Ingeniería del Software I

9 – Testing (Capítulo 10)

Testing

"El crecimiento es fruto del potencial acumulativo que siempre requiere tiempo. La destrucción por el contrario, es punto de fallos concretos que pueden ocurrir en segundos y de la pérdida de confianza que puede darse en un instante"

Morgan Housel '21

Testing

Principal objetivo de un proyecto: desarrollar un producto de alta calidad con alta productividad.

- La calidad tiene muchas dimensiones:
  Confiabilidad, mantenibilidad, interoperabilidad, etcétera.
- Confiabilidad es la más conocida.
- Habla de las posibilidades que el software falle:

Más defectos => más chances de que falle => menor confiabilidad.

Objetivo de calidad: que el producto entregado tenga la menor cantidad de defectos como sea posible.

Conceptos fundamentales

Defecto y desperfecto (Fault & failure)

Desperfecto: Un desperfecto de software ocurre si el comportamiento de éste es distinto del esperado/especificado.

Defecto: es lo que causa el desperfecto.

Defecto = bug

- Un desperfecto implica la presencia del defecto.
- La existencia del defecto no implica la ocurrencia del desperfecto.
- Pero: un defecto tiene el potencial para causar el desperfecto.
- Qué cosa es considerada un desperfecto depende del proyecto que se está llevando a cabo.

Testing

When software fails: “it did exactly what it was told to do. The reason it failed is that it was told to do the wrong thing"

Somers ‘17

Conceptos fundamentales

Defecto y desperfecto (Fault & failure)

Rol del testing:

- Las revisiones son procesos humanos: no pueden encontrar todos los defectos.
- En consecuencia habrá defectos de requerimientos, defectos de diseño y defectos de codificación dentro del código.
- Estos defectos tienen que identificarse por medio del testing.

=> El testing juega un rol crítico cuando se trata de garantizar calidad.

Conceptos fundamentales

Defecto y desperfecto (Fault & failure)

- Durante el testing, un programa se ejecuta siguiendo un conjunto de casos de test.
- Si hay desperfectos en la ejecución de un test, entonces hay defectos en el software.
- Si no ocurren desperfectos, entonces la confianza en el software crece.
- Pero: no podemos negar la presencia de defectos.
- Los defectos se detectan a través de los desperfectos.
- Para detectar los defectos debemos causar desperfectos durante el testing.
- Para identificar el defecto real, que causa el desperfecto, debemos recurrir al debugging.

Testing

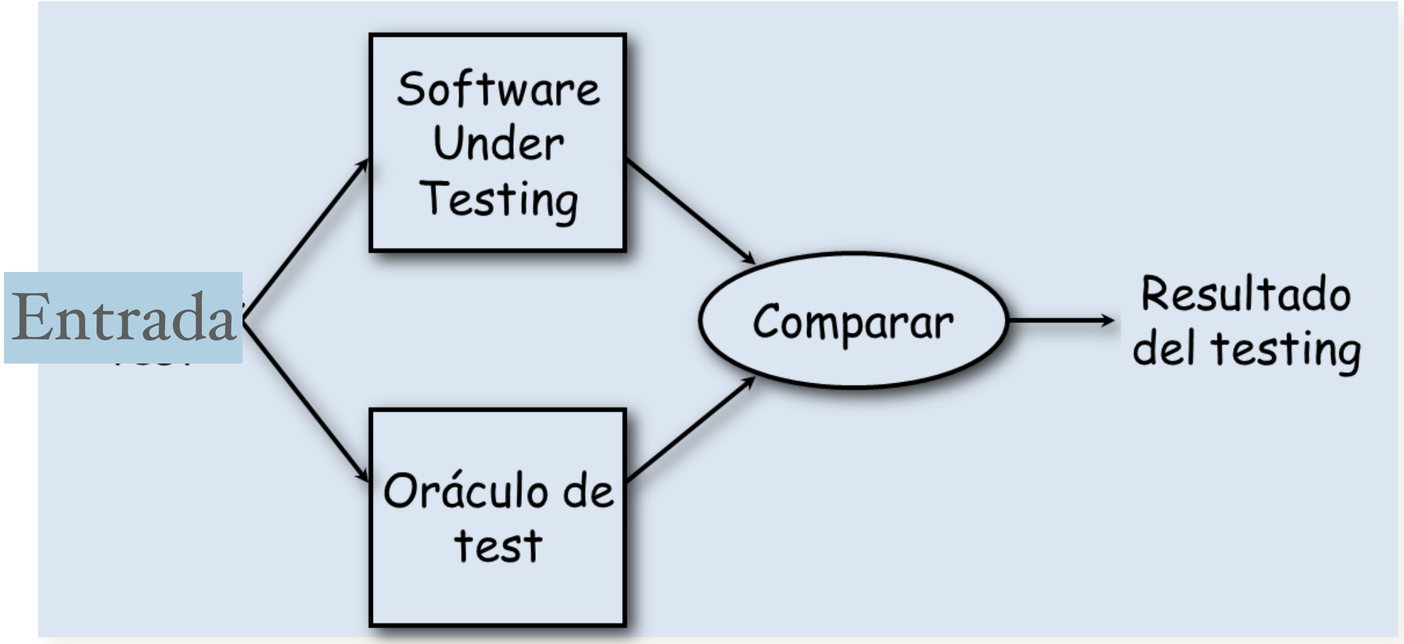
"Because defects are defined relative to intent, whether a behavior is a failure depends entirely the definition of intent"

Amy Ko

Conceptos fundamentales

Oráculos de tests

- Para verificar la ocurrencia de un desperfecto en la ejecución de un caso de test, necesitamos conocer el comportamiento correcto para este caso.
  i.e. necesitamos un oráculo de test.



Conceptos fundamentales

Oráculos de tests

- Idealmente pretendemos que el oráculo siempre dé el resultado correcto.
- Oráculo humano?
- Las mismas especificaciones pueden contener errores.
- En algunos casos, los oráculos pueden generarse automáticamente de la especificación.

Conceptos fundamentales

Casos de test y criterios de selección

- Si existen defectos, deseamos que los casos de test los evidencien a través de fallas.
- Deseamos:
  - Construir un conjunto de casos de test tal que la ejecución satisfactoria de todos ellos implique la ausencia de defectos. Conjunto de test => ¬ defectos.
  - Como el testing es costoso, deseamos que sea un conjunto reducido.
- Estos dos deseos son contradictorios. => Usar algún criterio de selección de tests.

Conceptos fundamentales

Casos de test y criterios de selección

- El criterio de selección especifica las condiciones que el conjunto de casos de test debe satisfacer con respecto al programa y/o a la especificación.
- Hay dos propiedades fundamentales que esperamos de los criterios de test:

  Confiabilidad & Validez

- Es prácticamente imposible obtener un criterio que sea confiable y válido al mismo tiempo, y que también sea satisfecho por una cantidad manejable de casos de test.

Conceptos fundamentales

- La psicología del testing es importante:
  - debe revelar los defectos (contrariamente a mostrar que funciona)
  - => los casos de test deben ser “destructivos”.
- Dos enfoques para diseñar casos de test:
  - de caja negra o funcional,
  - de caja blanca o estructural.
- Ambos son complementarios.

Testing de caja negra

- El software a testear se trata como una caja negra.

- La especificación de la caja negra está dada.
- Para diseñar los casos de test, se utiliza el comportamiento esperado del sistema. i.e. los casos de test se seleccionan sólo a partir de la especificación.

Testing de caja negra

- Premisa: el comportamiento esperado está especificado.
- Entonces sólo se definen test para el comportamiento esperado especificado.
- Para el testing de módulos: la especficificación producida en el diseño define el comportamiento esperado.
- Para el testing del sistema: la SRS define el comportamiento esperado.

Testing de caja negra

- El testing funcional más minucioso es el exhaustivo:
  - El software esta diseñado para trabajar sobre un espacio de entrada.
  - => Testear el software con todos los elementos del espacio de entrada.
- No es viable: demasiado costoso, si no es imposible.
- Se necesitan mejores métodos para seleccionar los casos de test.
- Veremos diferentes enfoques.

Testing de caja negra
Particionado por clase de equivalencia

Dividir el espacio de entrada en clases de equivalencias.

- Parte de esta idea: Si el software funciona para un caso de test en una clase => muy probablemente funcione de la misma manera para todos los elementos de la misma clase.
  - La obtención del particionado ideal es imposible.

    => Aproximar identificando las clases para las cuales se especifican distintos comportamientos.
- Base lógica: la especificación requiere el mismo comportamiento en todos los elementos de una misma clase.

  => Es muy probable que el software se construya de manera tal que falle para todos o para ninguno.
  Ejemplo: si una función no fue diseñada para números negativos => fallará para todos los números negativos.

Testing de caja negra
Particionado por clase de equivalencia

- Cada condición especificada como entrada es una clase de equivalencia.
- Para lograr robustez, se deben armar clases de equivalencias para entradas inválidas.
  Ejemplo: se especifica el rango $0 \leq x \leq MAX$. Luego:
  - el rango [0..MAX] forma una clase,
  - $x < 0$ define una clase inválida,
  - $x > MAX$ define otra clase inválida.
- Cuando el rango completo no se trate uniformemente => dividir en clases.

Testing de caja negra
Particionado por clase de equivalencia

- También se deben considerar las clases de equivalencia de los datos de salida. Generar los casos de test para estas clases eligiendo apropiadamente las entradas.

- Una vez elegidas las clases de equivalencia, se deben seleccionar los casos de test:
  1. Seleccionar cada caso de test cubriendo tantas clases como sea posible.
  2. O dar un caso de test que cubra a lo sumo una clase válida por cada entrada.

  - Además de los casos de test separados por cada clase inválida.

Testing de caja negra

Particionado por clase de equivalencia

Ejemplo:

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

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

Testing de caja negra
Particionado por clase de equivalencia

Ejemplo (continuación):

- Casos de test (i.e. los valores de s y n) según el método 1:
  - s con long < N y conteniendo mayúsculas, minúsculas, números y caracteres especiales, y n = 4.
  - Más un caso de test por cada clase de equivalencia inválida.
  - Total: 1 + 3 = 4 casos de test.
- Con el método 2:
  - Un string aparte por cada caso de test (i.e. uno para números, uno para minúsculas, etc.) + los casos inválidos.
  - Total: 5 + 3 = 8 casos de test.

Testing de caja negra

Análisis de valores límites

- Los programas generalmente fallan sobre valores especiales.
- Estos valores usualmente se encuentran en los límites de las clases de equivalencia.
- Los casos de test que tienen valores límites tienen alto rendimiento.
- También se denominan casos extremos.
- Un caso de test de valores límites es un conjunto de datos de entrada que se encuentra en el borde de las clases de equivalencias de la entrada o la salida.

Testing de caja negra

Análisis de valores límites

- Para cada clase de equivalencia:
  - Elegir valores en los límites de la clase.
  - Elegir valores justo fuera y dentro de los límites.
    Además del valor normal.
    Ejemplo: si $0 \leq x \leq 1$
    0 y 1 están en el límite, -0.1 y 1.1 están apenas afuera y 0.1 y 0.9 están dentro.

    Ejemplo: para una lista acotada: la lista vacía y la lista de mayor longitud.
- Considerar las salidas también y producir casos de test que generen salidas sobre los límites.

Testing de caja negra

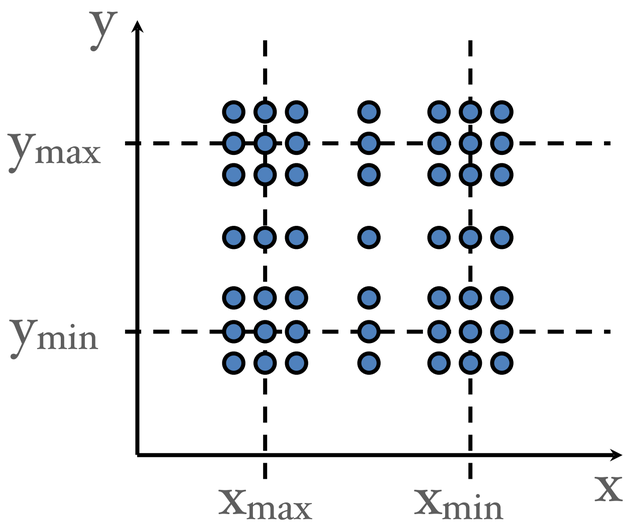
Análisis de valores límites

- En primer lugar se determinan los valores a utilizar para cada variable.
- Si la entrada tiene un rango definido => hay 6 valores de límite más un valor normal. Total: 7.
- Para el caso de múltiples entradas hay dos estrategias para combinarlas en un caso de test:
  1. Ejecutar todas las combinaciones posibles de las distinta variables, si hay **n** => 7^n casos de test!
  2. Seleccionar los casos límites para una variable y mantener las otras en casos normales y considerar el caso de todo normal. Total $6n + 1$.

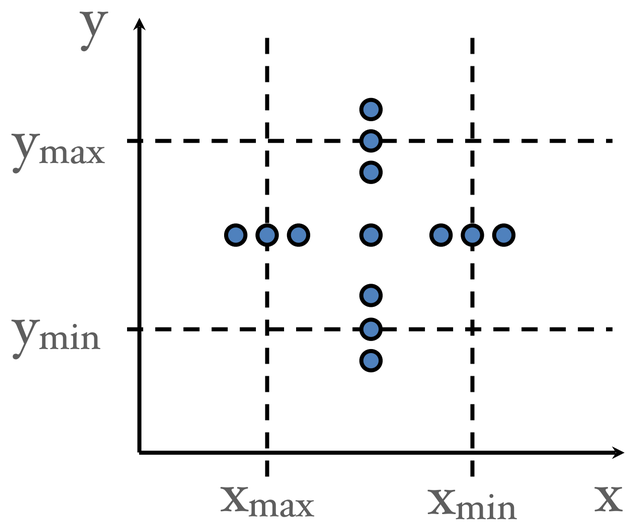
Testing de caja negra

Comparación

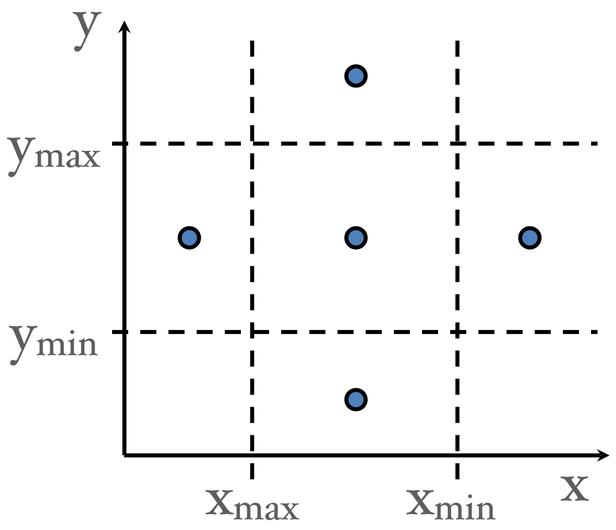
1er método



2do método



por clases de equivalencia



Testing de caja negra

Grafo de causa-efecto

- Los análisis de clase de equivalencia y valores límites consideran cada entrada separadamente.
- Para manipular las entradas distintas combinaciones de las clases de equivalencia deben ser ejecutadas.
- La cantidad de combinaciones puede ser grande:

  si hay n condiciones distintas en la entrada que puedan hacerse válidas o inválidas

  => 2^n clases de equivalencia.
- El grafo de causa-efecto ayuda a seleccionar las combinaciones como condiciones de entrada.

Testing de caja negra

Grafo de causa-efecto

- Identificar las causas y efectos en el sistema
  - Causa: distintas condiciones en la entrada que pueden ser verdaderas o falsas.
  - Efecto: distintas condiciones de salidas: verdaderas/falsas también.
- Identificar cuáles causas pueden producir qué efectos; las causas se pueden combinar.
- Causas y efectos son nodos en el grafo.
- Las aristas determinan dependencia: hay aristas “positivas” y “negativas”.
- Existen nodos “and” y “or” para combinar la causalidad.

Testing de caja negra
Grafo de causa-efecto

- A partir del grafo de causa-efecto se puede armar una tabla de decisión.
  - Lista las combinaciones de condiciones que hacen efectivo cada efecto.
- La tabla de decisión puede usarse para armar los distintos casos de test.

Testing de caja negra

Grafo de causa-efecto

Ejemplo:

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

Testing de caja negra

Grafo de causa-efecto

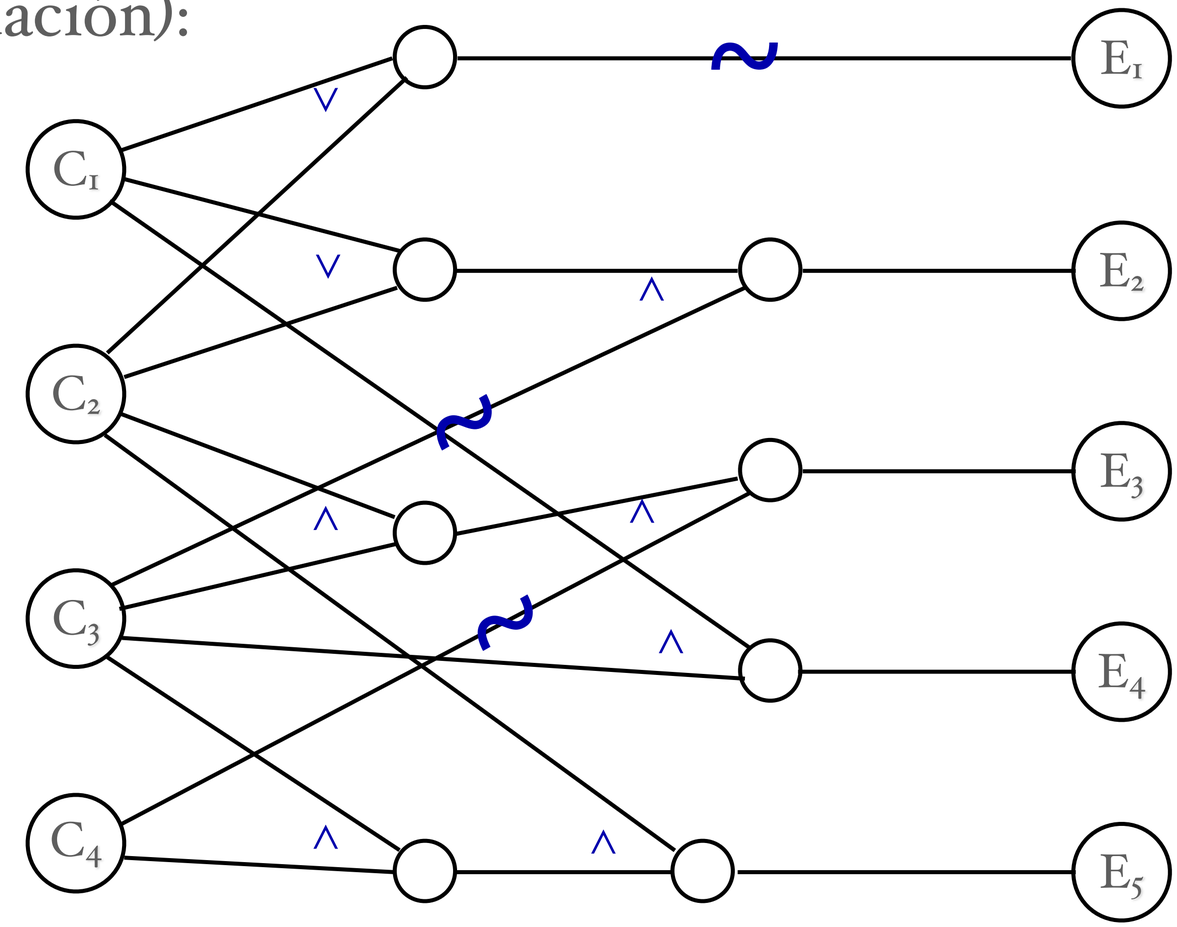
Ejemplo (continuación):

- Causas:
  - C1: el comando es “acreditar”
  - C2: el comando es “debitar”
  - C3: número de cuenta es válido
  - C4: la cantidad es válida
- Efecto:
  - E1: Imprimir “Comando inválido”
  - E2: Imprimir “Número de cuenta inválida”
  - E3: Imprimir “Cantidad débito inválida”
  - E4: Acreditar en la cuenta
  - E5: Debitar de la cuenta

Testing de caja negra

Grafo de causa-efecto

Ejemplo (continuación):



Testing de caja negra

Grafo de causa-efecto

Ejemplo (continuación):

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

Testing de caja negra

Testing de a pares

- Usualmente muchos parámetros determinan el comportamiento del sistema.
- Los parámetros pueden ser entradas o seteos y pueden tomar distintos valores, o distintos rangos de valores.
- Muchos defectos involucran sólo una condición, defecto de modo simple.
  Ej.: el software no puede imprimir en un tipo dado de impresora.
  - Los defectos de modo simple pueden detectarse verificando distintos valores de los distintos parámetros.
  - Si hay n parámetros con m, tipos de valores c/u, podemos testear cada valor distinto en cada test por cada parámetro.
    i.e: m casos de test en total.

Testing de caja negra

Testing de a pares

- Pero no todos los defectos son de modo simple: el software puede fallar en combinaciones:
  Ejemplo: el software de la cuenta del teléfono calcula mal las llamadas nocturnas, un parámetro, a un país particular, otro parámetro.
  Ejemplo: el sistema de reserva falla en las reservas en clase ejecutiva (parámetro 1) para menores (parámetro 2) que no abarcan un fin de semana (parámetro 3)
- Los defectos de modo múltiple se revelan con casos de test que contemplen las combinaciones apropiadas.
- Esto se denomina test combinatorio.

Testing de caja negra

Testing de a pares

- Pero el test combinatorio no es factible:
  - n parámetros / m valores => nm casos de test.
  - Si **n** = **m** = 5, n^m = 3125. Si cada test tarda 5 min => más de un mes testeando.
- Se investigó que la mayoría de tales defectos se revelan con la interacción de pares de valores, i.e. la gran mayoría de los defectos tienden a ser de modo simple o de modo doble.
- Para modo doble necesitamos ejercitar cada par => se denomina testing de a pares.

Testing de caja negra

Testing de a pares

- En testing de a pares, todos los pares de valores deben ser ejercitados.
- Si n parámetros / m valores, para cada par de parámetros tenemos m*m pares:

  el 1er param. tendrá m*m contra n-1 otros

  el 2do param. tendrá m*m contra n-2 otros

  el 3er param. tendrá m*m contra n-3 otros ...

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

  En el ejemplo: $5^2*5*4/2 = 250$ (menos de 3 días).

Testing de caja negra

Testing de a pares

- Un caso de test consiste en algún seteo de los n parámetros.
- El menor conjunto de casos de test se obtiene cuando cada par es cubierto sólo una vez.
- Un caso de test puede cubrir hasta $(n-1)+(n-2)+\ldots = n*(n-1)/2$ pares.
- En el mejor caso, cuando cada par es cubierto exactamente una vez, tendremos m^2 casos de test distintos que proveen una cobertura completa, en el ejemplo: $5^2=25$ => hasta un min de 2 hs 5 min.

Testing de caja negra

Testing de a pares

- La generación del conjunto de casos de test más chico que provea cobertura completa de los pares no es trivial.
- Existen algoritmos eficientes para generación de casos de test que pueden reducir el esfuerzo de testing considerablemente.
  - En un caso de 13 parámetros cada uno con 3 valores, la cobertura de a pares puede resultar en 15 casos de test.
- El testing de a pares es un enfoque práctico y muy utilizado en la industria.

Testing de caja negra

Testing de a pares

Ejemplo:

- Supongamos un producto de software multiplataforma que usa browsers como interfaz y debe trabajar sobre distintos sistemas operativos:
- Tenemos tres parámetros:
  - SO (parámetro A): Linux, MacOSX, Windows vista.
  - Tamaño memoria (B): 512MB, 1GB, 2GB
  - Browser (C): Firefox, Opera, Safari/Konqueror
- Número total de combinaciones de a pares: 27
- Cantidad total de casos de test puede ser menor.

Testing de caja negra

Testing de a pares

Ejemplo (continuación):

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

Testing de caja negra
Casos especiales

- Los programas usualmente fallan en casos especiales.
- Estos dependen de la naturaleza de la entrada, tipos de estructuras de datos, etcétera.
- No hay buenas reglas para identificarlos.
- Una forma es adivinar cuando el software puede fallar y crear esos casos de test.
- También se denomina “adivinanza del error”.
- La idea es jugar al abogado del diablo y cuestionar los puntos débiles.

Testing de caja negra

Casos especiales

- Usar la experiencia y el juicio para adivinar situaciones donde el programador pueda haber cometido errores.
- Los casos especiales pueden ocurrir debido a suposiciones sobre la entrada, usuario, entorno operativo, negocio, etcétera.
- Ejemplo: Volviendo al programa que contaba las palabras:

  archivo vacío, archivo inexistente, archivo que tiene solo blancos, o con una sola palabra, o múltiples blancos entre palabras, o líneas en blanco consecutivas, o blancos al comienzo, palabras ordenadas, blancos al final del archivo, etcétera.
- El testing por adivinanza del error es quizás el más utilizado en la práctica.

Testing de caja negra

Testing basado en estados

- Algunos sistemas no tienen estados: para las mismas entradas, se exhiben siempre las mismas salidas.
- En muchos sistemas, el comportamiento depende del estado del sistema, i.e. para la misma entrada, el comportamiento podría diferir en distintas ocasiones
  i.e. el comportamiento y la salida depende tanto de la entrada como del estado actual del sistema.
- El estado del sistema representa el impacto acumulado de las entradas pasadas.
- El testing basado en estado está dirigido a tales sistemas.

Testing de caja negra

Testing basado en estados

- Un sistema puede modelarse como una máquina de estados.
- El espacio de estados puede ser demasiado grande, es el producto cartesiano de todos los dominios de todas las variables.
- El espacio de estados puede particionarse en pocos estados, cada uno representando un estado lógico de interés del sistema.
- El modelo de estado se construye generalmente a partir de tales estados.

Testing de caja negra

Testing basado en estados

Un modelo de estados tiene cuatro componentes:

- Un conjunto de estados: son estados lógicos representando el impacto acumulativo del sistema.
- Un conjunto de transiciones: representa el cambio de estado en respuesta a algún evento de entrada.
- Un conjunto de eventos: son las entradas al sistema.
- Un conjunto de acciones: son las salidas producidas en respuesta a los eventos de acuerdo al estado actual.

Testing de caja negra

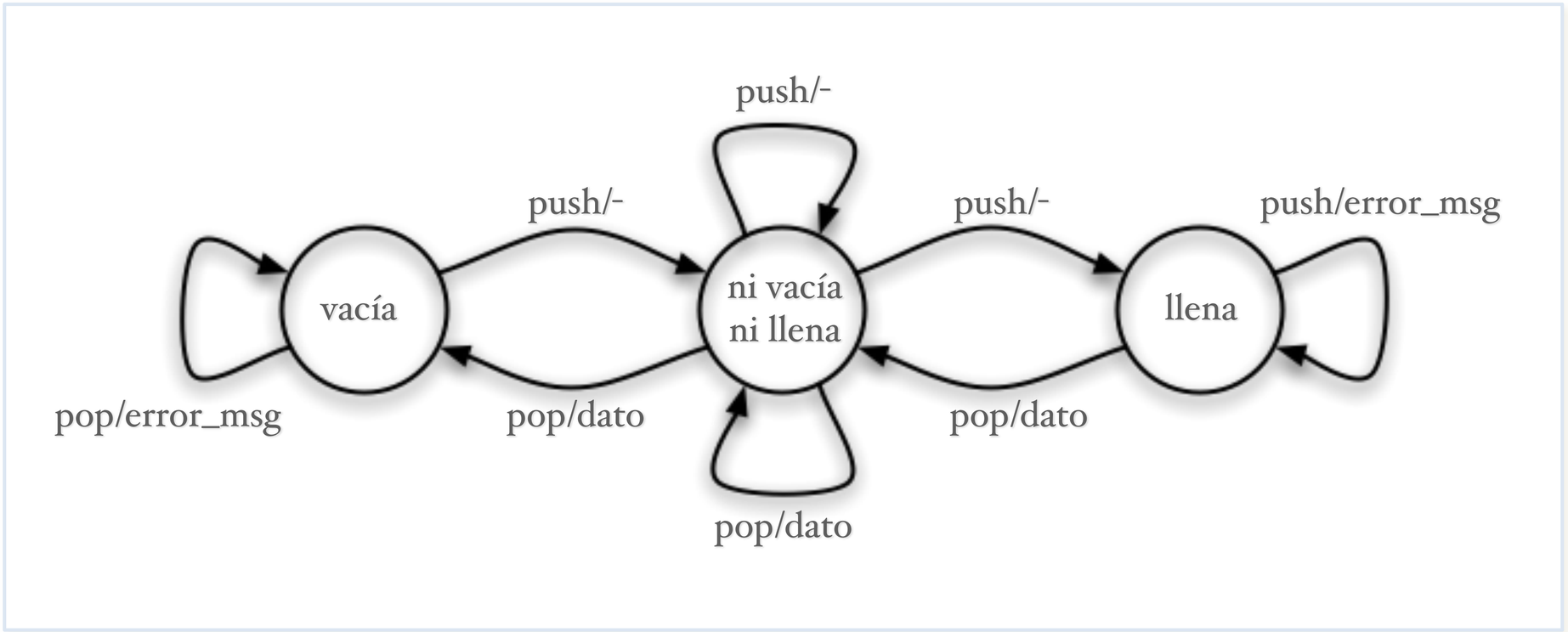
Testing basado en estados

- El modelo de estado muestra la ocurrencia de las transiciones y las acciones que se realizan.
- Usualmente el modelo de estado se construye a partir de las especificaciones o los requerimientos.
- El desafío más importante es, a partir de la especificación/requerimientos, identificar el conjunto de estados que captura las propiedades claves pero es lo suficientemente pequeño como para modelarlo.

Testing de caja negra

Testing basado en estados

Ejemplo: Una pila de tamaño acotado.
El interés esta en los casos en que la pila está vacía, llena, o ni uno ni lo otro.



Testing de caja negra

Testing basado en estados

- El modelo de estado puede crearse a partir de la especificación o del diseño.
- En el caso de los objetos, los modelos de estado se construyen usualmente durante el proceso del diseño.
- Los casos de test se seleccionan con el modelo de estado y se utilizan posteriormente para testear la implementación.
- Existen varios criterios para generar los casos de test. Ejemplos:
  - Cobertura de transiciones: el conjunto T de casos de test debe asegurar que toda transición sea ejecutada.
  - Cobertura de par de transiciones: T debe ejecutar todo par de transiciones adyacentes que entran y salen de un estado.
  - Cobertura de árbol de transiciones: T debe ejecutar todos los caminos simples, del estado inicial al final o a uno no visitado.

Testing de caja negra

Testing basado en estados

- El test basado en estados se enfoca en el testing de estados y transiciones.
- Se testean distintos escenarios que de otra manera podrían pasarse por alto.
- El modelo de estado se realiza usualmente luego de que la información de diseño se hace disponible.
- En este sentido, se habla a veces de testing de caja gris, dado que no es de caja negra puro.

Testing de caja blanca

- El testing de caja negra se enfoca sólo en la funcionalidad:
  - Lo que el programa hace no cómo implementa.
- El testing de caja blanca se enfoca en el código:
  - El objetivo es ejecutar las distintas estructuras del programa con el fin de descubrir errores.
- Los casos de test se derivan a partir del código.
- Se denomina también testing estructural.
- Existen varios criterios para seleccionar el conjunto de casos de test.

Testing de caja blanca

Tipos de testing estructural:

- Criterio basado en el flujo de control.
  - Observa la cobertura del grafo de flujo de control.
- Criterio basado en el flujo de datos.
  - Observa la cobertura de la relación definición-uso en las variables.
- Criterio basado en mutación.
  - Observa a diversos mutantes del programa original.

Nos vamos a enfocar en los dos primeros criterios, el tercero lo leen del libro.

Testing de caja blanca

Criterio basado en flujo de control

- Considerar al programa como un grafo de flujo de control.
  - Los nodos representan bloques de código, i.e., conjuntos de sentencias que siempre se ejecutan juntas.
  - Una arista (i, j) representa una posible transferencia de control del nodo i al j.
- Suponemos la existencia de un nodo inicial y un nodo final.
- Un camino es una secuencia del nodo inicial al nodo final.

Testing de caja blanca

Criterio basado en flujo de control

1. Criterio de cobertura de sentencia
2. Criterio de cobertura de ramificaciones
3. Criterio de cobertura de caminos

Testing de caja blanca

Criterio basado en flujo de control

Criterio de cobertura de sentencia

- Cada sentencia se ejecuta al menos una vez durante el testing.
  i.e. el conjunto de caminos ejecutados durante el testing debe incluir todos los nodos.
- Limitación: puede no requerir que una decisión evalúe a falso en un if si no hay else:
  Ejemplo:

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

  El conjunto de casos de test { (x = 0, 0) } tiene el 100% de cobertura pero el error pasa desapercibido.

- No es posible garantiza 100% de cobertura debido a que puede haber nodos inalcanzables.

Testing de caja blanca

Criterio basado en flujo de control

Test suite: conjunto de casos de tests

Criterio de cobertura de ramificaciones

- Cada arista debe ejecutarse al menos una vez en el testing.
  i.e. cada decisión debe ejercitarse como verdadera y como falsa durante el testing.
- La cobertura de ramificaciones implica cobertura de sentencias.
- Si hay múltiples condiciones en una decisión luego no todas las condiciones se ejercitan como verdadera y falsa.

Todo test suite que cubre ramificaciones, también cubre sentencias.
Pero puede haber test suites que cubran ramificaciones que no encuentren el defecto mientras haya tests suites que cubren sentencias que si lo detecten.

Ejemplo:

```
abs(x) :   if (x<=0) then x=x;
           return(x)
```

Cobertura de Sentencias: { (x=-1, 1) }
Cobertura Ramificaciones: { (x=1, 1); (x=0, 0) }

Testing de caja blanca

Criterio basado en flujo de control

Criterio de cobertura de caminos

- Todos los posibles caminos del estado inicial al final deben ser ejercitados.
- Cobertura de caminos implica cobertura de bifurcación.
- Problema: la cantidad de caminos puede ser infinita, considerar loops.
- Notar además que puede haber caminos que no son realizables.

Testing de caja blanca

Criterio basado en flujo de control

- Existen criterios intermedios, entre el de caminos y el de bifurcación: cobertura de predicados, basado en complejidad ciclomática, etcétera.
- Ninguno es suficiente para detectar todos los tipos de defectos, ejemplo: se ejecutan todos los caminos pero no se detecta una división por 0.
- Proveen alguna idea cuantitativa de la “amplitud” del conjunto de casos de test.
- Se utiliza más para evaluar el nivel de testing que para seleccionar los casos de test.

Testing de caja blanca

Criterio basado en flujo de datos

- Se construye un grafo de definición-uso etiquetando apropiadamente el grafo de flujo de control.
- Una sentencia en el grafo de flujo de control puede ser de tres tipos:
  - def: representa la definición de una variable, i.e. cuando la var está a la izquierda de la asignación.
  - uso-c: cuando la variable se usa para cómputo.
  - uso-p: cuando la variable se utiliza en un predicado para transferencia de control.

Testing de caja blanca

Criterio basado en flujo de datos

- El grafo de def-uso se construye asociando variables a nodos y aristas del grafo de flujo de control:
  - Por cada nodo i, def(i) es el conjunto de variables para el cual hay una definición en i.
  - Por cada nodo i, c-use(i) es el conjunto de variables para el cual hay un uso-c.
  - Para una arista (i,j), p-use(i,j) es el conjunto de variables para el cual hay un uso-p.
- Un camino de i a j se dice libre de definiciones con respecto a una var x si no hay definiciones de x en todos los nodos intermedios.

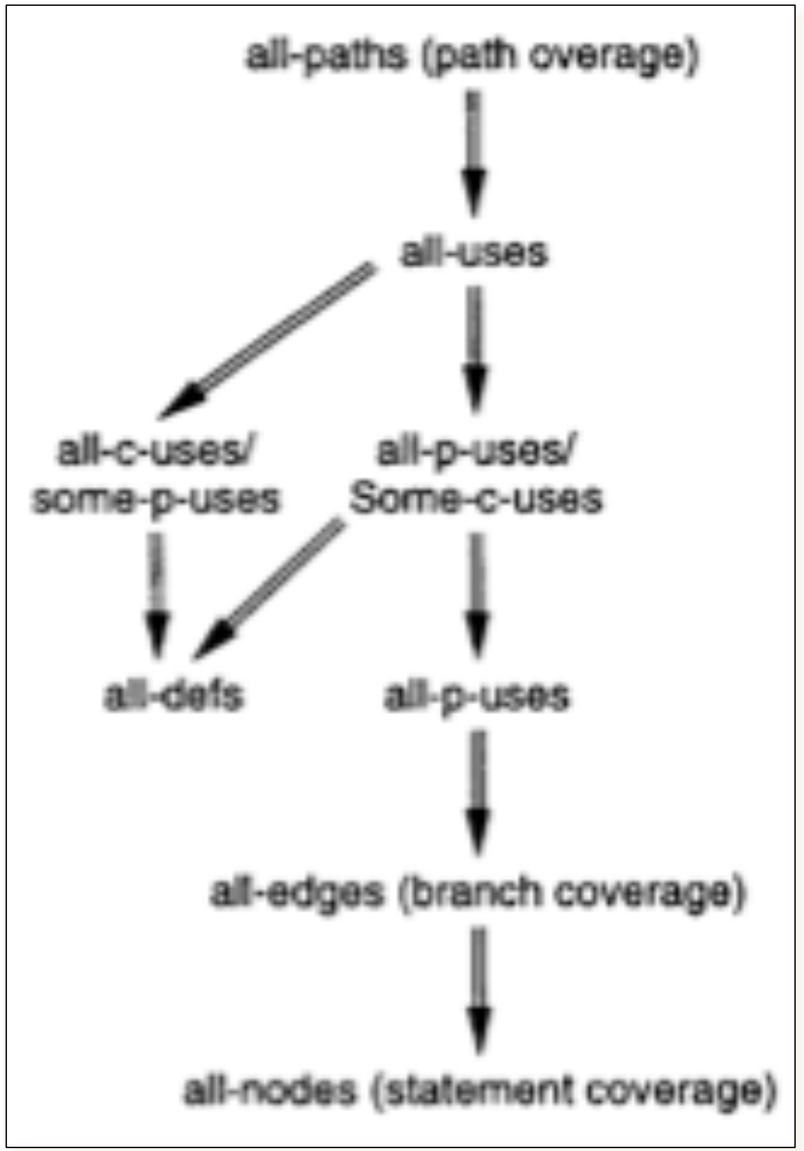
Testing de caja blanca

Criterio basado en flujo de datos

Criterios:

- todas las definiciones: por cada nodo i y cada x en def(i) hay un camino libre de definiciones con respecto a x hasta un uso-c o uso-p de x.
- todos los usos-p: todos los usos-p de todas las definiciones deben testearse.
- otros criterios: todos los usos-c, algunos usos-p, algunos usos-c.

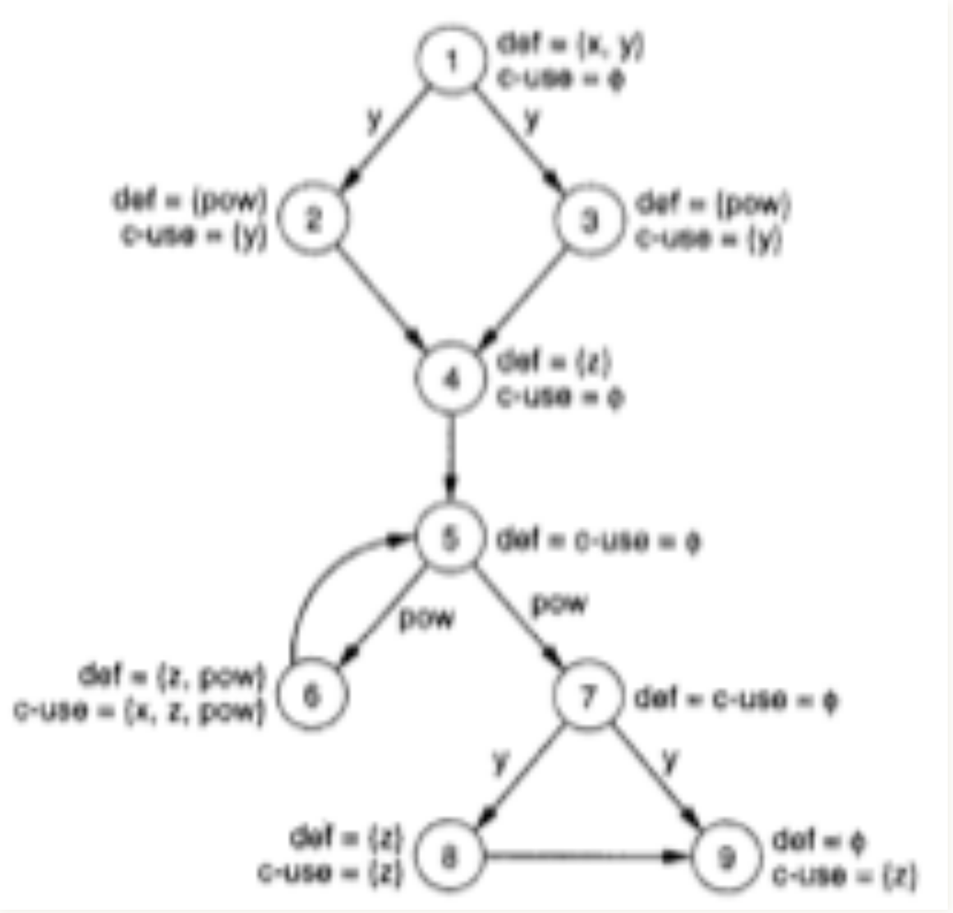
Relaciones entre los distintos criterios



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

Ejemplo

1. Identificar el conjunto de caminos que satisface el criterio deseado.
2. Identificar los casos de test que ejercitan dichos caminos.



Cobertura de ramificación:
Caminos: (1,2,4,5,6,5,7,8,9) y (1,3,4,5,6,5,7,9)
Tests: {((x=3, y=1); z=…) ; ((x=3, y =-1); z=…)}

Cobertura de todas las definiciones:
Caminos: (1,2,4,5,6,5,6,5,7,8,9) y (1,3,4,5,6,5,7,9)
Tests: {((x=3, y=2); z=…) ; ((x=3, y =-1); z=…)}

Soporte con herramientas

- Una vez elegido el criterio surgen dos problemas:
  - ¿El test suite satisface el criterio?
  - ¿Como generar el test suite que asegure cobertura?
- Para determinar cobertura se pueden usar herramientas.
  - Usualmente son de asistencia. El problema de generación de test que cubra un criterio es habitualmente indecidible.
- Las herramientas dicen qué sentencias o ramificaciones quedan sin cubrir.
- El proceso de selección de casos de test es mayormente manual.

Comparación y uso

- Se deben utilizar tanto test funcionales, de caja negra, como estructurales, de caja blanca.
- Ambas técnicas son complementarias:
  - Caja blanca => bueno para detectar errores en la lógica del programa, i.e. errores estructurales del programa.
  - Caja negra => bueno para detectar errores de entrada/salida, i.e. errores funcionales.
- Los métodos estructurales son útiles a bajo nivel solamente, donde el programa es “manejable”. Ejemplo: test de unidad.
- Los métodos funcionales son útiles a alto nivel, donde se busca analizar el comportamiento funcional del sistema o partes de éste.

El proceso de testing

Testing incremental

- Los objetivos del testing son:
  - detectar tantos defectos como sea posible, y
  - hacerlo a bajo costo.
- Objetivos contrapuestos: incrementar el testing permite encontrar más defectos pero a la vez incrementa el costo.
- Testing incremental: agregar partes no testeadas incrementalmente a la parte ya testeada.
- El testing incremental es esencial para conseguir los objetivos antedichos:
  - ayuda a encontrar más defectos y
  - ayuda a la identificación y eliminación.
- El testing de grandes sistemas se realiza siempre de manera incremental.

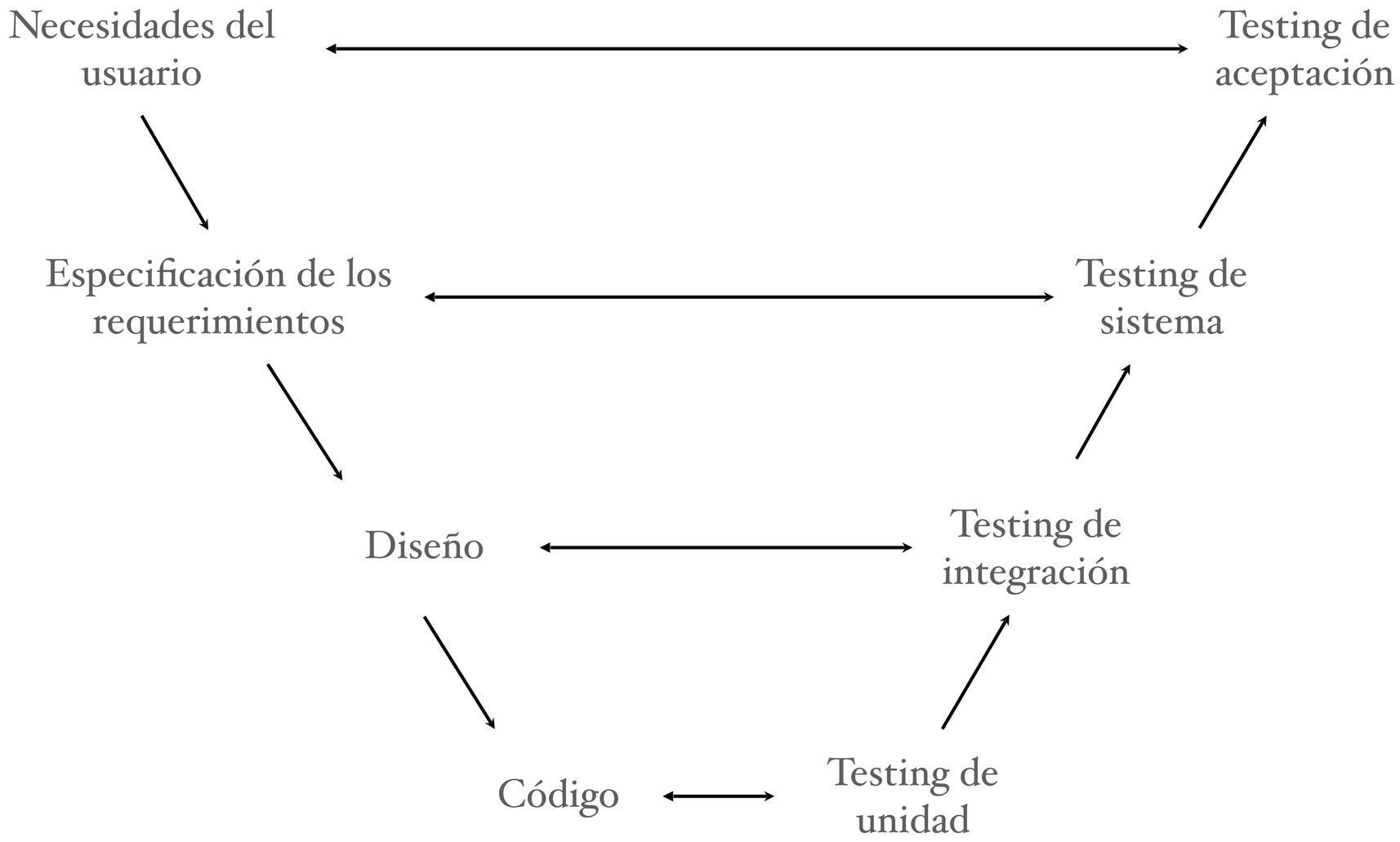
El proceso de testing

Niveles de testing

- El código contiene defectos de requerimiento, de diseño y de codificación.
- La naturaleza de los defectos es diferente para cada etapa de inyección del defecto.
- Un sólo tipo de testing sería incapaz de detectar los distintos tipos de defectos.
- Por lo tanto se utilizan distintos niveles de testing para revelar los distintos tipos de defectos.

El proceso de testing

Niveles de testing



El proceso de testing

Niveles de testing

Testing de unidad:

- Los distintos módulos del programa se testean separadamente contra el diseño, que actúa como especificación del módulo.
- Se enfoca en los defectos inyectados durante la codificación.
  => El objetivo es testear la lógica interna de los módulos.
- Frecuentemente el mismo programador realiza el test de unidad. La fase de codificación se suele denominar también de “codificación y testing de unidad”.

El proceso de testing

Niveles de testing

Testing de integración:

- Se enfoca en la interacción de módulos de un subsistema.
- Los módulos que ya fueron testeados unitariamente se combinan para formar subsistemas, los que son sujetos a testing de integración.
- Los casos de tests deben generarse con el objeto de ejercitar de distinta manera la interacción entre los módulos.
  => énfasis en el testing de las interfaces entre los módulos.
- Se podría omitir si el sistema no es muy grande.

El proceso de testing

Niveles de testing

Testing del sistema:

- El sistema de software completo es testeado.
- Se enfoca en verificar si el software implementa los requerimientos.
- Realiza el ejercicio de validar el sistema con respecto a los requerimientos.
- Generalmente es la etapa final del testing antes de que el software sea entregado.
- Debería ser realizado por personal independiente.
  Pero: los defectos son eliminados por los desarrolladores.
- Es la fase de testing que consume mas tiempo.

El proceso de testing

Niveles de testing

Testing de aceptación:

- Se enfoca en verificar que el software satisfaga las necesidades del usuario.
- Generalmente se realiza por el usuario/cliente en el entorno del cliente y con datos reales.
  Pero: los defectos son eliminados por los desarrolladores.
- Sólo después de que el testing de aceptación resulte satisfactorio, el software es puesto en ejecución (“deployed”).
- El plan del test de aceptación se basa en el criterio del test de aceptación y la SRS.

El proceso de testing

Niveles de testing

Otras formas de testing:

- Testing de desempeño:
  - Requiere de herramientas para medir el desempeño.
- Testing de estrés (stress testing):
  - El sistema se sobre carga al máximo; requiere de herramientas de generación de carga.
- Testing de regresión:
  - Se realiza cuando se introduce algún cambio al software. ¡es importante de realizar!
  - Verifica que las funcionalidades previas continúen funcionando bien.
  - Se necesitan los registros previos para poder comparar.
    => tests deben quedar apropiadamente documentados (+ scripts para automatizarlos).
  - Priorizar los casos de tests necesarios cuando el test suite completo no pueda ejecutarse cada vez que se realiza un cambio.

El proceso de testing

El plan de test

- El testing usualmente comienza con la realización del plan de test y finaliza con el testing de aceptación.
- El plan de test es un documento general que define el alcance y el enfoque del testing para el proyecto completo.
- Entradas: plan del proyecto, SRS, diseño.
- El plan de test identifica qué niveles de testing se realizarán, qué unidades serán testeadas, etc.
- Se puede realizar significativamente antes de comenzar con la tarea del testing real, conjuntamente con las actividades de diseño y codificación.

El pro sting

Necesario para asegurar que el plan de test es consistente con el plan de calidad del proyecto y que el cronograma de testing es consistente con el del proyecto

Forman la documentación básica utilizada para seleccionar las unidades de test y decidir los enfoques de testing utilizados.

- El testing usualmente comie plan d finaliza con el testing de aceptación.
- El plan de test es un documento general que define e l testing para el proyecto completo.
- Entradas: plan del proyecto, SRS, diseño.
- El plan de test identifica qué niveles de testing se realizarán, qué unidades serán testeadas, etc.
- Se puede realizar significativamente antes de comenzar con la tarea del testing real, conjuntamente con las actividades de diseño y codificación.

El proceso de testing

El plan de test

Usualmente contiene:

1. Especificación de la unidad de test: qué unidad necesita testearse separadamente.
2. Características a testear: esto incluye funcionalidad, desempeño, usabilidad, restricciones de diseño, ...
3. Enfoque: criterios a utilizarse, cuando detenerse, como evaluar, etcétera.
4. “Entregables”, i.e. testing deliverables.
5. Cronograma y asignación de tareas.

El pr ng

Una unidad de test es un conjunto de uno o más módulos conjuntamente con datos asociados que son el objeto del testing.

Atención: las unidades de test deben ser "testeables".

Usualmente contiene:

1. Especificación de la unidad de test: qué unidad necesita testear separadamente.
2. Características a testear: esto incluye funcionalidad, desempe ad, restricciones de diseño, ...
3. Enfoque: criterios a utilizarse, cuando detenerse, como
4. "Entregables", i.e. testing deliverables
5. Cronograma y asignación de tareas.

Establece qué nivel de testing realizar.

Ej.: lista de casos de test utilizados, resultados detallados del testing (incluyendo lista de defectos encontrados), reporte resumido, o información sobre cobertura.

El proceso de testing
Especificación de los casos de test

- El plan de test se enfoca en cómo proceder y qué testear, pero no trata con los detalles del testeo de una unidad.
- La especificación de casos de test se tiene que realizar separadamente para cada unidad.
- Por cada unidad de test se determinan los casos de test de acuerdo con en el plan: enfoques, características, etcétera.
- Conjuntamente, con cada caso de test se especifica:
  - las entradas a utilizar,
  - las condiciones que éste testeará, y
  - el resultado esperado.

El proceso de testing

Especificación de los casos de test

- La efectividad y costo del testing dependen del conjunto de casos de test seleccionado.
- ¿Cómo determinar si un conjunto de casos de test es bueno? i.e. que detecte la mayor cantidad de defectos y que ningún conjunto más pequeño también lo encuentre.
- No existe una manera de determinar la bondad; usualmente el conjunto de casos de test es revisado por expertos.
- Esto requiere que los casos de test se especifiquen antes de realizar el testing: una razón clave para tener especificaciones de casos de test.

El proceso de testing

Especificación de los casos de test

- La efectividad y costo del testing dependen del conjunto de casos de test seleccionado.
- ¿Cómo determinar si un conjunto de casos de test es bueno? i.e. que detecte la mayor cantidad de defectos y que ningún conjunto de casos de test también lo encuentre.
- No existe una manera de determinar la bondad; usualmente el conjunto de casos de test es revisado por expertos.
- Esto requiere que los casos de test se especifiquen antes de realizar el testing: una razón clave para tener especificaciones de casos de test.

Siguiendo el proceso de inspección usual

Para ello es importante la justificación del test

El proceso de testing

Especificación de los casos de test

La especificación de los casos de test es esencialmente una tabla:

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

El proceso de testing
Especificación de los casos de test

- Para cada testeo, se desarrolla una especificación de casos de test que se revisa y se ejecuta.
- La preparación de la especificación de los casos de test es una tarea exigente y que demanda tiempo:
  - Se pueden utilizar criterios para casos de test.
  - Se pueden utilizar casos especiales y escenarios.
- Una vez especificados, la ejecución y la verificación del resultado se puede automatizar mediante scripts:
  - Es deseado si se necesita repetir el testing.
  - Se hace regularmente en grandes proyectos.

El proceso de testing

Ejecución de los casos de test y análisis

- La ejecución de los casos de test puede requerir de la escritura de “drivers” o “stubs”.
- También se requerirán módulos extras para preparar el entorno acorde a las condiciones establecidas en la especificación del caso de test.
- Algunos test pueden automatizarse, otros necesitan ser manuales:
  - En ambos casos se puede preparar un documento separado respecto del procedimiento de test.
- Se realizan reportes con resúmenes del test: reporta un resumen de los casos de test ejecutados, el esfuerzo, los defectos encontrados.
- El seguimiento y control del esfuerzo del testing es importante para asegurar que se invirtió el tiempo suficiente.
- El tiempo de computadora también es un indicador de cómo está procediendo el testing.

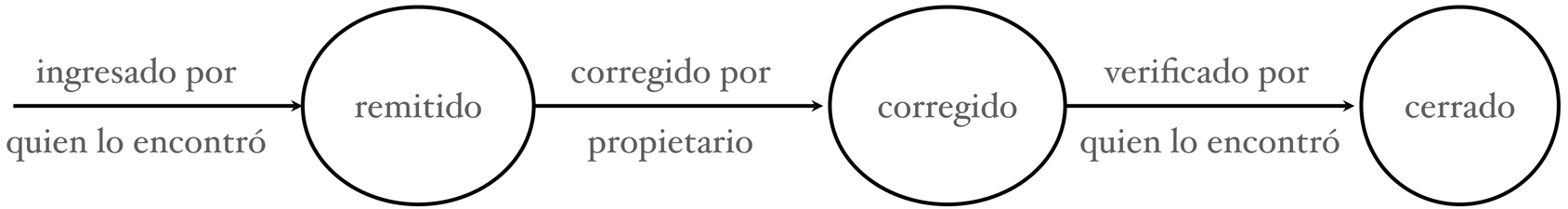
El proceso de testing
Registro de defectos y seguimiento

- Un software grande puede tener miles de defectos, encontrados por muchas personas distintas.
- Usualmente las personas que los corrigen no son las mismas que lo encontraron.
- Debido a este gran alcance, el registro y la corrección de los defectos no puede realizarse informalmente.
- Los defectos encontrados usualmente se registran en un sistema seguidor de defectos (“tracking”) que permite rastrearlos hasta que se “cierren”.
- El registro de defectos y su seguimiento es una de las mejores prácticas en la industria.

El proceso de testing
Registro de defectos y seguimiento

Un defecto en un proyecto de software tiene su propio ciclo de vida; por ejemplo:

- Alguien lo encuentra en algún momento y lo registra junto con toda la información relevante (defecto remitido).
- Se asigna la tarea de corrección; la persona hace el debugging y lo corrige (defecto corregido).
- El administrador o quien lo remitió verifica que el defecto fue efectivamente corregido (cerrado).



También son posibles otros ciclos de vida más elaborados

El proceso de testing
Registro de defectos y seguimiento

- Durante el ciclo de vida, se registra información sobre el defecto en las distintas etapas para ayudar al debugging y al análisis.
- Los defectos se categorizan generalmente en algunos tipos; los tipos de los defectos son registrados.
  - Una posible calificación es la denominada “Orthogonal defect classification”.
  - Algunas categorías: funcional, lógica, standards, asignación, interfaz de usuario, interfaz de componente, desempeño, documentación, etcétera.

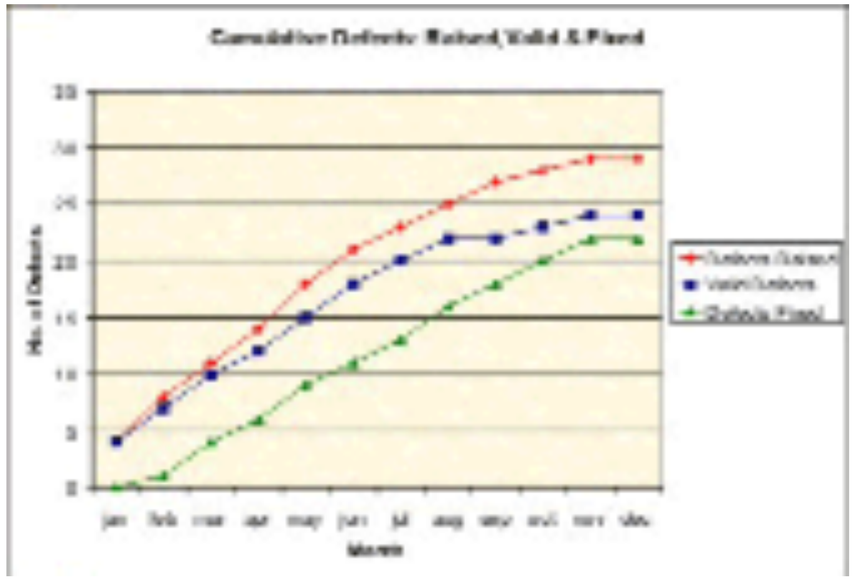
El proceso de testing
Registro de defectos y seguimiento

- También se registra la severidad del defecto en términos de su impacto en el software.
- La severidad es útil para priorizar la corrección.
- Una posible categorización:
  - Crítico: puede demorar el proceso; afecta a muchos usuarios.
  - Mayor: tiene mucho impacto pero posee soluciones provisorias; requiere de mucho esfuerzo para corregirlo, pero tiene menor impacto en el cronograma.
  - Menor: defecto aislado que se manifiesta raramente y que tiene poco impacto.
  - Cosmético: pequeños errores sin impacto en el funcionamiento correcto del sistema.

El proceso de testing

Registro de defectos y seguimiento

- Idealmente, todos los defectos deben cerrarse.
- Algunas veces, las organizaciones entregan software con defectos conocidos (ej.: no hay defectos críticos o mayores, y menores < X).
- Las organizaciones tienen estándares para determinar cuando un producto se puede entregar (o poner a la venta).
- El registro de defectos puede utilizarse para seguir la tendencia de como ocurren los arribos de los defectos y sus correcciones.



Testing

Lectura complementaria:

- Capítulo 10 Jalote
- Verification by Amy J. Ko https://faculty.washington.edu/ajko/books/cooperative-software-development/#/verification