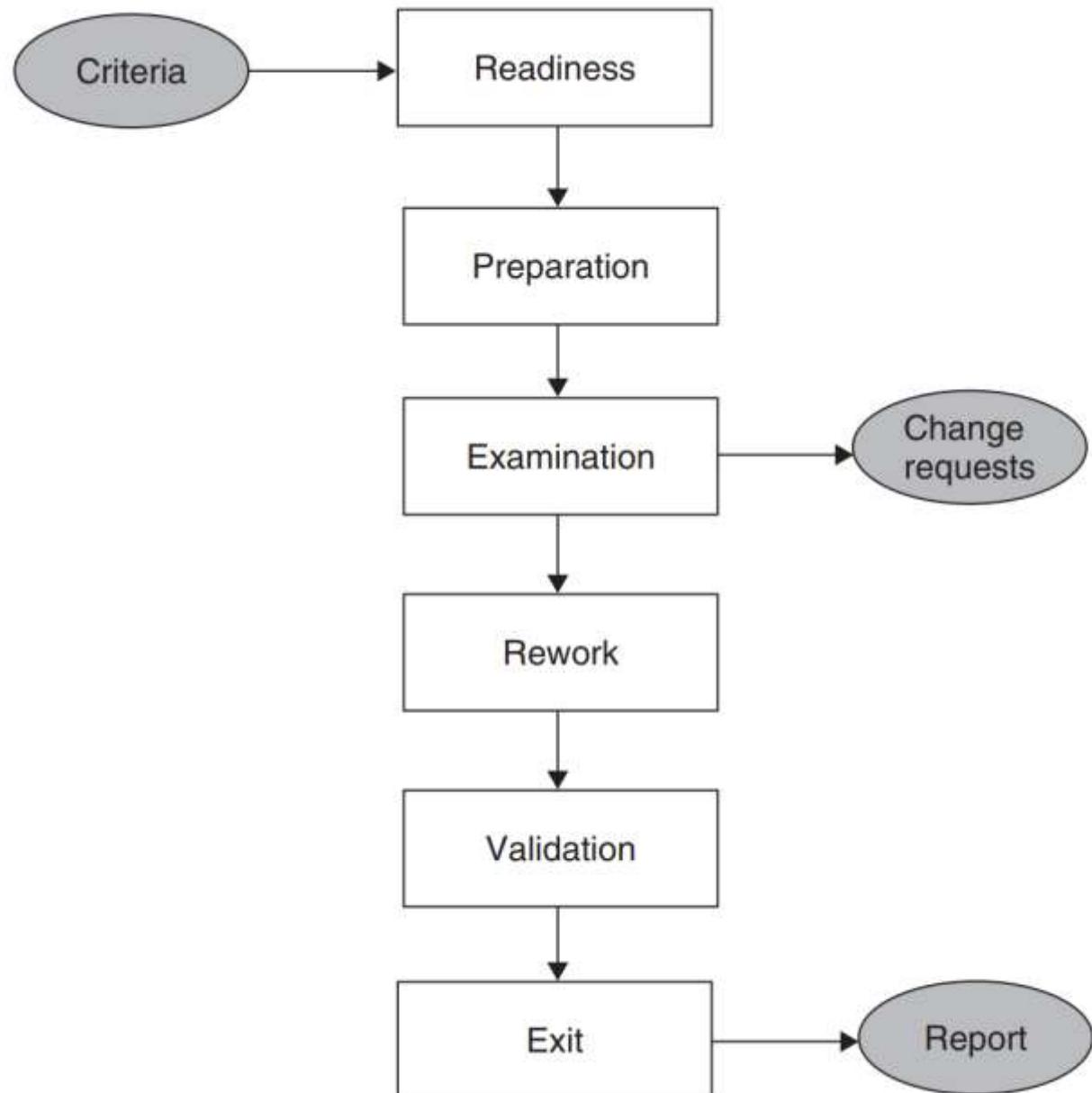


Figure 3.1 Steps in the code review process.

Listo para ser revisado

- Completeness
- Minimal Functionality
- Readability
- Complexity
- Requirements and Design Documents
- 125 lines of code per hour

TABLE 3.1 Hierarchy of System Documents

Requirement: High-level marketing or product proposal.

Functional specification: Software engineering response to the marketing p

High-level design: Overall system architecture.

Low-level design: Detailed specification of the modules within the architec

Programming: Coding of the modules.

Roles

Moderador

Autor

Presentador

Registrador

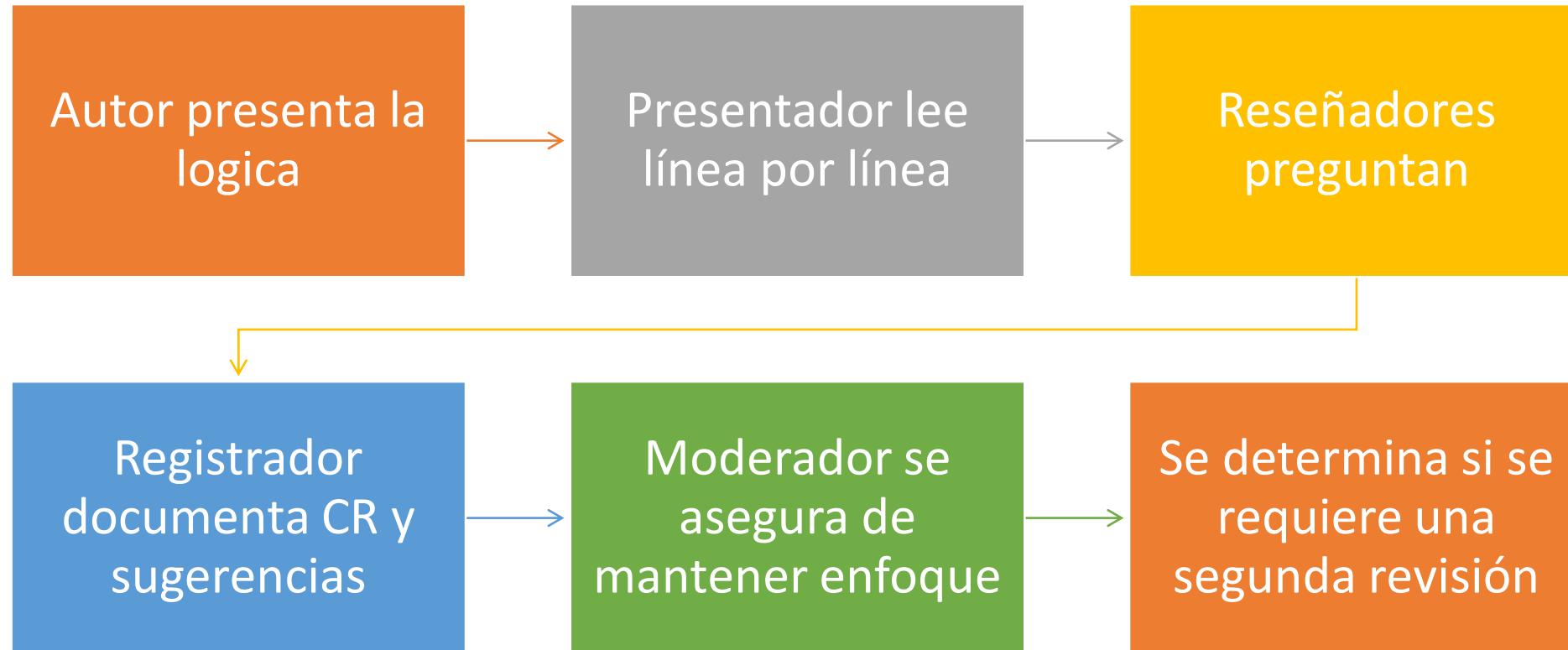
Reseñadores

Observadores

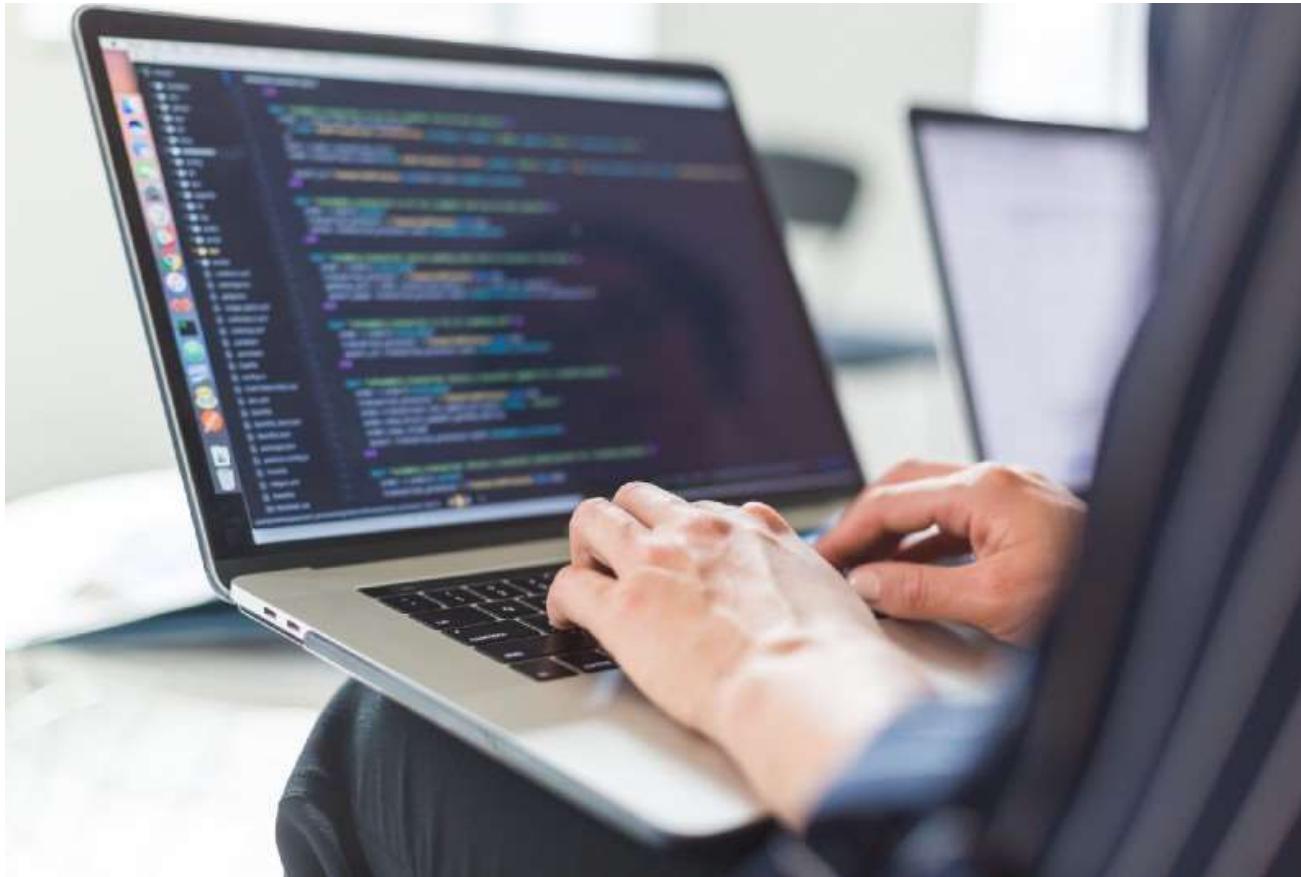
Preparación

- Reseñador:
 - Preguntas
 - CR
 - Oportunidades de mejora

Examinacion



Retrabajo



Validación



Salida

Cada línea fue revisada

Consenso

Cambios revisados

Reporte enviado a todos

Métricas

- LOC: Lines of Code per hour
- KLOC: CR generados por 1000 líneas de código
- CR per hour
- CR per Project
- Num. of hours per Project

Prevención de defectos

Instrumentation code

Controles estándares de errores

Código termina para todos los valores de entrada

Overflow y underflow

Mensajes de error

Validar entradas

Assertions

Comparar entrada y salidas

Loops infinitos

Prueba dinámica

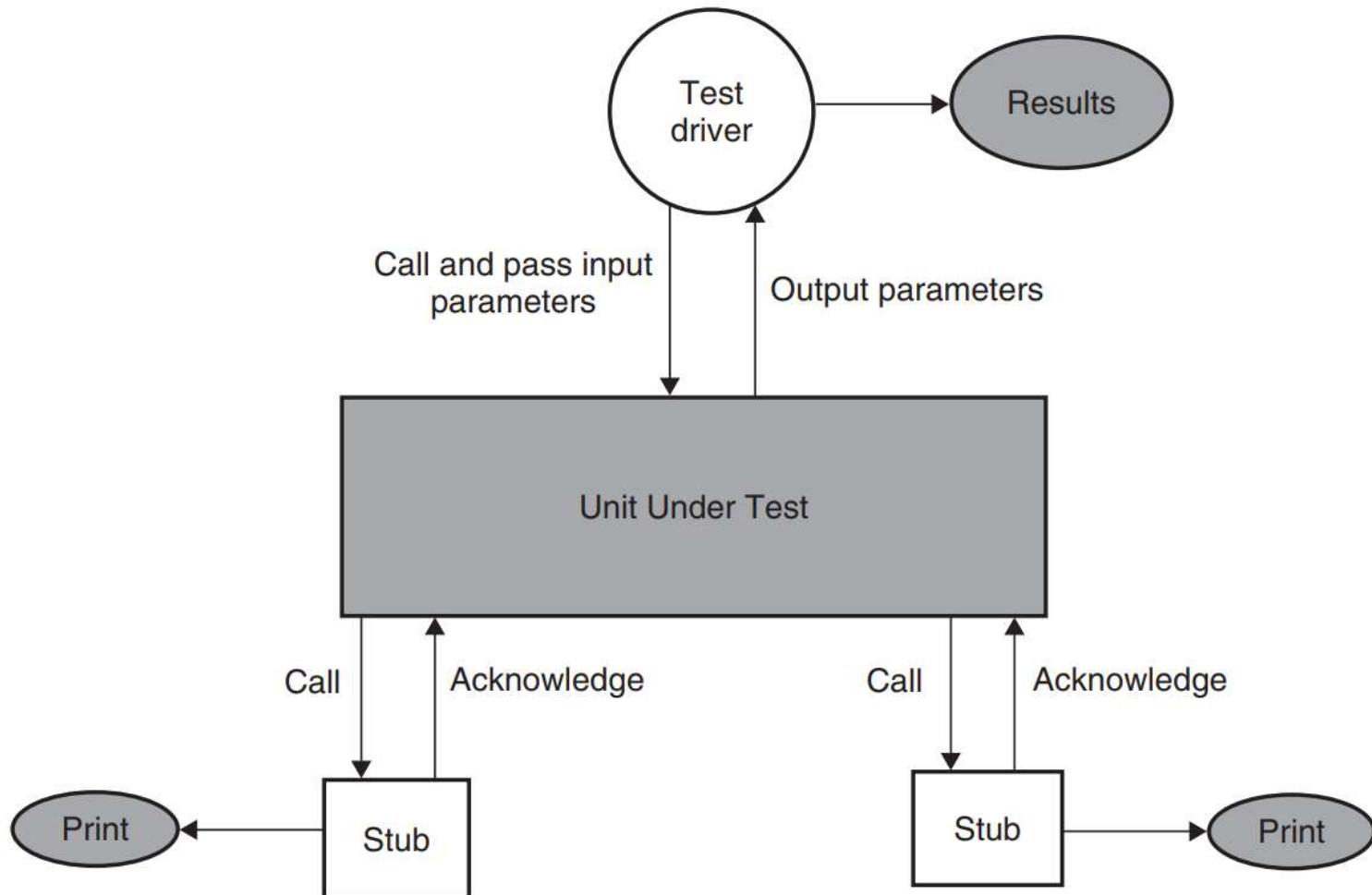


Figure 3.2 Dynamic unit test environment.

Ejemplo...

- A4 Unit Testing
- A5 TDD

Mutation Testing

TDD: Test Driven Development

Test Data

- To be continued

Calidad y Pruebas de Software

Universidad Autónoma de Coahuila
Facultad de Sistemas

Carlos Nassif Trejo García



Contacto

c-trejo@uadec.edu.mx

844 2058966



Redes

<https://github.com/CarlosTrejo2308>

<https://www.linkedin.com/in/carlosnassif/>

Evaluación

- Parcial 1: 25%
- Parcial 2: 25%
- Proyecto Final: 25%
- Examen Final: 25%

Cada parcial se evaluará de la siguiente manera:

- Asistencia y participación: 10%
- Presentación: 15%
- Tareas: 15%
- Proyecto: 25%
- Examen: 35%

Temario

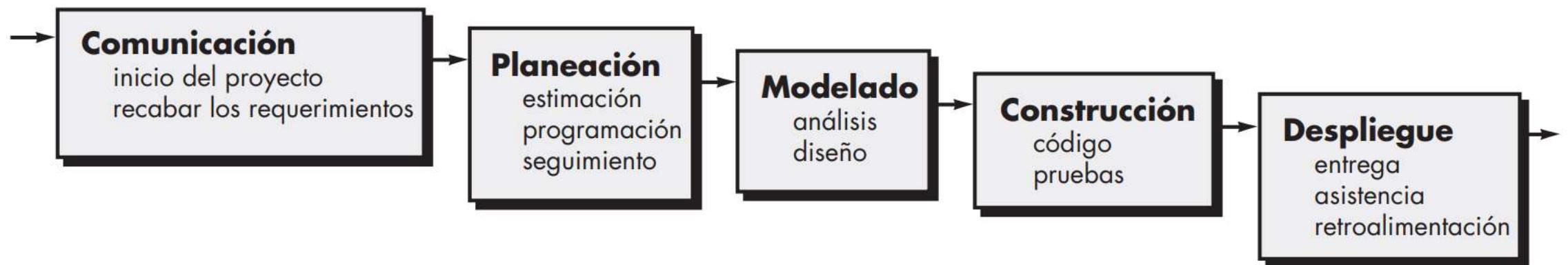
- **U1. Conceptos Básicos de Calidad**
 - Presentación
 - Definición de Calidad
 - Definición de calidad de software
 - Quién define la calidad
 - Importancia de la calidad
 - La calidad y el mundo globalizado
 - Calidad de vida
 - Calidad total
- **U2. Aseguramiento de la calidad del software (SQA)**
 - Relación de la Ingeniería del software con SQA. Definición y propósito del SQA.
 - Problemas que resuelve la SQA.
 - Calidad del software en el ciclo de vida del mismo.
 - Roles y responsabilidades de los equipos de desarrollo.
 - Habilidades y capacidades del personal del SQA.
 - Actividades del SQA.
 - Métodos y herramientas.

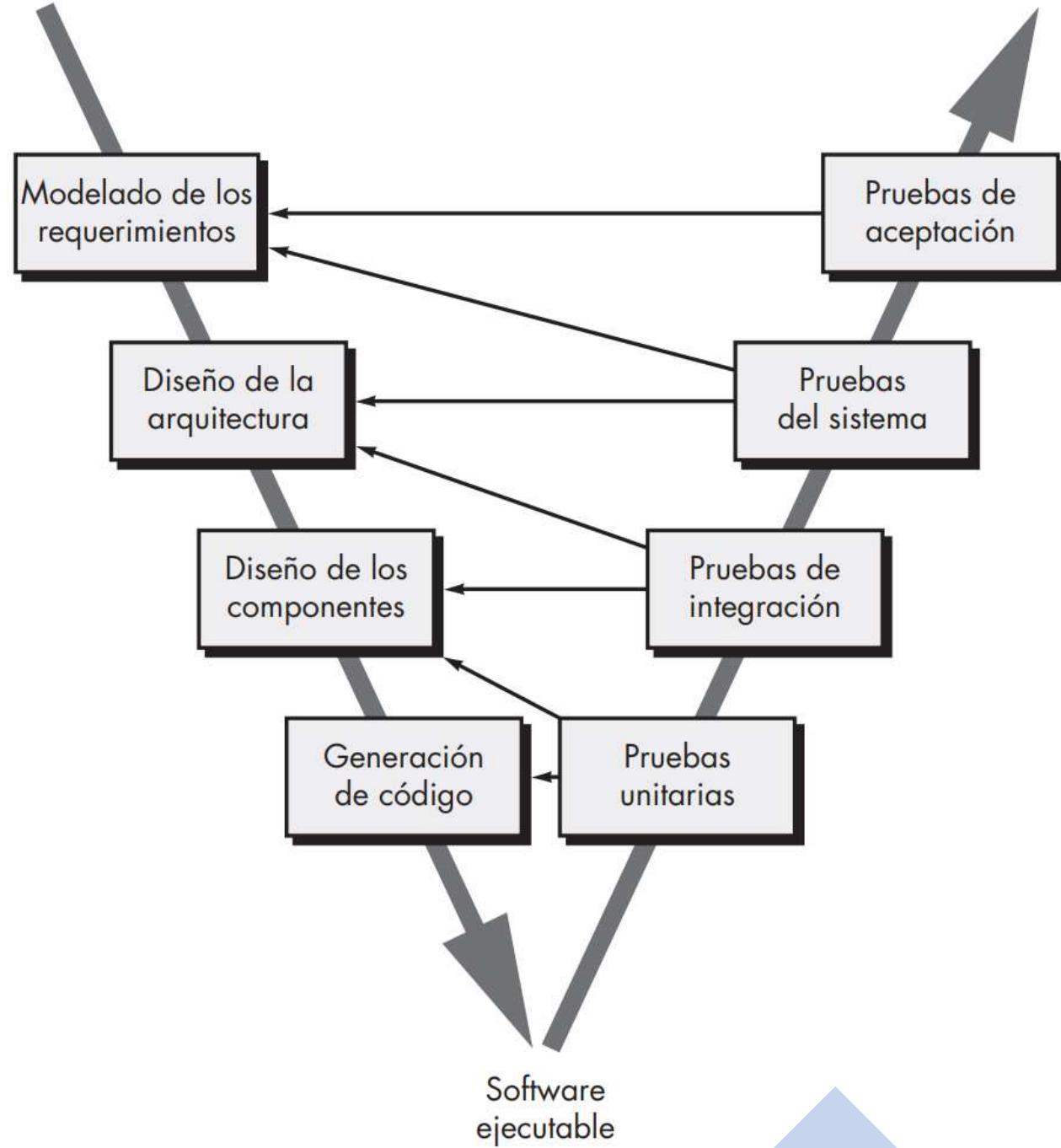
Temario

- **U3. Estándares de calidad aplicados al software.**
 - ISO
 - SPICE
 - CMM
 - Definición del modelo.
 - Nivel inicial.
 - Nivel repetido.
 - Nivel definido.
 - Nivel administrado.
 - Nivel optimizado.
- **U4. Calidad enfocada al desarrollo de software**
 - Qué es la calidad del software.
 - Cómo obtener calidad de software (métodos, metodologías, estándares).
 - Cómo controlar la calidad del software.
 - Costo de la calidad del software.
 - Nomenclatura y certificación ISO 9001:2000.
 - La norma ISO/IEC 9126.
 - Análisis de factores que determinan la calidad del software.
 - Análisis del proceso del ciclo de vida del software.
 - Funciones de evaluación del software.

Un poco de historia

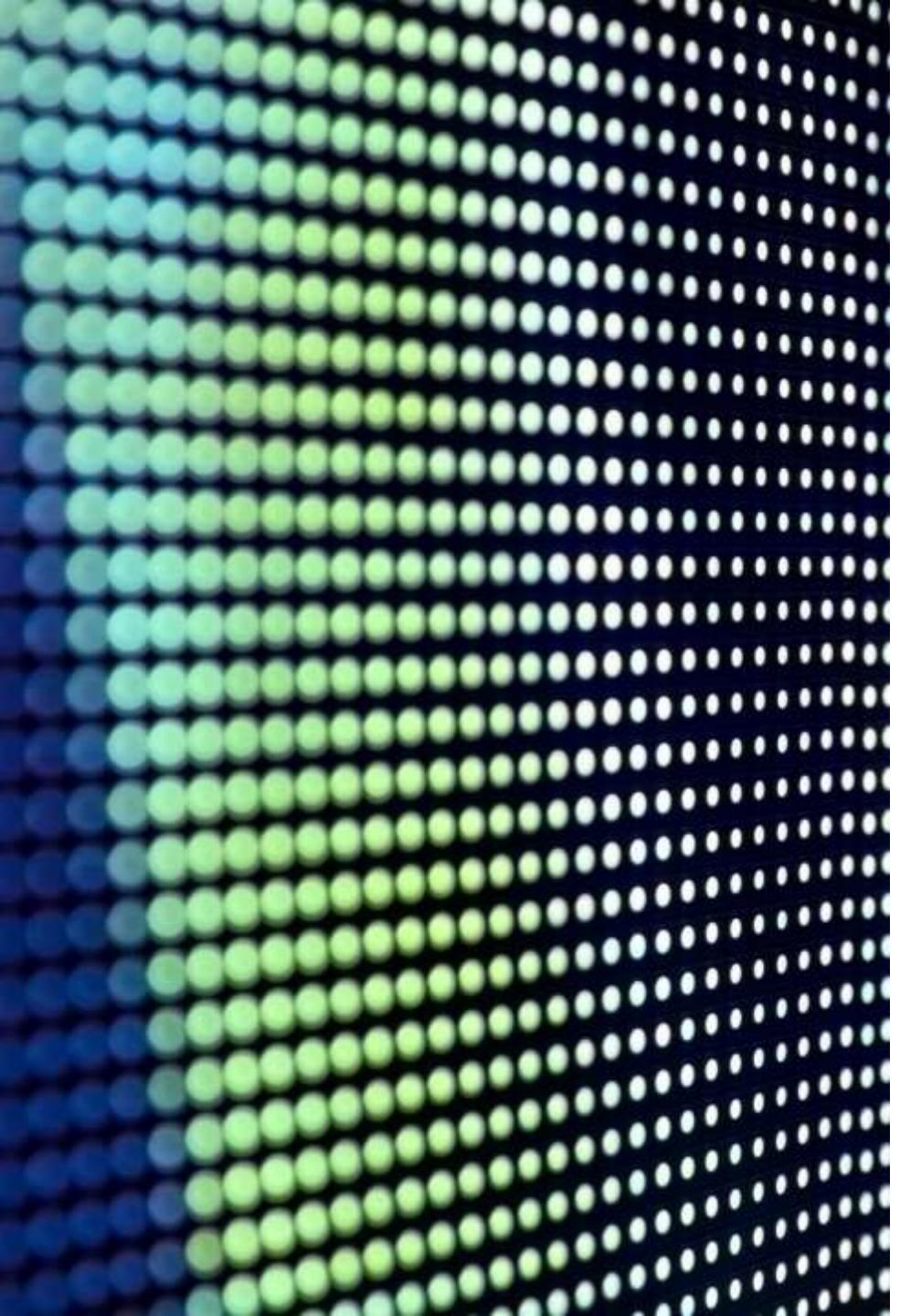
Modelo de la cascada





“Dejemos de desperdiciar \$78 mil millones de dólares al año”

- “las empresas estadounidenses gastan miles de millones de dólares en software que no hace lo que se supone que debe hacer”
- [CIO Magazine: Let's Stop Wasting \\$78 Billion A Year | Linux Today](#)



Código defectuoso

Responsable del hasta el 45% del tiempo que están fuera los sistemas

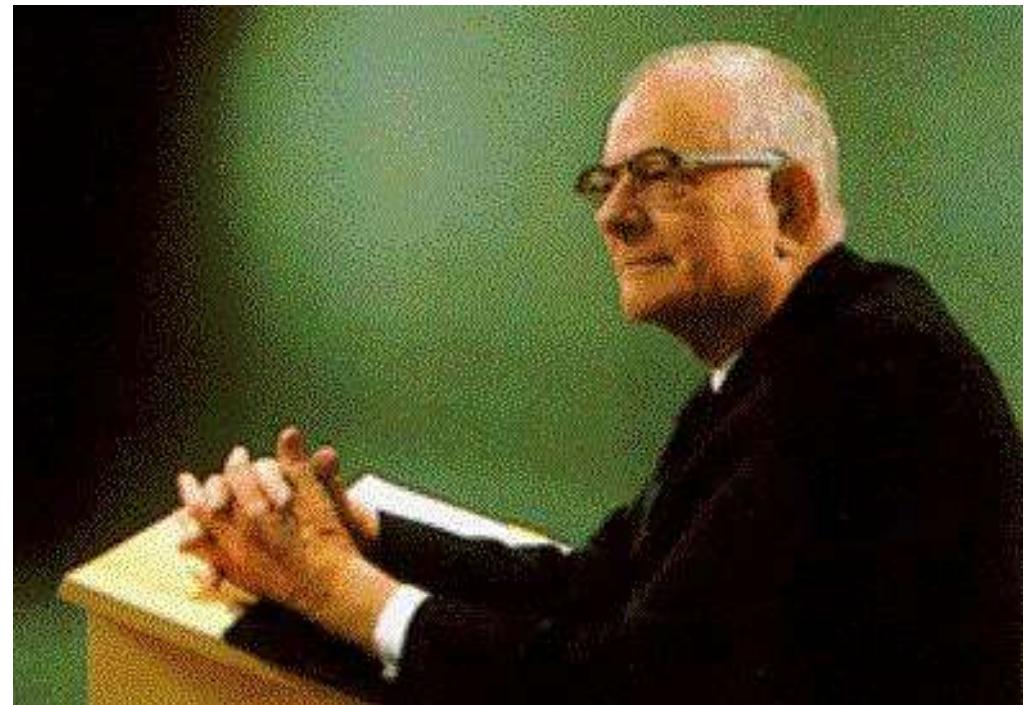
Se requiere de 3 a 4 defectos por 1,000 líneas de código para que un programa tenga mal desempeño

Los programadores cometen 1 error cada 10 líneas

InformationWeek

Dr. W. Edwards Deming

- Padre de la evolución de la calidad
- Cuatro principios:
 - Quality improvement drives the entire economy
 - The customer always comes first
 - Do not blame the person, fix the system
 - Plan-Do-Check-Act



Quality improvement drives the entire economy

- “Organizations that focused on improving quality would automatically reduce costs while those that focused on reducing cost would automatically reduce quality and actually increase costs as a result”





The
customer
always
comes first

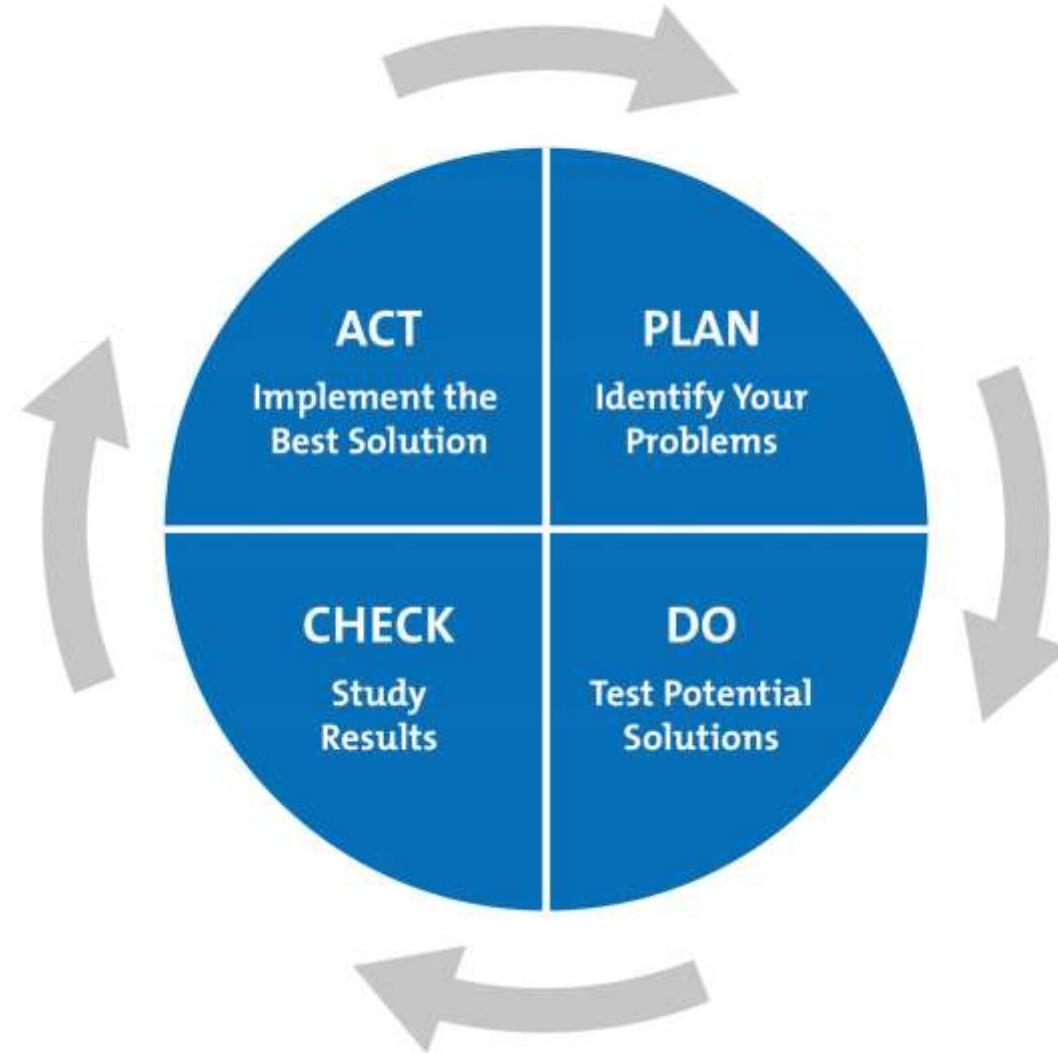
Do Not Blame The Person, Fix The System

Employees are internal
customers



Figure 1: The Plan-Do-Check-Act Cycle

Plan-Do-
Check-Act



¿Qué es calidad?

New Registration

Choose User Id	T\$1Dw_5&	Enter User ID
Password	*****	Enter Password
Confirm Password	hello123	
Name	Tester1.1 and Tester 1.2	
Email	(Requires verification. Will not be published.)	
Country	India	<input type="button" value="▼"/>
 Please enter the verification number exactly as shown in left.		
<input type="text" value="4"/>		<input type="button" value="Register"/>

calidad¹

Del lat. *qualitas*, *-ātis*, y este calco del gr. ποιότης *poiótēs*.

1. f. Propiedad o conjunto de propiedades inherentes a algo, que permiten juzgar su valor. *Esta tela es de buena calidad.*
2. f. Buena **calidad**, superioridad o excelencia. *La calidad de ese aceite ha conquistado los mercados.*
3. f. Adecuación de un producto o servicio a las características especificadas. *Control de la calidad de un producto.*
4. f. Carácter, genio, índole.
5. f. Condición o requisito que se pone en un contrato.
6. f. Estado de una persona, naturaleza, edad y demás circunstancias y condiciones que se requieren para un cargo o dignidad.
7. f. Nobleza del linaje.
8. f. Importancia o gravedad de algo.
9. f. pl. Prendas personales.
10. f. pl. Condiciones que se ponen en algunos juegos de naipes.

El Zen y el Arte del mantenimiento de la motocicleta

Calidad... sabes lo que es, pero no sabes lo que es. Pero eso es una contradicción. Algunas cosas son mejores que otras; es decir, tienen más calidad. Pero cuando tratas de decir lo que es la calidad, además de las cosas que la tienen, todo se desvanece... No hay nada de qué hablar. Pero si no puede decirse qué es Calidad, ¿cómo saber lo que es, o incluso saber que existe? Si nadie sabe lo que es, entonces, para todos los propósitos prácticos, no existe en absoluto. Pero para todos los propósitos prácticos, en realidad sí existe. ¿En qué otra cosa se basan las calificaciones? ¿Por qué paga fortunas la gente por algunos artículos y tira otros a la basura? Es obvio que algunas cosas son mejores que otras... pero, ¿en qué son mejores? Y así damos vueltas y más vueltas, ruedas de metal que patinan sin nada en lo que hagan tracción. ¿Qué demonios es la Calidad? ¿Qué es?



David Garvin



La calidad es un concepto complejo y de facetas múltiples



Transcendental: Se reconoce pero no se puede definir



Usuario: Concibe la calidad en términos de las metas especificadas por el usuario final



Fabricante: Especificaciones originales del producto



Producto: Características inherentes

Valor: Lo que el cliente este dispuesto a pagar por él

Diseño y conformidad

Calidad de diseño: Grado en la que el diseño cumple las funciones y características especificadas en los requerimientos

Calidad de la conformidad: Grado en la que la implementación se apega al diseño y el sistema resultante cumple sus metas

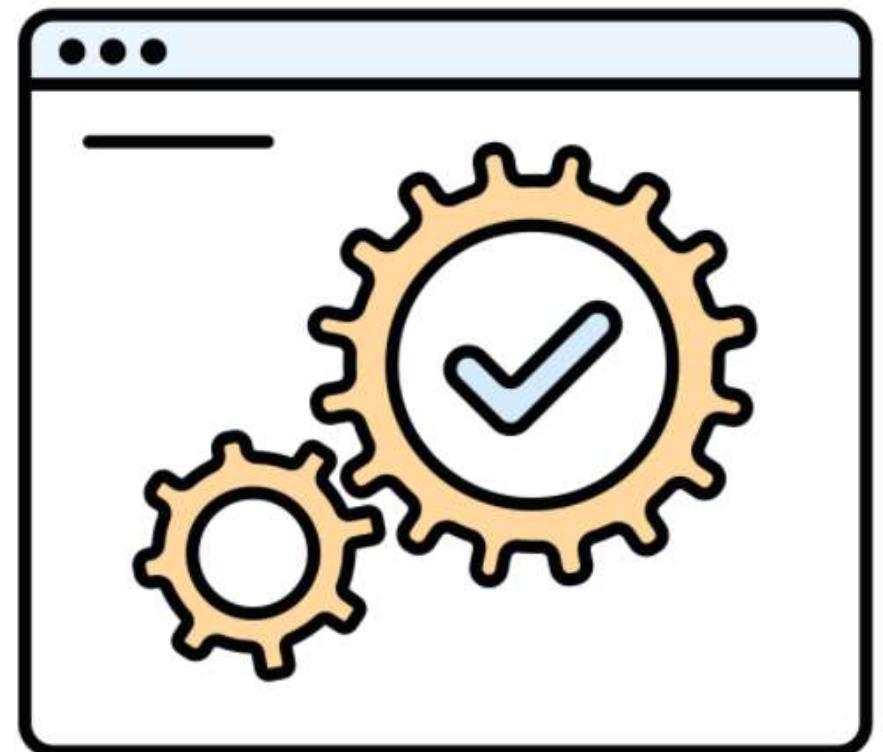
¿Es todo?

- Robert Glass: “si el usuario no está satisfecho, nada de lo demás importa.”

satisfacción del usuario = producto que funciona + buena calidad + entrega dentro del presupuesto y plazo

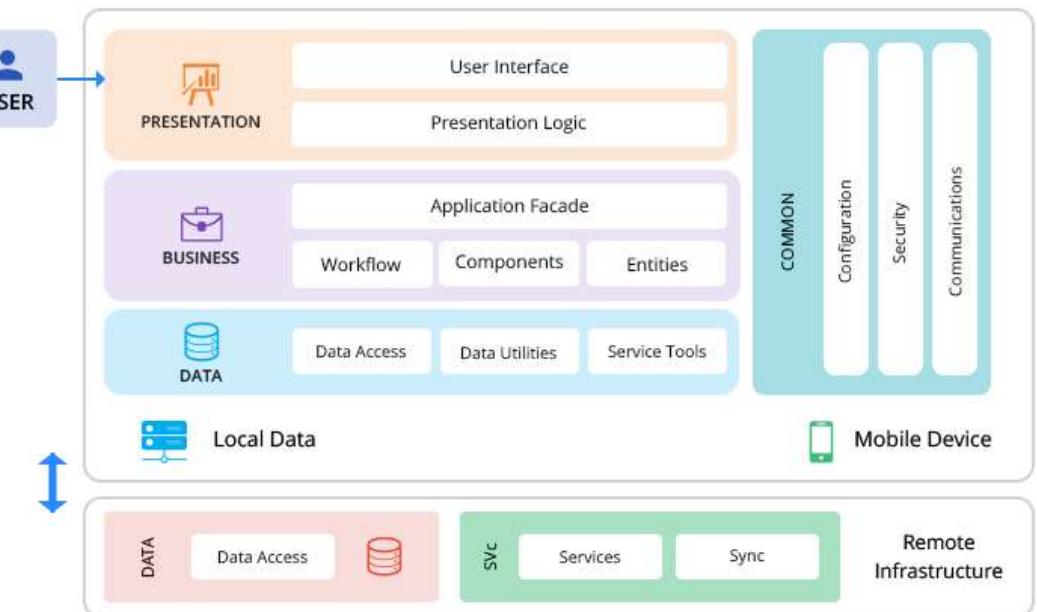
Calidad de Software

- “Proceso eficaz de software que se aplica de manera que crea un producto útil que proporciona valor medible a quienes lo producen y a quienes lo utilizan”



Proceso eficaz de software

- Establece infraestructura
- Analizar y diseñar
- Actividades sombrilla: Cambio y revisiones técnicas



Producto Útil

- Lo que el usuario final desea



Agrega
valor

Mayores utilidades por el
producto del Software

Mayor rentabilidad cuando
apoya un proceso de negocio

Mayor disponibilidad de
información

Dimensiones de calidad de Garvin

Desempeño:
Valor al usuario
final

Características:
Primera vista

Confiabilidad:
Disponibilidad

Conformidad:
Estándares

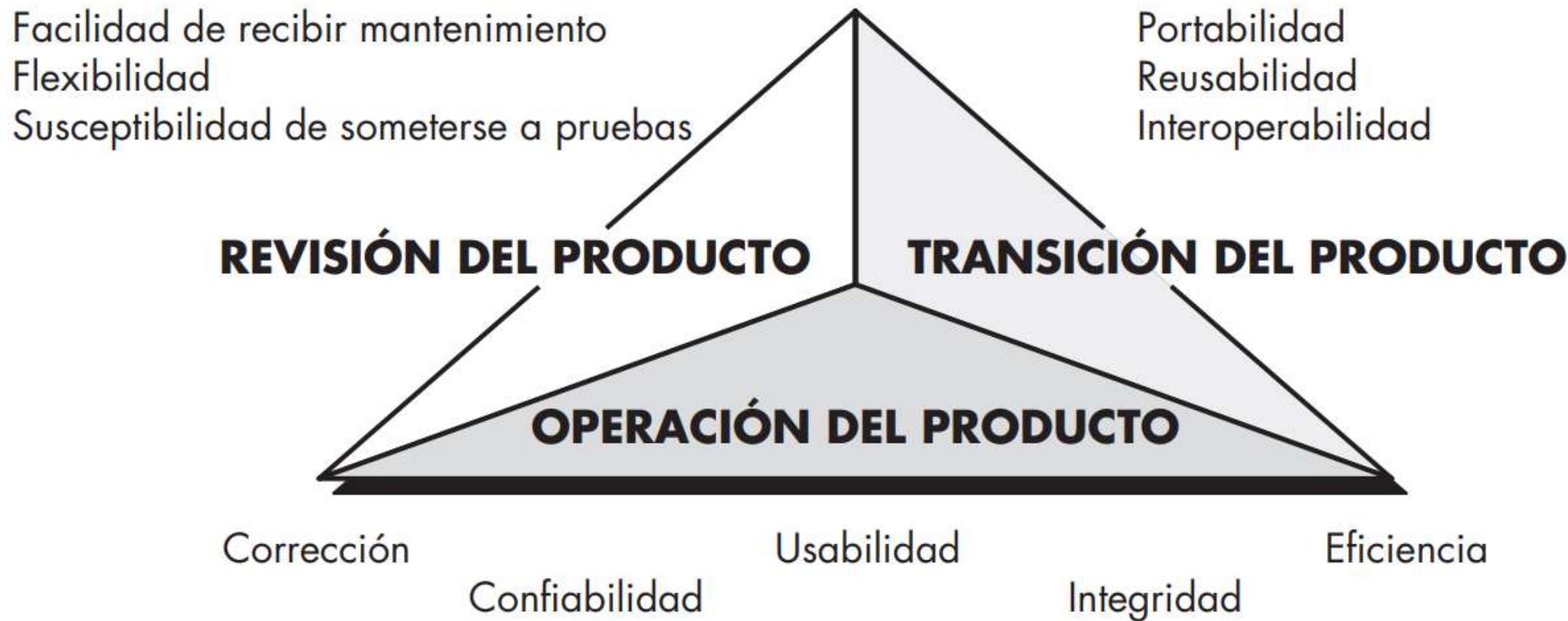
Durabilidad:
Mantenimiento

Servicios:
Rapidez de
mantenimiento

Estética:
Elegancia

Percepción:
Reputación

Calidad de McCall



OPERACIÓN DEL PRODUCTO

Corrección

Usabilidad

Eficiencia

Confiabilidad

Integridad

- Corrección: Cumple con los objetivos del cliente
- Confiabilidad: Cumple con su función en una precisión deseable
- Usabilidad: Esfuerzo para usar el sistema
- Integridad: Datos y usuarios
- Eficiencia: Recursos utilizados

Facilidad de recibir mantenimiento

Flexibilidad

Susceptibilidad de someterse a pruebas

REVISIÓN DEL PRODUCTO

- Facilidad de recibir mantenimiento: Detectar -> Corregir
- Flexibilidad: Modificar un programa en producción
- Susceptibilidad de someterse a pruebas: Esfuerzo para probar un sistema

- Portabilidad: Usar otro sistema
- Reusabilidad: Modular
- Interoperabilidad: Acoplar un sistema con otro



ISO 9126

<i>Características</i>	<i>Pregunta central</i>
<i>Funcionalidad</i>	¿Las funciones y propiedades satisfacen las necesidades explícitas e implícitas; esto es, el qué . . . ?
<i>Confiabilidad</i>	¿Puede mantener el nivel de rendimiento, bajo ciertas condiciones y por cierto tiempo?
<i>Usabilidad</i>	¿El software es fácil de usar y de aprender?
<i>Eficiencia</i>	¿Es rápido y minimalista en cuanto al uso de recursos?
<i>Mantenibilidad</i>	¿Es fácil de modificar y verificar?
<i>Portatilidad</i>	¿Es fácil de transferir de un ambiente a otro?

Tabla 1. Características de ISO-9126 y aspecto que atiende cada una.

Funcionalidad

Satisface las necesidades por la cual fue diseñado

- Adecuación
- Exactitud
- Interoperabilidad
- Conformidad
- Seguridad

Funcionalidad

- El software presenta resultados acordes a la necesidad a la que fue diseñado
- El software cuenta con funciones apropiadas para efectuar las tareas que fueron especificadas
- El software previene el acceso no autorizado
- El software sigue estándares de la industria
- El software tiene la habilidad de interactuar con otros sistemas

Adecuación

Exactitud

Interoperabilidad

Conformidad

Seguridad

Funcionalidad

- El software presenta resultados acordes a la necesidad a la que fue diseñado
 - El software cuenta con funciones apropiadas para efectuar las tareas que fueron especificadas
 - El software previene el acceso no autorizado
 - El software sigue estándares de la industria
 - El software tiene la habilidad de interactuar con otros sistemas
-
- The diagram illustrates the relationship between functional requirements and quality attributes. It features five orange arrows originating from the right side of the slide and pointing towards the left, where the requirements are listed. The requirements are aligned with the following quality attributes:
 - A adecuación (Fit) - aligned with the first requirement.
 - E exactitud (Accuracy) - aligned with the second requirement.
 - I interoperabilidad (Interoperability) - aligned with the fourth requirement.
 - C conformidad (Conformity) - aligned with the fifth requirement.
 - S seguridad (Security) - aligned with the third requirement.

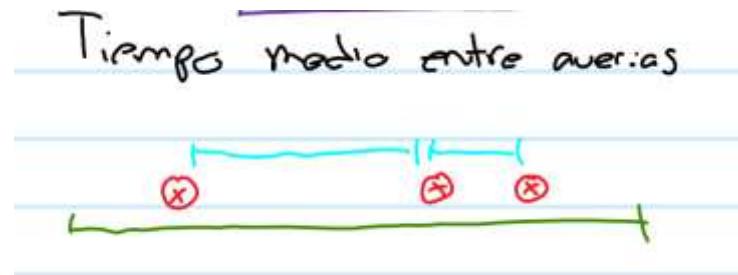
Confiabilidad

Capacidad del software de mantener su nivel de ejecución bajo condiciones normales en un periodo de tiempo establecido

- Nivel de madurez
- Tolerancia a fallos
- Recuperación

Confiabilidad

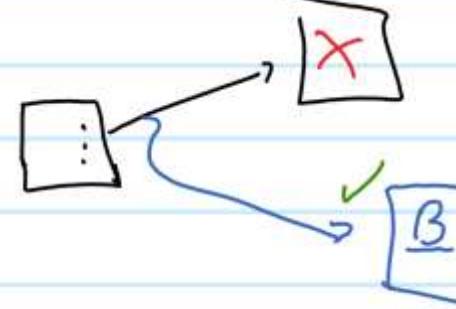
Nivel de madurez
Tolerancia a fallos
Recuperación



Cantidad de tiempo fuera de servicio



Chaos testing

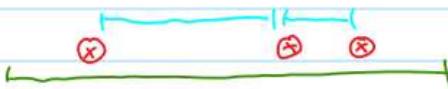


Tiempo medio hasta la recuperación

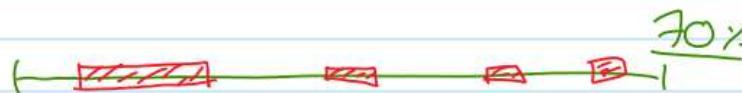


Confiabilidad

Madurez
Tiempo medio entre averías

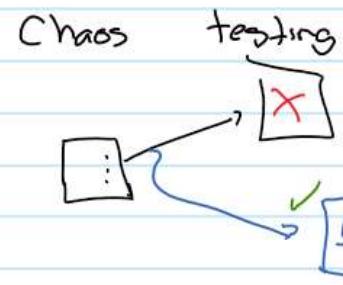


Disponibilidad
Cantidad de tiempo fuera de servicio



Area de operación

Tolerancia a fallos



Capacidad de recuperación
Tiempo medio hasta la recuperación



Usabilidad

Esfuerzo necesario que deberá invertir el usuario para utilizar el sistema

- Comprensibilidad
- Facilidad de aprender
- Operabilidad

Usabilidad

- Un usuario requiere de una sesión de 3 horas para entender lo que hace una aplicación
- Una aplicación fue desarrollada siguiendo los mismos flujos que otras aplicaciones en el mercado usan
- Métricas de conversión y pasos hasta lograr un objetivo son ejemplos de:

Comprendibilidad
Facilidad de aprender
Operabilidad

Eficiencia

Relación entre el nivel de funcionamiento del software y la cantidad de recursos usados

- Tiempo
- Recursos

Mantenibilidad

Esfuerzo necesario para realizar modificaciones al software

- Capacidad de análisis
- Modificación
- Estabilidad
- Facilidad de prueba

Mantenibilidad

- Un cambio de requerimientos produce un retrabajo en el código y en el modulo de pruebas
- Un cambio en el modulo C produce un cambio inesperado en la base de datos
- Un nuevo requerimiento del cliente produce un retrabajo en las funciones B y D
- Un nuevo requerimiento pide que además de estar disponible en Android, lo este también para iOS

Capacidad de análisis, Modificación, Estabilidad, Facilidad de prueba

Portabilidad

Habilidad del software de ser transferido de un ambiente a otro

- Adaptabilidad
- Facilidad de instalación
- Conformidad
- Capacidad de reemplazo

Portabilidad

- Un editor de videos guarda los videos en un formato único llamado “.vid7”
- Una aplicación sigue estándares de portabilidad
- Un instalador es un ejemplo de
- Un cambio en el requerimiento obliga a que el sistema sea transferido a un nuevo servidor

Adaptabilidad

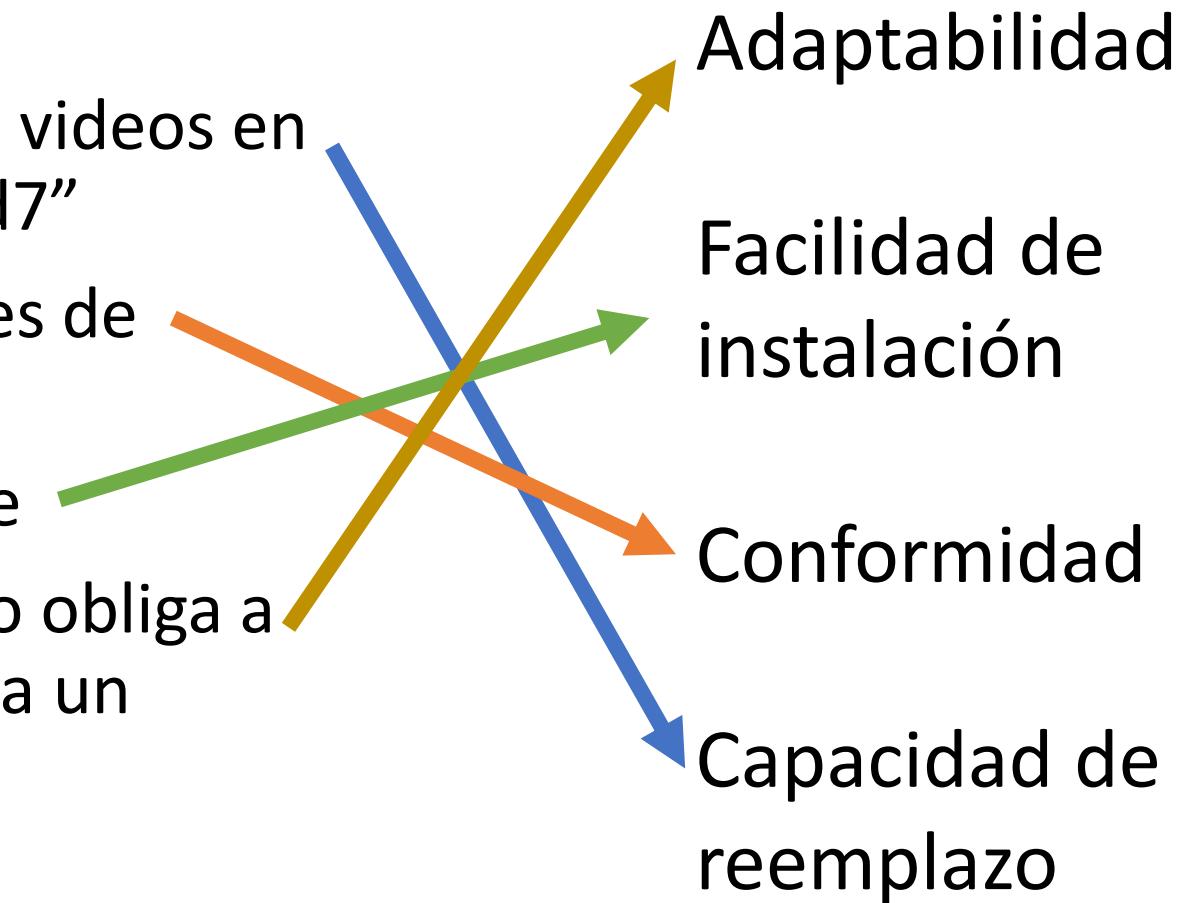
Facilidad de instalación

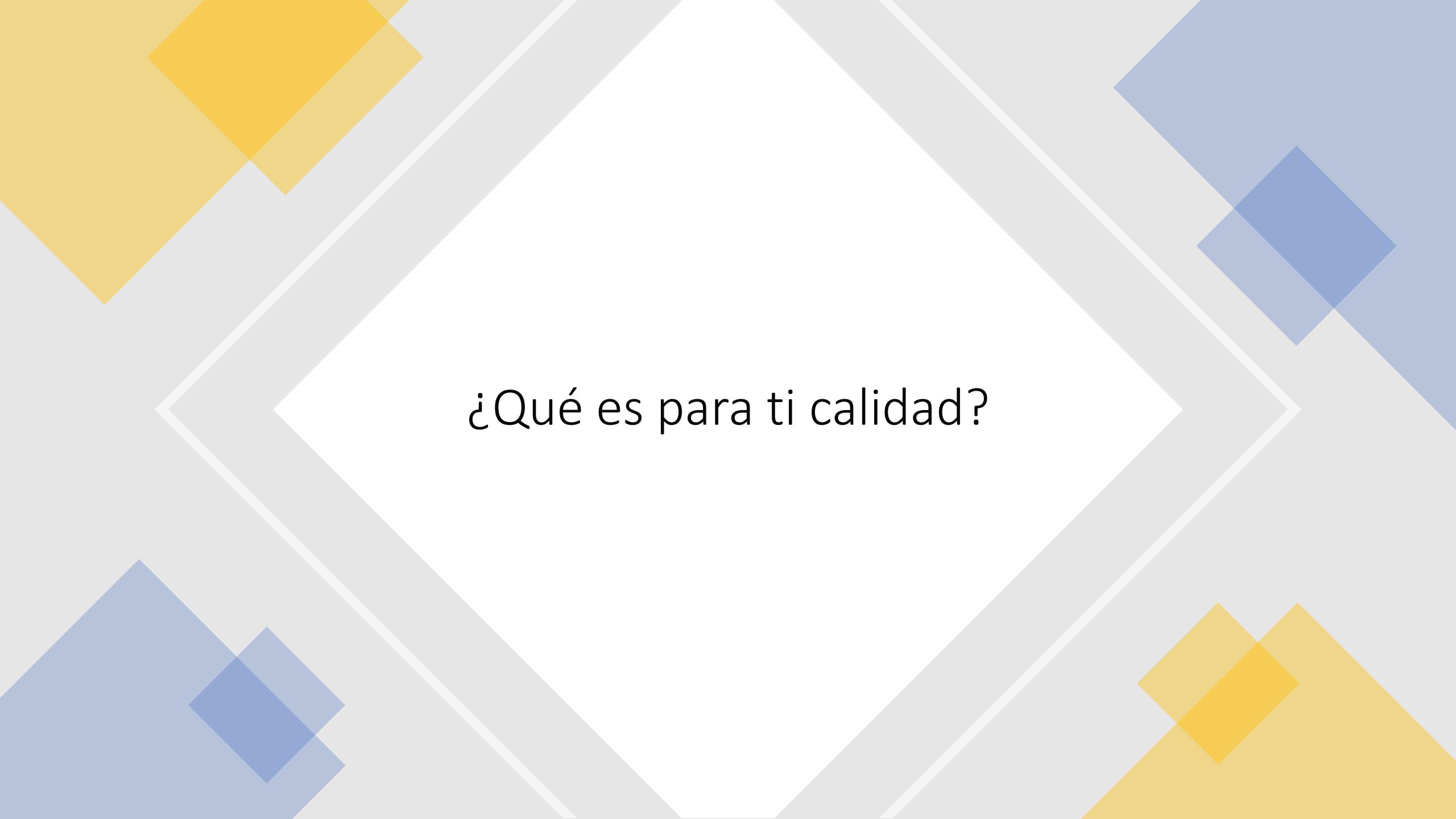
Conformidad

Capacidad de reemplazo

Portabilidad

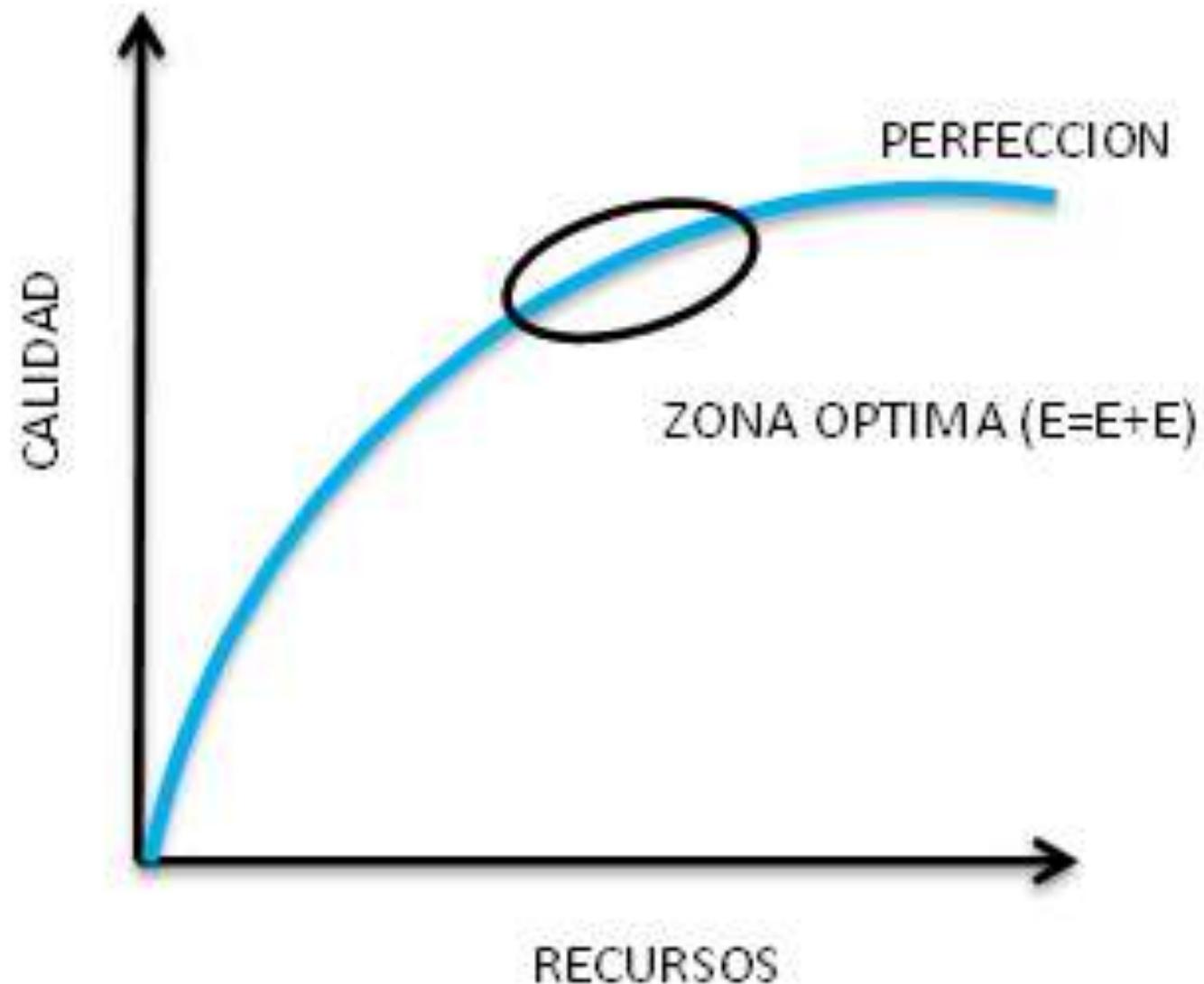
- Un editor de videos guarda los videos en un formato único llamado “.vid7”
- Una aplicación sigue estándares de portabilidad
- Un instalador es un ejemplo de
- Un cambio en el requerimiento obliga a que el sistema sea transferido a un nuevo servidor





¿Qué es para ti calidad?

“Perfect is the enemy of good”



Costos



Prevención

Actividades de administración
Actividades técnicas:
Requerimientos y diseño
Planear las pruebas
Capacitación



Evaluación

Revisiones técnicas
Recabar datos
Hacer pruebas



Falla

Internos
Externos

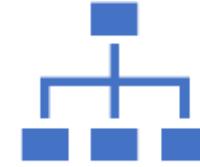
Costo de corregir errores



¿Cómo aseguramos la calidad?



Métodos de la ingeniería
de software



Técnicas de administración
de proyectos



Control de calidad



Aseguramiento de la
calidad

Mutation Testing

Universidad Autónoma de Coahuila

Facultad de Sistemas

Calidad y Pruebas de Software

Carlos Nassif Trejo García



Dick Lipton

1970s

Medir la calidad de
los casos de prueba

Mutantes

Vivo

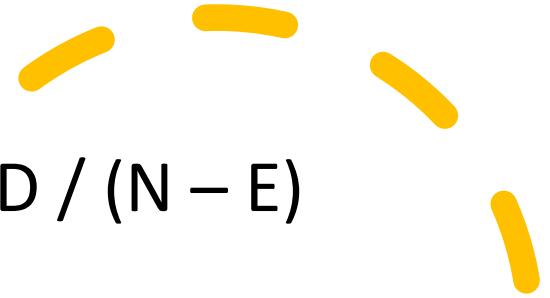
- Equivalente
- Matable

Muerto

Mutation adequate



- Mutation Score = $100 \times D / (N - E)$
 - D mutantes muertos
 - N Numero de mutantes totales
 - E mutantes equivalentes



Mutation Analysis

Un banco de pruebas es determinado para distinguir entre el programa original y sus mutantes

Nuevos casos de prueba son creados para matar a los “matables”

El proceso se repite hasta tener una mutation score deseable



Pasos

1. Programa P, casos de prueba T correctos
2. Correr cada T al programa P y asegurarse que pasen
3. Crear mutantes {Pi}
4. Ejecutar cada T a cada mutante Pi
 1. No pasa = Muerto
 2. Pasa = Equivalente o matable
5. Calcular el “mutation score”
 1. Mutation score bajo = Crear nuevos casos de prueba e ir al paso 2
 2. Score alto = Terminar

Suposiciones

- Hipótesis del programador competente: El programador no hace programas al azar
- Efecto de acoplamiento: Si un software tiene **fallas**, usualmente habrá un par de mutantes que solo podrán ser asesinados por casos de prueba que también detecte la **falla**





Pruebas de integración

Universidad Autónoma de Coahuila
Facultad de Sistemas

Carlos Nassif Trejo García

Sistema

- Colección de modulo interconectados con un único objetivo
- Subsistema: Sistema provisional que no esta integrado completamente

Importancia

- Errores de interfaz
- Limitaciones de las pruebas unitarias
- Identificar módulos con errores



Crear una versión funcionable del sistema

Juntar los módulos de forma
incremental

Asegurarse que cualquier
modulo adicional funcione sin
afectar las funcionalidades ya
implementadas

Interfaces

- Procedure call
- Shared Memory
- Message Passing



Put them
together
errors

Construcción

Funcionalidad
inadecuada

Ubicación de
la
funcionalidad

Cambios en la
funcionalidad

Funcionalidad
agregada

Uso incorrecto
de la interfaz

Mal entendido
de la interfaz

Alteración en
estructura de
datos

Manejo
incorrecto de
errores

Etc...

Ventajas



Defectos son detectados en una etapa temprana



Mas fácil de corregir



Feedback temprano sobre las métricas de los módulos individuales y el sistema en general



Agendar arreglos de defectos es flexible y puede llevarse junto con la etapa de desarrollo

Tipos de Integration Testing

Intrasistema

Intersistema

Por pares

Intrasystem Testing

Integración de bajo nivel

Combinar módulos para construir un sistema

Ejemplo: Cliente servidor

Intersystem Testing

Integración de alto nivel

Sistemas no dependiente de interfaces

Todos los sistemas se conectan y se hacen pruebas end-to-end

Asegurar interconexión del sistema

Solo se prueba una funcionalidad a la vez

Pairwise Testing

Integración de nivel medio

Solo un par de sistemas interconectados son probados

Probar que 2 sistemas en construcción funcionan como lo esperado

Técnicas de integración

No es necesario esperar a que todos los módulos estén termiandos

Incremental

Top Down

Bottom Up

Sandwich

Big Bang

Incremental

Ciclos incrementales creando builds

En cada ciclo se prueba, arreglan errores y se pasa al siguiente ciclo

Por capas

- Self-Contained: Contiene el código necesario
- Stable: El subsistema puede estar en servicio por 24hrs sin anomalías

Check-in request

Release note

Creando build

Tomar la ultima versión aprobada de los módulos

Compilar

Push al repo

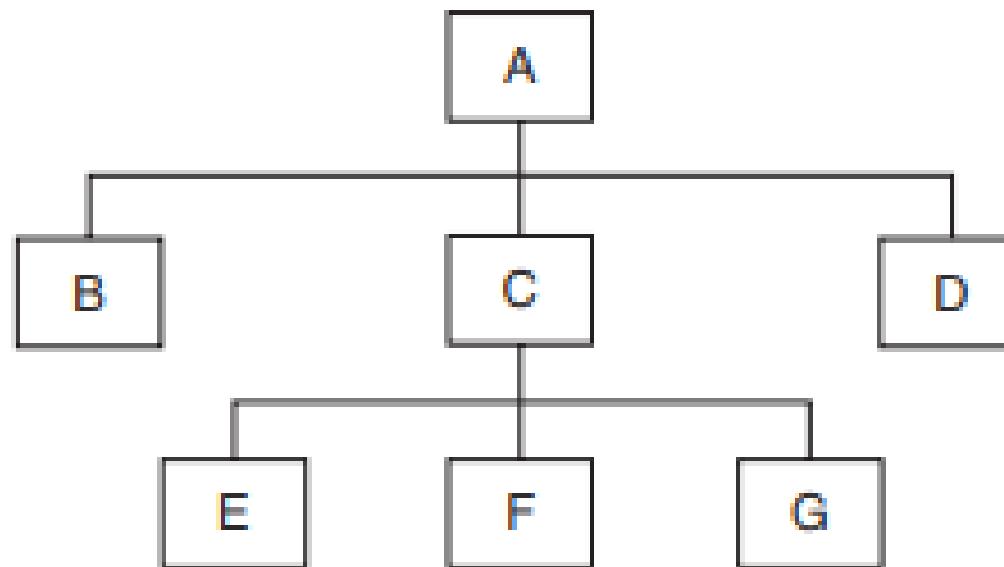
Enlazar los módulos a subbassemblies

Verificar que sean correctos

Control de versiones

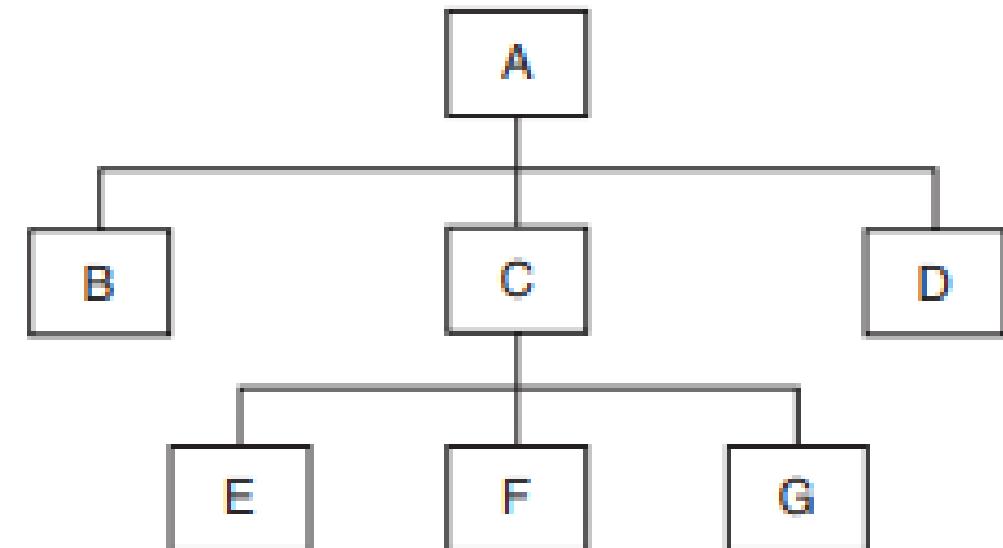
Top Down

- Sistema jerárquico: Módulos principales se descomponen en módulos secundarios



PASOS

- Tomar como base el modulo raíz con sus stubs
- Escoger un stub para remplazarlo con el modulo
- Hacer pruebas
- Repetir



Ventajas

Se puede observar las funciones del nivel de sistema en lo que la integración se va desarrollando

Encontrar problemas de interfaz se vuelve sencillo por la naturaleza de integración

Las pruebas desarrolladas en integración, se pueden reutilizar en pruebas de regresión

Es natural que las pruebas del nodo principal correspondan a funciones de sistema

Limitaciones

Normalmente no se puede apreciar funciones del sistema en una etapa temprana

Selección entre casos de prueba y diseño de stubs se vuelve compleja por la distancia entre el modulo principal y el stub

Bottom Up

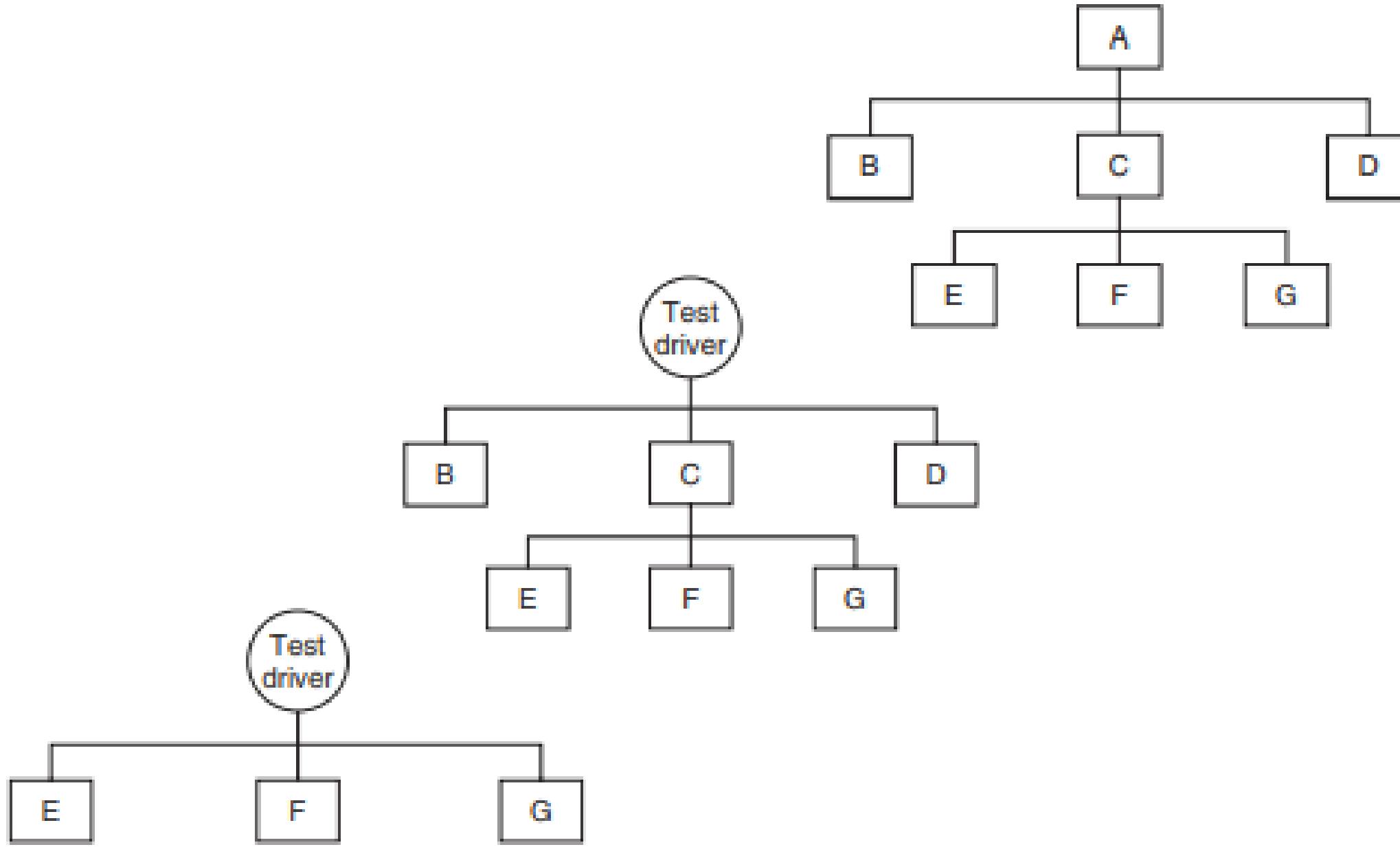
Empieza con módulos de bajo nivel

- No llaman a otro modulo

Construir un test driver que llame a esos módulos

Probar con el test driver

Si las pruebas son satisfactorias, remplazarlo con el modulo real



Ventajas

- Si los módulos de bajo nivel normalmente son llamadas por otros módulos, tiene mas sentido integrar de esta manera

Desventajas

No se pueden observar funcionalidades del sistemas hasta que se integre módulos de alto nivel

Los errores suelen ocurrir en módulos de alto nivel, por lo que su detección es mas lenta

Comparación

Validación en decisiones de diseño:
Top-Down

Observar funciones de sistema: Top-
Down

Dificultad en diseñar casos de
prueba: Bottom-Up

Reusar casos de prueba: Top-Down

Sandwich

- Capas:
 - Top Layer: Top-Down
 - Middle Layer: Big Bang
 - Bottom Layer: Bottom Up



Big Bang

Pruebas los módulos de forma individual

Juntas todos los módulos para construir el sistema

Pruebas!



Técnicas de revisión

Universidad Autónoma de Coahuila

Facultad de Sistemas

Calidad y Pruebas de Software

Pruebas

Es el proceso de evaluar un producto, aprendiendo a través de la exploración y experimentación, lo cual incluye:

- Cuestionar
- Estudiar
- Modelar
- Observar
- Inferir
- Checar salidas de datos

Principios de la prueba

Glen Myers:

- La prueba es un proceso que ejecuta un programa con objeto de encontrar un error
- Un buen caso de prueba es el que tiene alta probabilidad de encontrar un error que no se ha detectado hasta el momento
- Una prueba exitosa es la que descubre un error no detectado hasta el momento

Principios de las pruebas

Davis

- Todas las pruebas deben poder rastrearse hasta los requerimientos del cliente.
- Las pruebas deben planearse mucho antes de que den comienzo
- El principio de Pareto se aplica a las pruebas de software
- Las pruebas deben comenzar “en lo pequeño” y avanzar hacia “lo grande”
- No son posibles las pruebas exhaustivas

Principios de las pruebas

Alan Page & Veren Genzen

1. Nuestra prioridad es mejorar el negocio
2. Nosotros aceleramos al equipo, usamos modelos como Lean Thinking y Teoria de las Restricciones para ayudar a identificar, priorizar y mitigar cuellos de botella en el sistema
3. Somos la fuerza para la mejora continua, ayudando al equipo a adaptarse y optimizar para tener éxito, en lugar de proporcionar una red de seguridad para detectar fallas
4. Nos preocupamos profundamente acerca de la cultura de calidad en el equipo, y asesoramos, lideramos y nutrimos el equipo para llevarlos a una cultura de calidad más madura
5. Nosotros creemos que el cliente es el único capaz de juzgar y evaluar la calidad de nuestro producto
6. Nosotros usamos datos de manera extensa y profunda para entender los casos de uso del cliente y entonces cerrar huecos entre hipótesis del producto e impacto del negocio
7. Expandimos las habilidades de testing y el conocimiento en todo el equipo; entendemos que esto reduce o elimina la necesidad de un especialista dedicado al testing

A vibrant, abstract background featuring a grid of binary digits (0s and 1s) in red and blue. Overlaid on this are several stylized charts and graphs, including a bar chart with teal bars and a line graph with a red line. The overall theme is digital data and technology.

Principios de validación

Terminando la codificación:

1. Realizar el recorrido de código
2. Ejecutar pruebas unitarias
3. Rediseñar el código

Failure, Error, Fault, and Defect

Anomalía: la manifestación de un error en el software.

Error: una acción **humana** que produce un resultado incorrecto.

Defecto: imperfección o deficiencia en un producto, el cual no cumple sus requerimientos o especificaciones y necesita ser reparado o remplazado.

Fallo: el cese de la habilidad de un producto de cumplir una función requerida o su incapacidad de funcionar dentro de márgenes previamente especificados.

Problema: dificultad o incertidumbre experimentada por una o más personas, como resultado de un encuentro insatisfactorio con el sistema usado.

"El error humano cometido inyecta un defecto en el software que, ocasionalmente, se observa como una anomalía a causa de un comportamiento incorrecto, no acorde a lo especificado, que finalmente provoca el fallo del sistema software"

Verificación y validación

Verificación: Asegurarse que en cada etapa se cumpla los requerimientos del cliente

Validación: Al final del proceso, comprobar con el cliente que todo esta bien

Estático y Dinámico

Estático:
Documentos

Dinámico:
Código

Revisión

En grupo

Resaltar las
mejoras

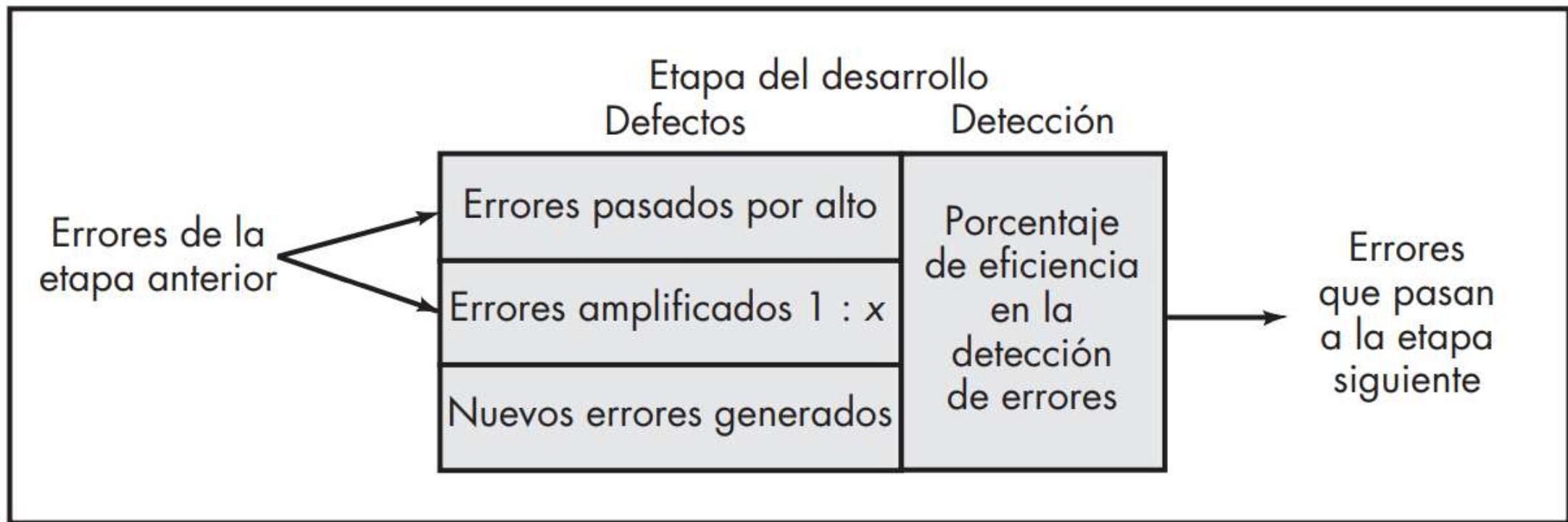
Confirmar partes
que no requieren
mejora

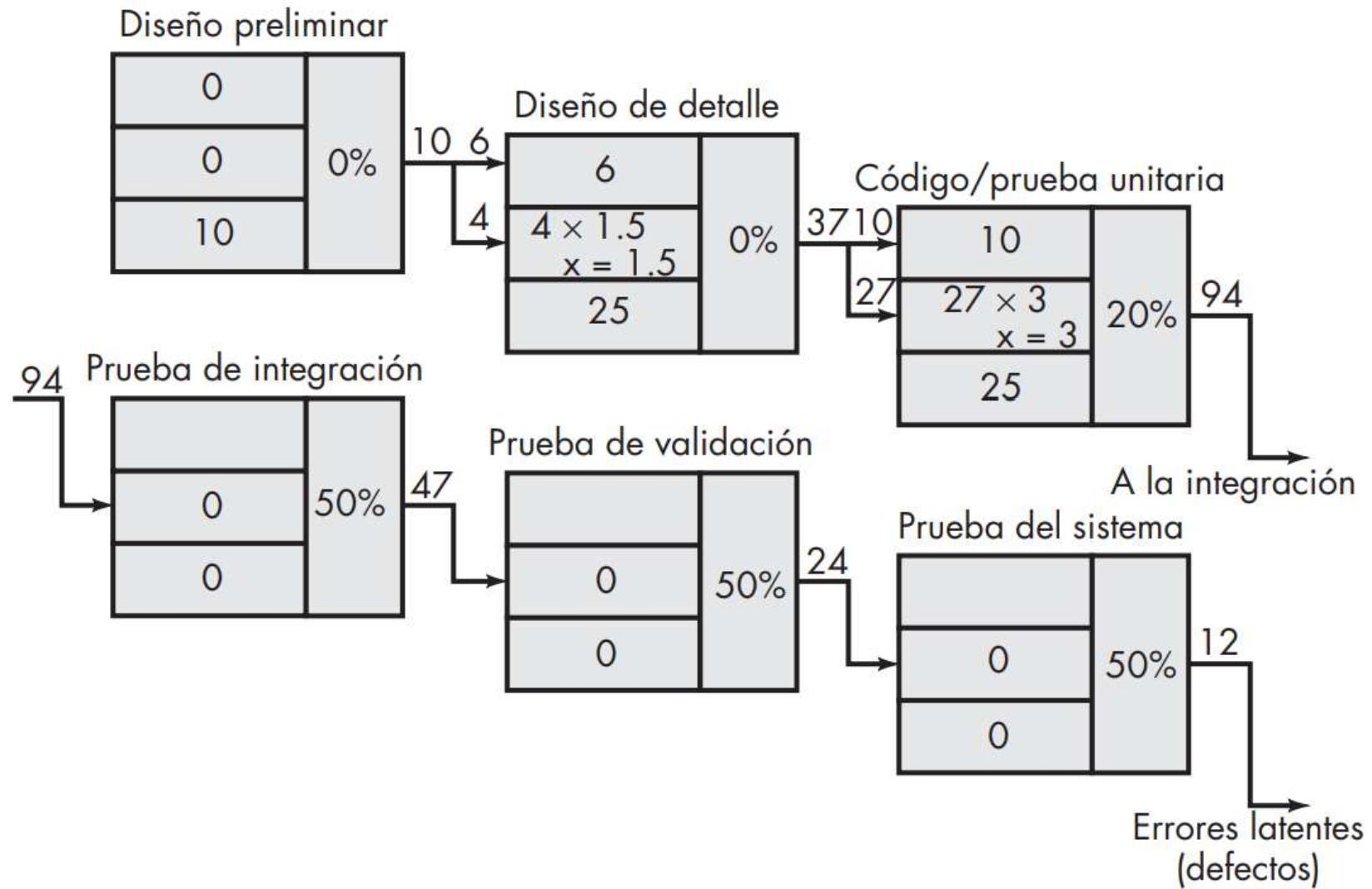
Trabajo técnico

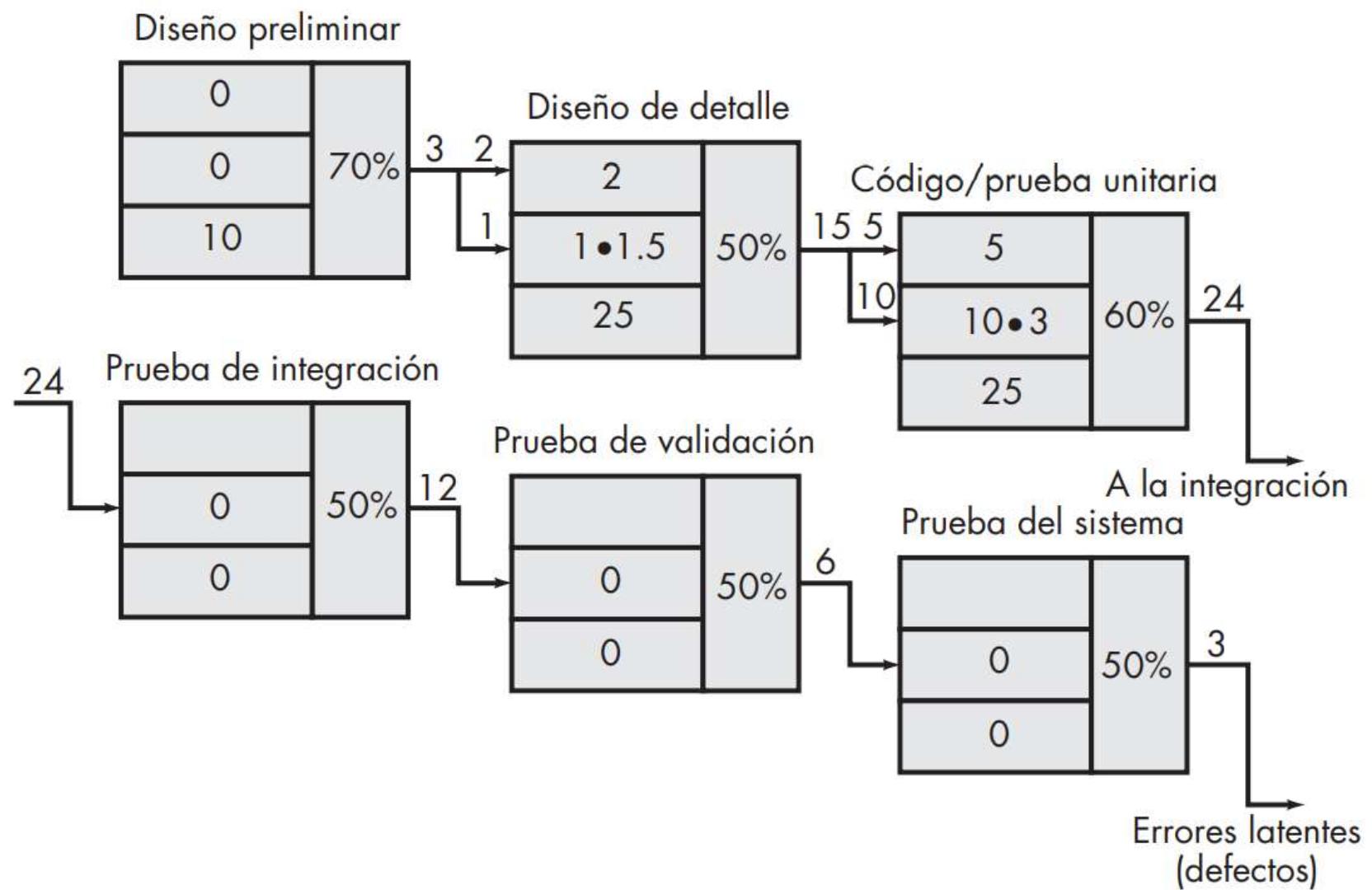
Revisión técnica

- Descubrimiento temprano de bugs de modo que no se propagan en las siguientes etapas
- Diseño: 50% - 65%
- Revisión: 75% eficaz

Modelo amplificación del defecto

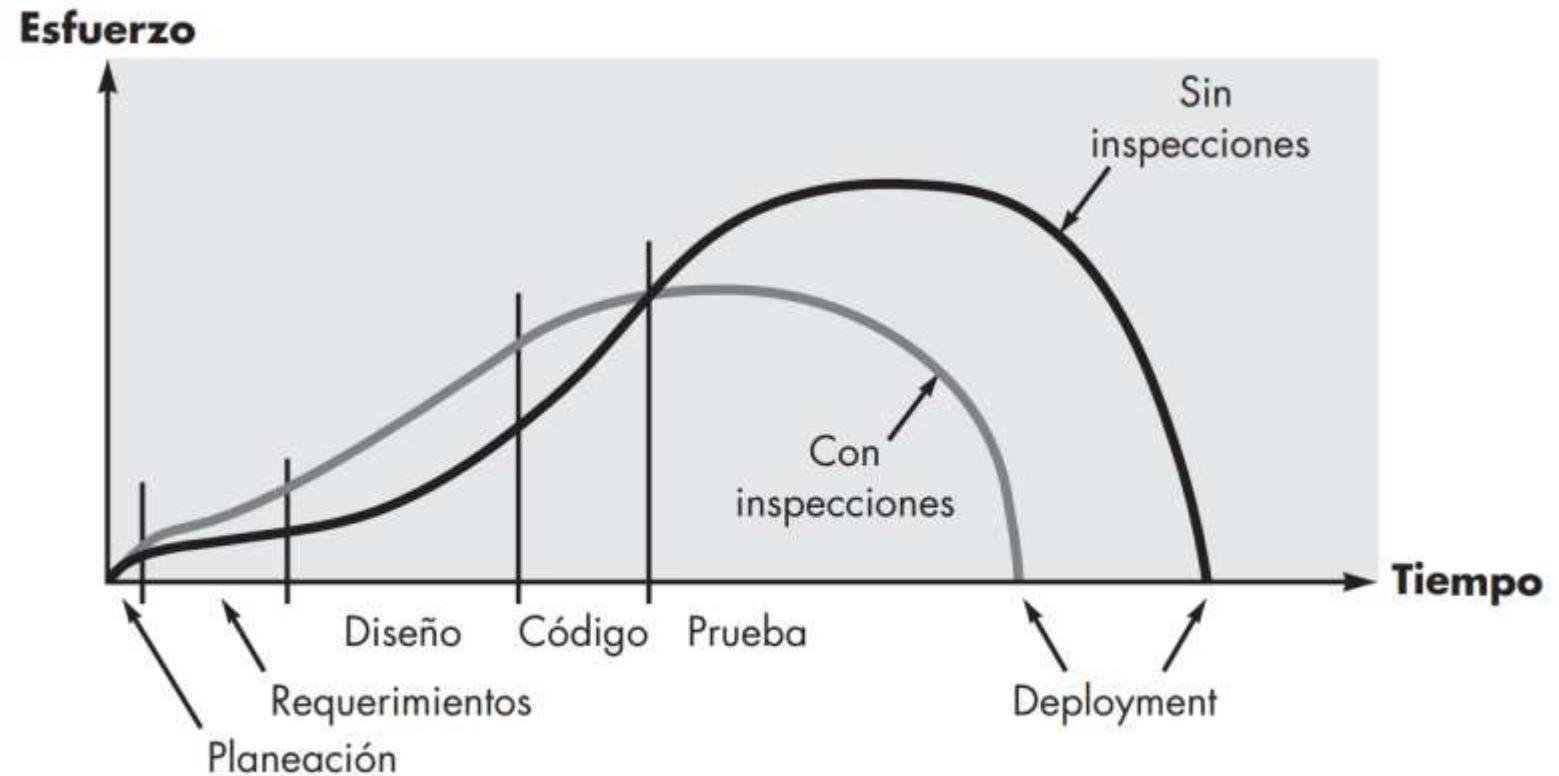






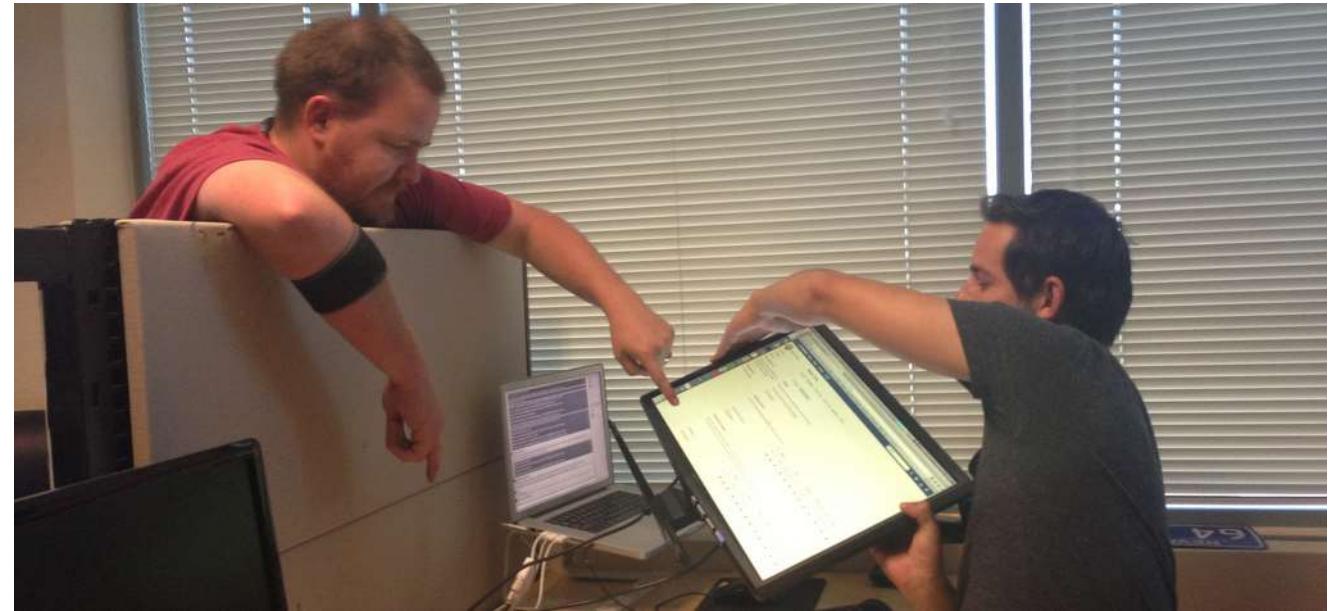
Nivel de formalidad

- Producto
- Plazo
- Personal



Revisión técnica informal

- Verificación de escritorio / Reunión Casual
- Listas de verificación*
- Material pequeño



Pair programming

- [Agile in Practice: Pair Programming - YouTube](#)

Revisión Técnica Formal

Descubrir errores en funcionamiento, lógica o implementación

Software cumple con requerimientos

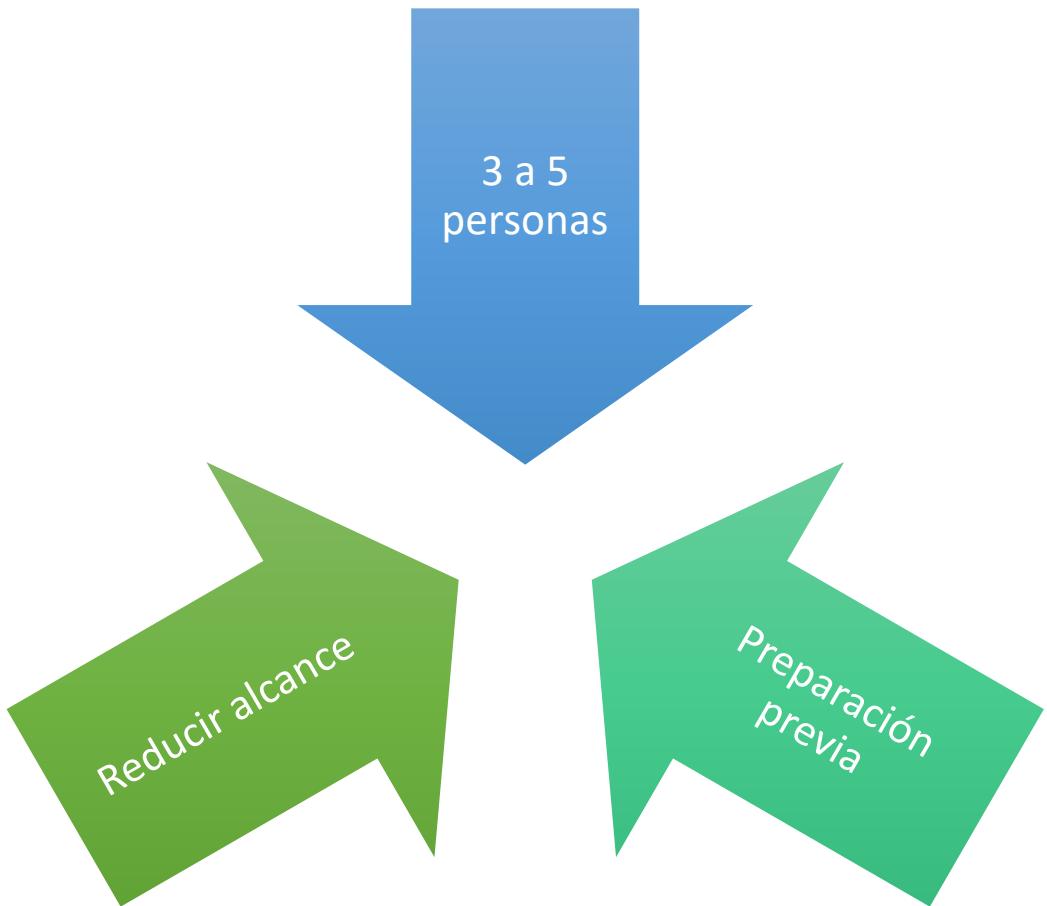
Software cumple con estándares

Software desarrollado de manera uniforme

Proyectos más manejables

Capacitación*

Guia



Reporte técnico formal de la revisión



¿QUÉ FUE LO QUE SE
REVISÓ?



¿QUIÉN LO REVISÓ?



¿DESCUBRIMIENTOS Y
CONCLUSIONES?

Lista de pendientes de la revisión

Identificar áreas de problemas en el producto

Guia para quien realice los correcciones

Lineamientos

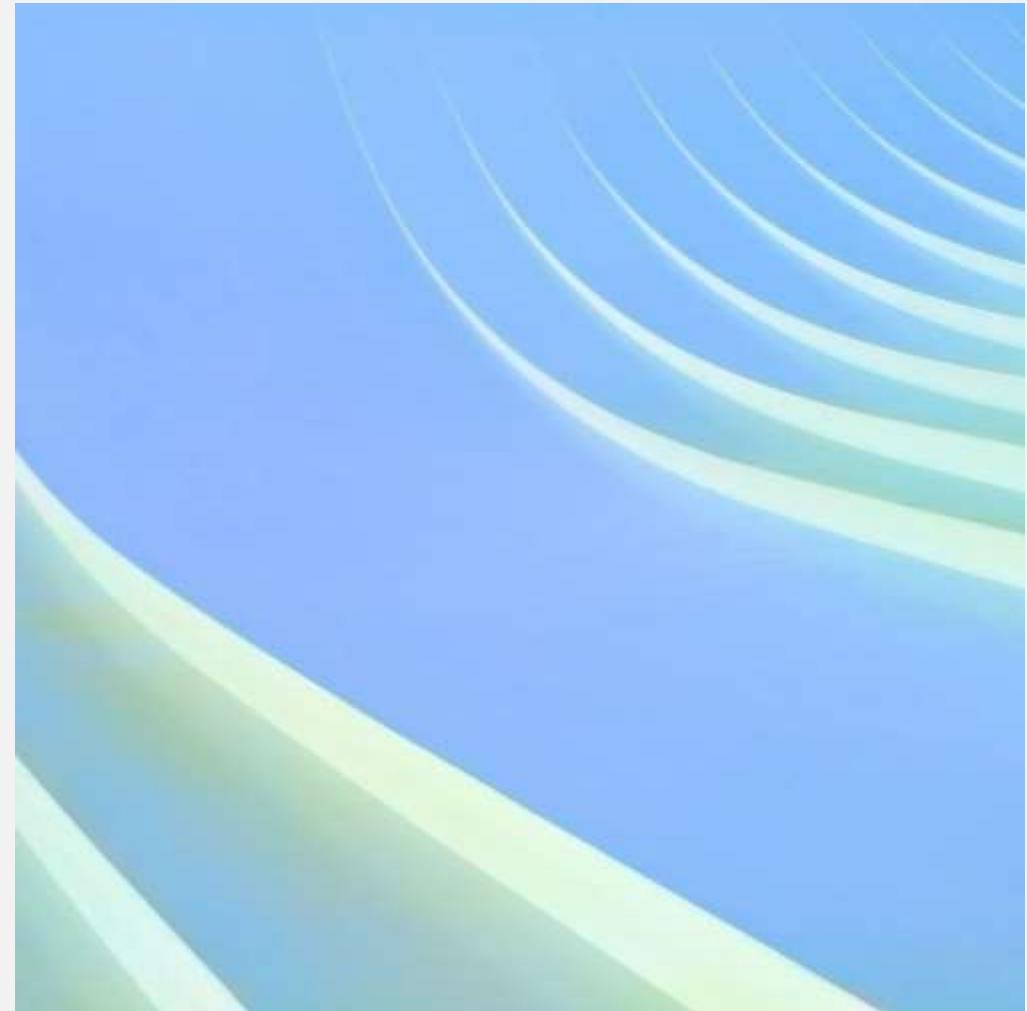
- Revise el producto, no el productor
- Establezca una agenda y sígala
- Limitar el debate y las contestaciones
- Enunciar áreas de problema, pero no intentar arreglar todos
- Notas por escrito
- Limitar numero de participantes
- Preparación previa
- Lista de verificación
- Asignar recursos y programar tiempo
- Capacitación a supervisores
- Revisar primeras versiones

Al terminar

Aceptar

Rechazar con errores graves:
Requiere de otra revisión formal

Rechazar con errores menores: No
requiere de otra revisión formal

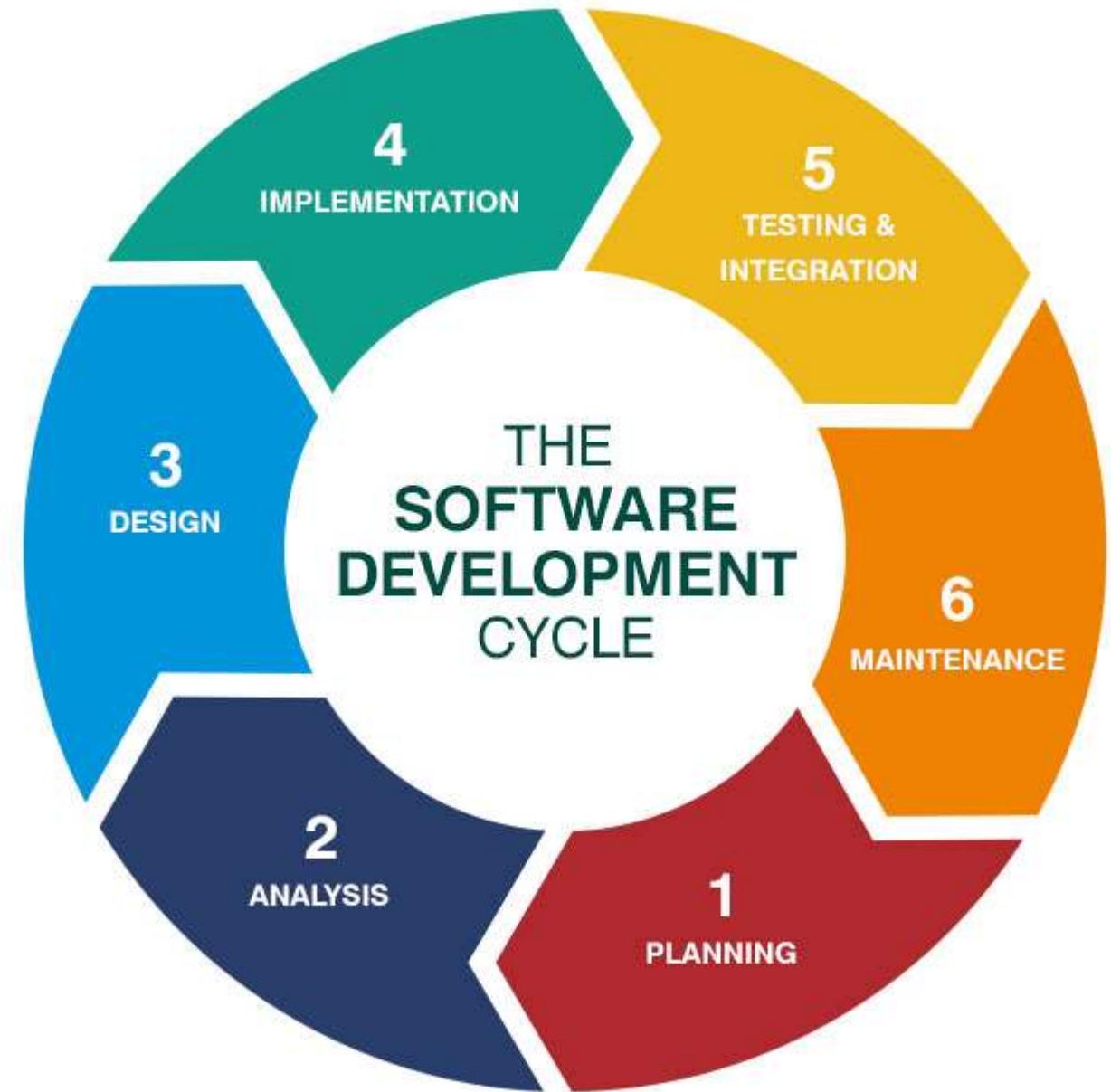


Revisión orientado al muestreo

- Determinar elementos susceptibles a errores



Recap ciclo de vida



Roles en las pruebas de software

QA Engineer

Test Manager

Test Engineer

Test Analyst

Test
Automation
Engineer

Security Tester

Data Science
Tester

Software
Development
Engineer in Test

QA Engineer

Producto y proceso

Monitorea cada fase

Asegura que el producto cumple con estandares

Test Manager

Equivalente al Project
manager

Posición gerencial
dentro del equipo de
QA

Test Engineer



MANUAL
TESTING



EXPLORATORY
TESTING



PERFORMANCE
TESTING

Test Analyst



Enfoca en problemas de negocio



Functional readiness



Diseña, Desarrolla, Ejecuta, y solución problemas en las pruebas

Test Automation Engineer

Procesos automáticos de prueba

Selenium

Cucumber

Puppeteer

UiPath

Security Tester



Proteger datos



Asegurar estándares

Data Science Tester



Datos



Analizar



Agrupar



Limpiar

Software Development Engineer in Test (SDET)



Desarrollador



Hace pruebas en todo el
proceso



Pruebas antes de liberar el
código

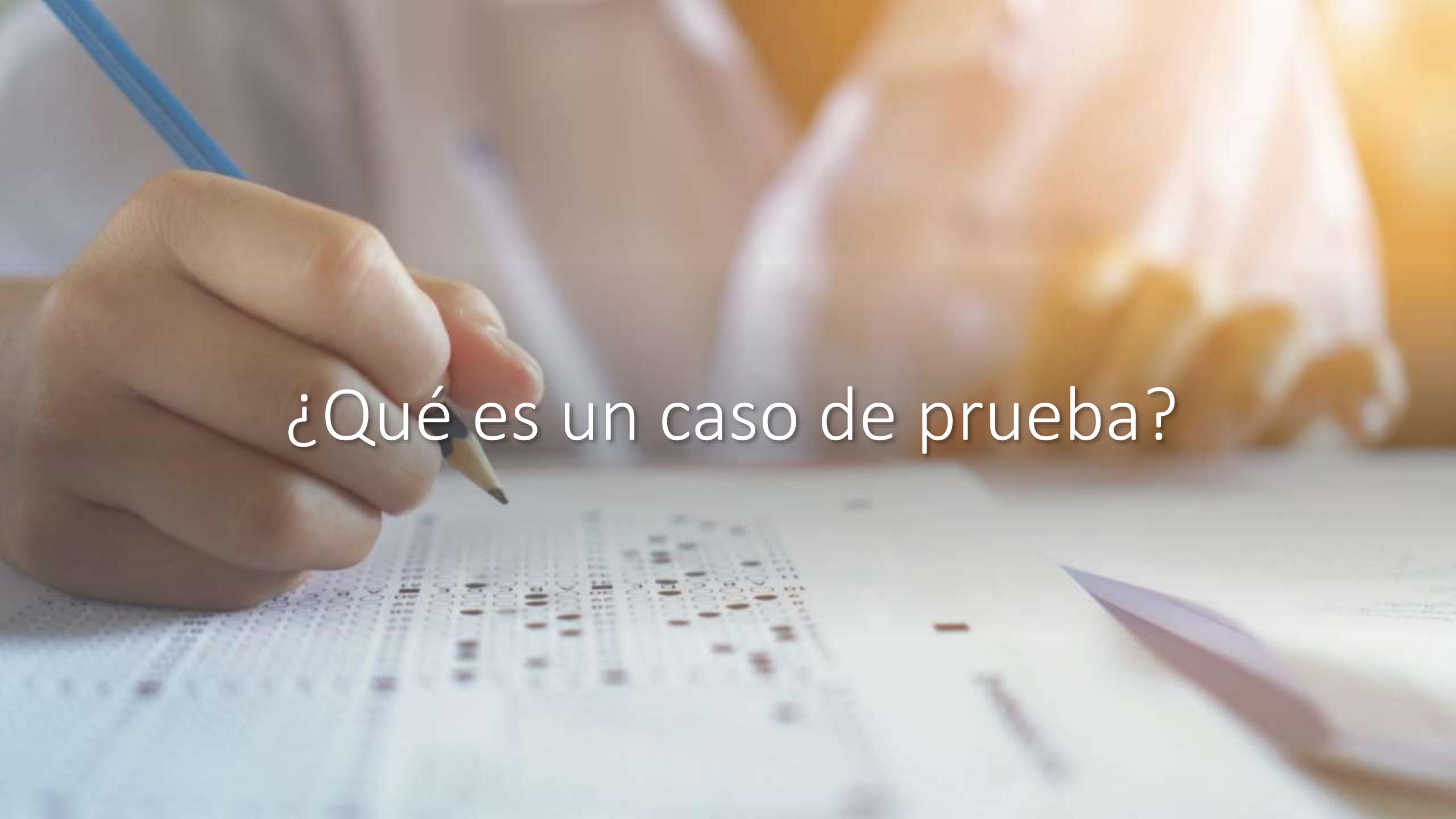
Testing

Universidad Autónoma de Coahuila

Facultad de Sistemas

Calidad y Pruebas de Software

Carlos Nassif Trejo García

A close-up photograph of a person's hand holding a blue pencil, poised to write on a blank answer sheet. The sheet features a grid of numbered columns, likely for a multiple-choice test. The background is blurred, showing other answer sheets and pencils.

¿Qué es un caso de prueba?

TB₁: <0, 0>,
TB₂: <25, 5>,
TB₃: <40, 6.3245553>,
TB₄: <100.5, 10.024968>.

Figure 1.3 Examples of basic test cases.

TS₁: <check balance, \$500.00>, <withdraw, “amount?”>,
<\$200.00, “\$200.00”>, <check balance, \$300.00> .

Figure 1.4 Example of a test case with a sequence of <input, expected outcome>.

Salidas esperadas



Valores



Cambios de estado



Secuencia de valores

Oráculo

- El que define la salida esperada

Complete Testing

- Dominio de las entradas
- Muy complejo
- Entornos de ejecución

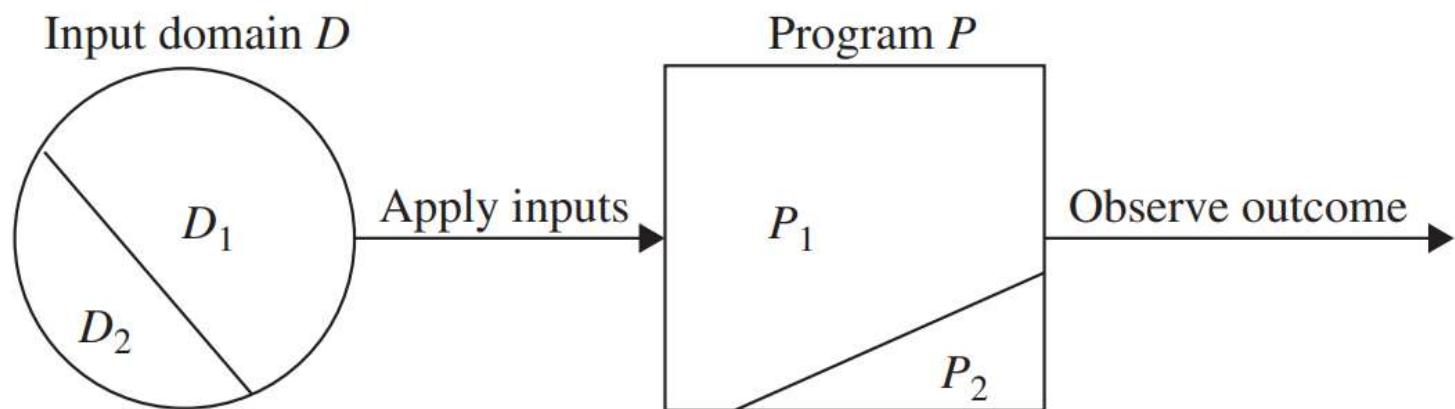


Figure 1.5 Subset of the input domain exercising a subset of the program behavior.

Actividades en el testing

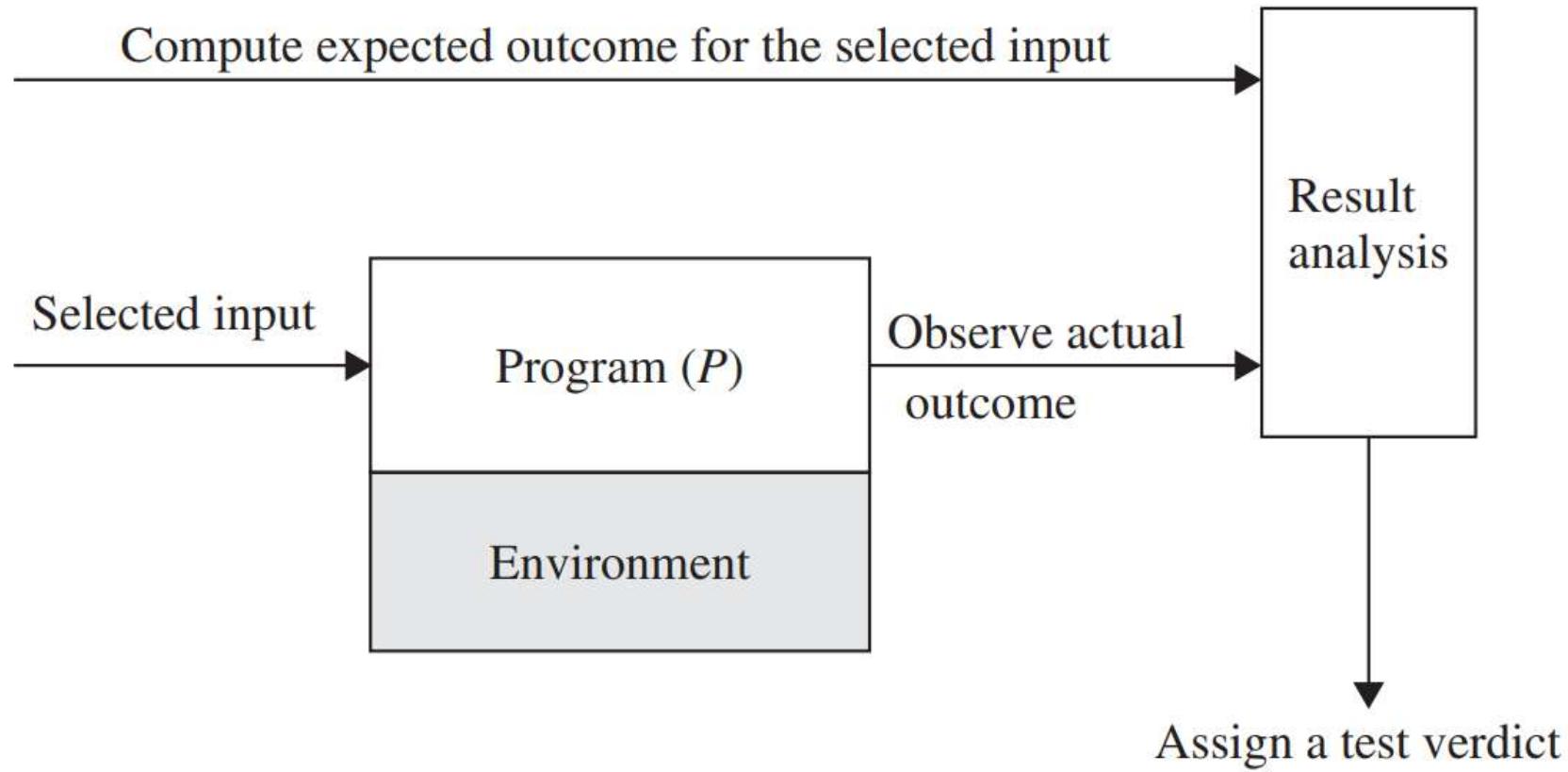


Figure 1.6 Different activities in program testing.

Test report

Como reproducir la falla

Analizar y describir la falla

Salida obtenida, caso de prueba,
entrada, salida esperada, ambiente
de ejecución

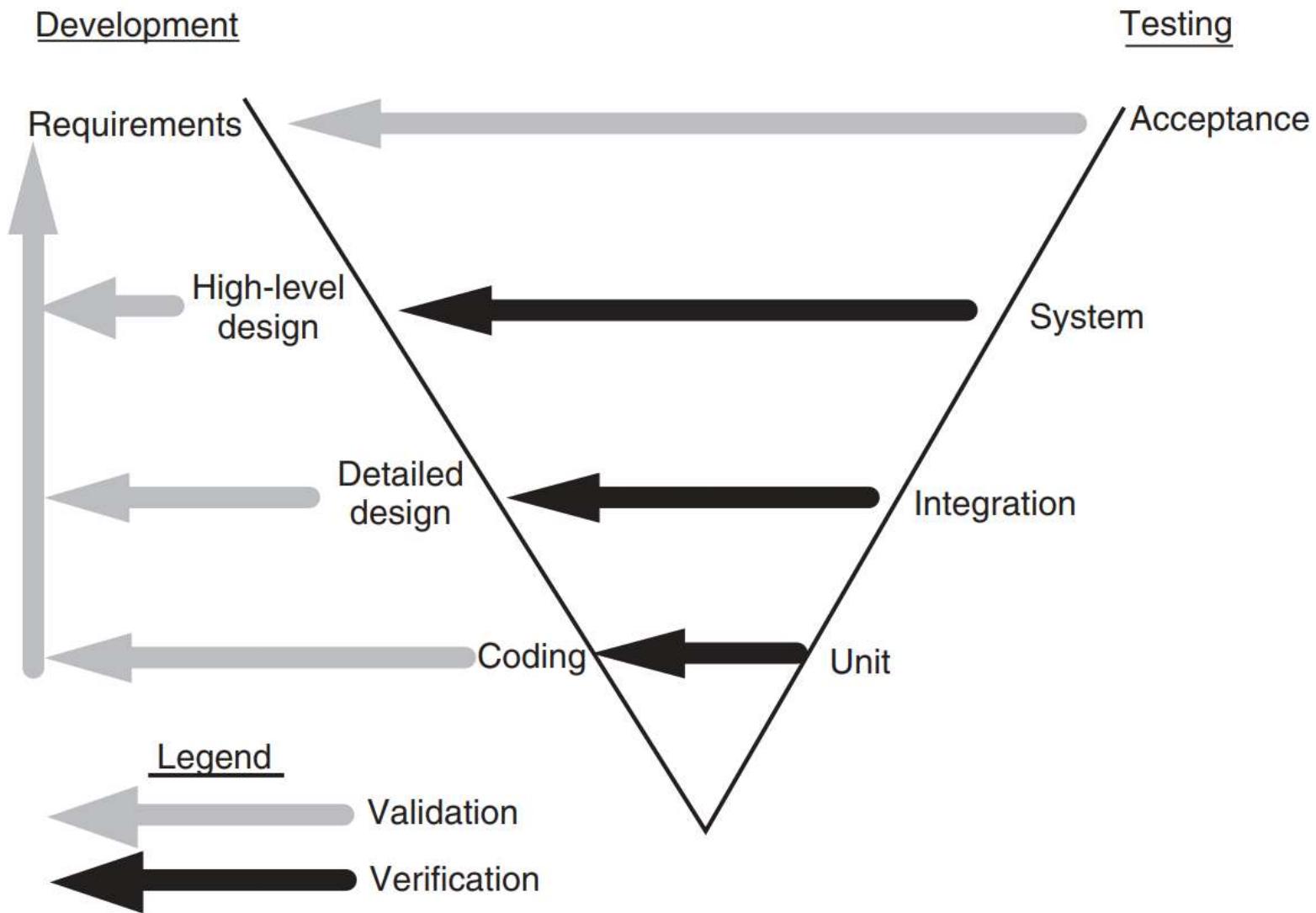


Figure 1.7 Development and testing phases in the V model.

Pruebas de regresión

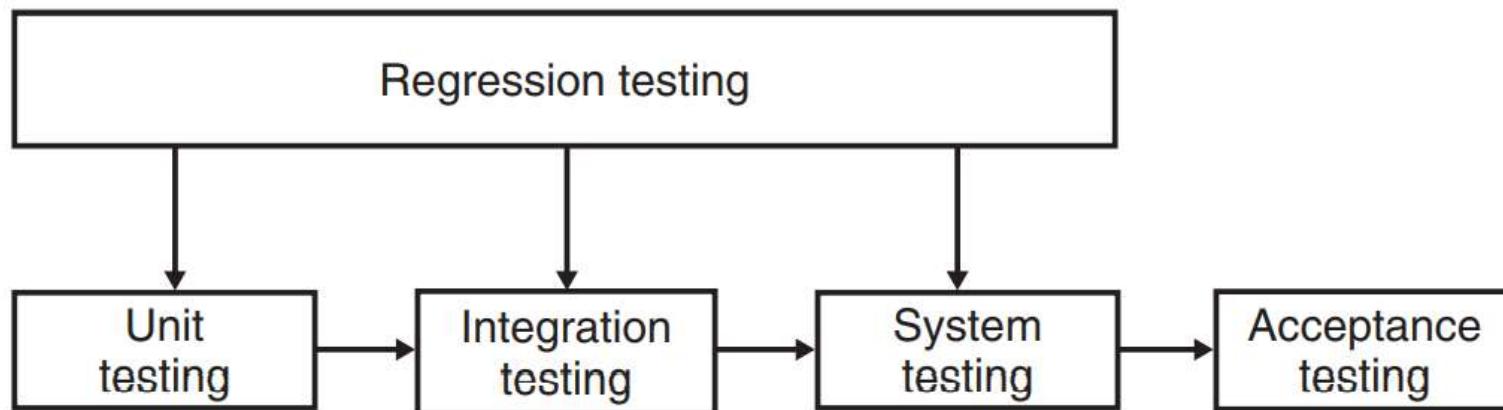


Figure 1.8 Regression testing at different software testing levels. (From ref. 41. © 2005 John Wiley & Sons.)

Diseñar casos de prueba

Fuentes de
información

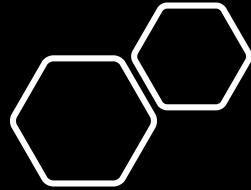
Requerimientos y
especificaciones
funcionales

Dominós de
entrada y salida

Código

Perfil operacional

Modelo de falla



Dominios de entrada y salida

- Valores especiales
- Valores frontera

```
factorial(0) = 1;  
factorial(1) = 1;  
factorial(n) = n * factorial(n-1);
```

```
factorial(n) = 1 * 2 * ... * n;
```

Perfil
operacional

Entender el
usuario

Distribución
probabilística

Modelo de fallas

Fallas

- Inicialización
- Lógica
- Interfaces

Pruebas:

- Error guessing
- Fault seeding
- Mutation Analysis

Error guessing

Analizar la situación y
adivinar que tipo de
fallas pueden existir

Diseñar pruebas para
sacar a la luz esas
fallas

Fault seeding



Insertar fallas



Probar la efectividad de pruebas

Mutation analysis

Probar la capacidad de detectar errores

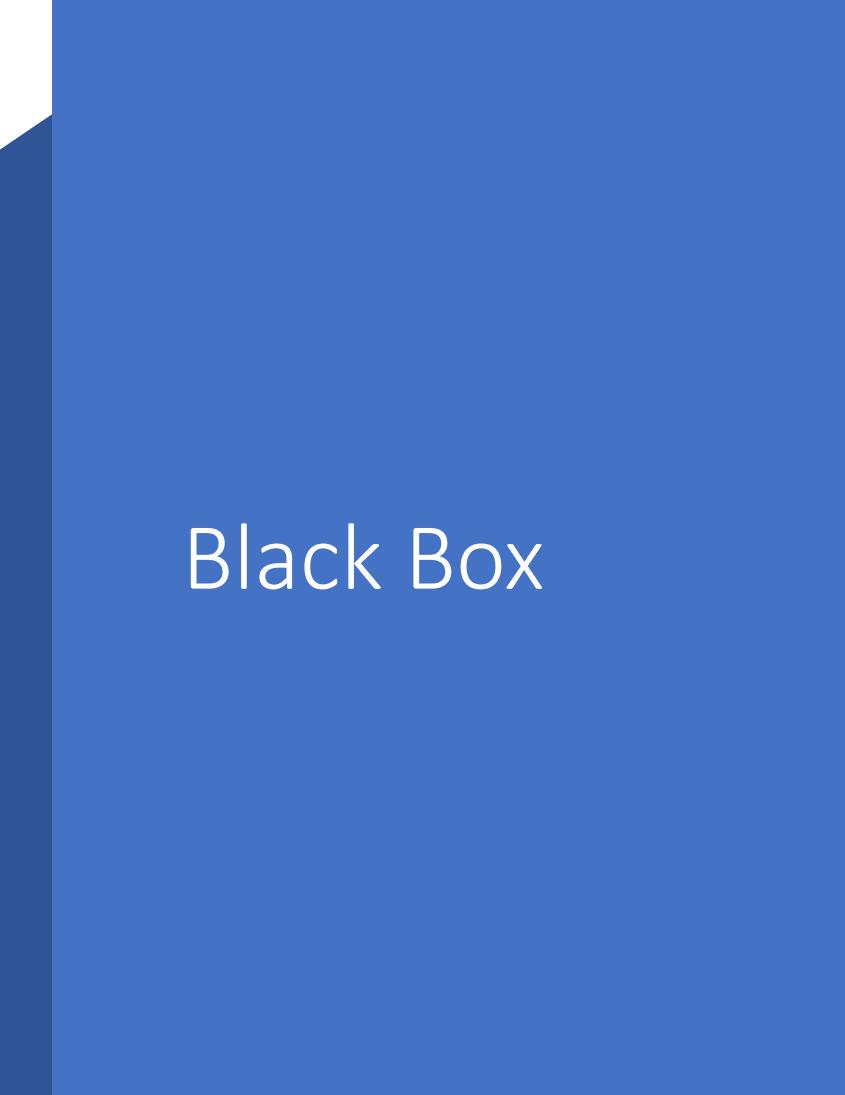
White box y Black box

White box

Estructural

- Código: Flujo de control y datos

Code
Coverage



Black Box



Funcional

Valores límite

Tabla de decisiones

Transición de estados

Casos de uso



Gray-Box

Casos de negocio

Pruebas end to end

Integración

Test Planning

Marco de trabajo

Alcance

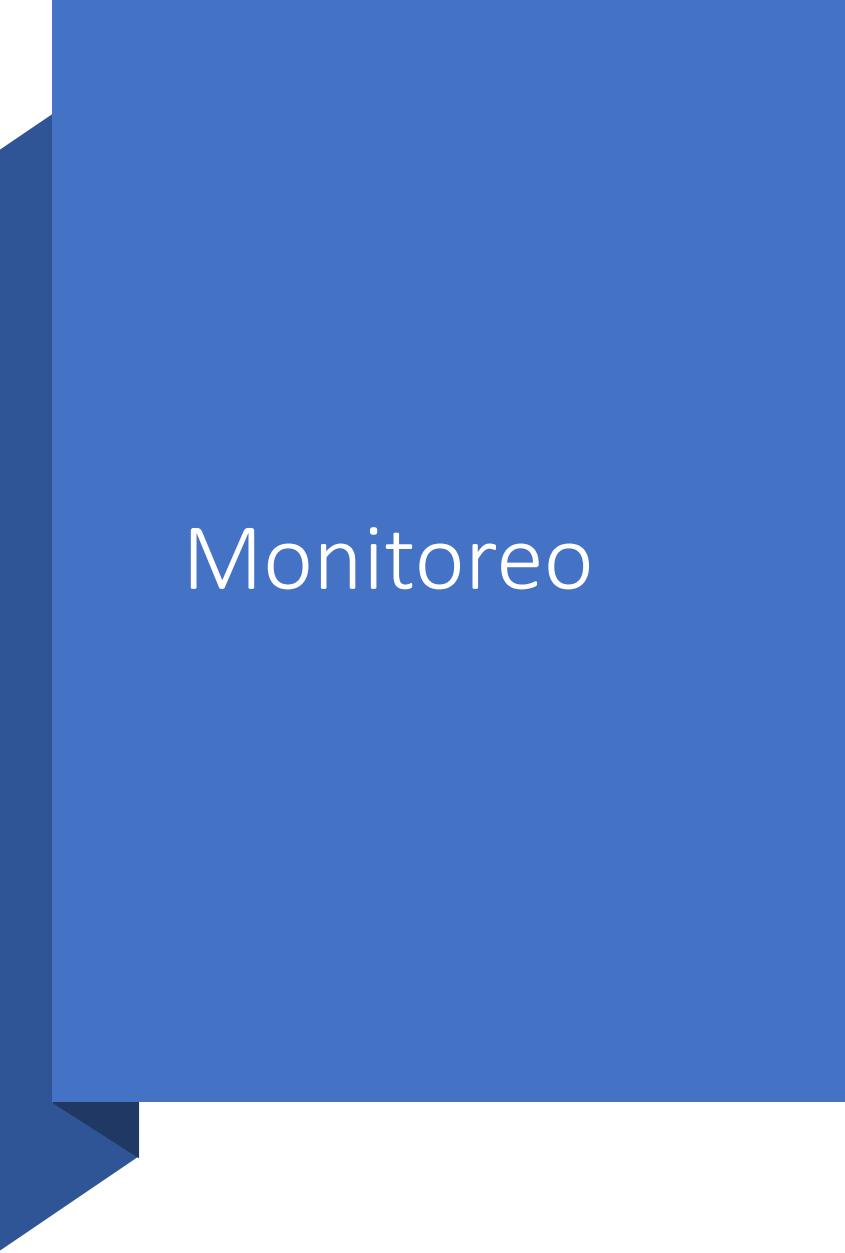
Recursos

Esfuerzo

Actividades

Presupuesto

Propósito



Monitoreo

Ejecución de pruebas

- Tiempo
- Recursos
- Alcance

Defectos

- Test aceptados / fallidos