FACULTAD DE INGENIERIA

Fundamentos de Ingeniería de Software

Pruebas de software: verificación y validación

Clase 1

María de las Nieves Freira

Universidad de la República

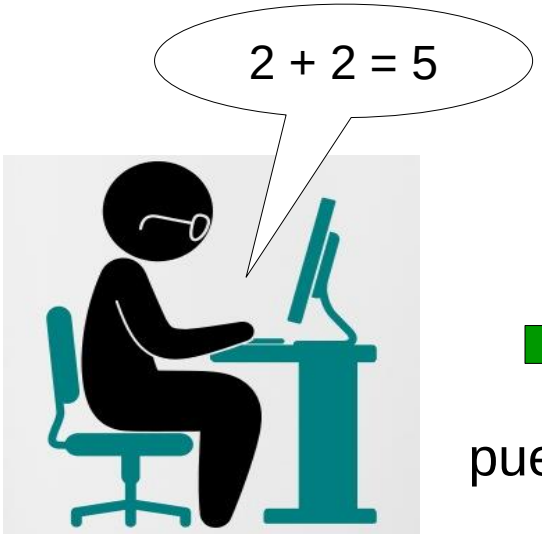
Temario

- Motivación
- Tipos y clasificación de defectos
- Proceso de V&V
- Pruebas
- Técnicas y comparación

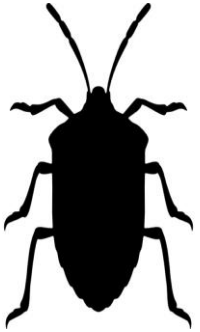
¿Por qué verificar el software?

- ¿Podemos construir software cero defecto?
  - ¿Por qué sí o por qué no?
- ¿Cuándo verificar durante el desarrollo de software?
  - ¿Cuál es el impacto económico?
- ¿Qué verificar (sub-productos)?
- ¿Cuánto verificar?
- ¿Cómo verificar?

Error, defecto y falla



puede generar



que puede generar

Un error humano

Un defecto (interno)

También llamada falta

Una falla (externa)

Fallas del software

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

Objetivos de V&V

- Descubrir defectos (para corregirlos)
  - Provocar fallas (una forma de detectar defectos)
  - Revisar los productos (otra forma de detectar defectos)
- Evaluar la calidad de los productos
  - El probar o revisar el software da una idea de su calidad
- Demostrar que el software cumple con los requisitos.

Prueba de función

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

Identificación y corrección de defectos

- Identificación de defectos
  - Es el proceso de determinar qué defecto o defectos causaron la falla
- Corrección de defectos
  - Es el proceso de cambiar el sistema para remover los defectos

Definición de V&V (Sommerville)

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

Ejemplo

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

- Quiero detectar defectos probando (testeando) el programa.
- Posibles casos a probar:
  - lado1 = 0, lado2 = 1, lado3 = 0. **Resultado** = error.
  - lado1 = 2, lado2 = 2, lado3 = 3. **Resultado** = isósceles.
- Estos son *casos de prueba.*

Ejemplo

Comparo el resultado esperado con el obtenido

- Si son distintos, probablemente haya fallado el programa.

Intuitivamente ¿qué otros casos sería bueno probar?

- lado1 = 2, lado2 = 3, lado3 = 4. **Resultado** = escaleno.
- lado1 = 2, lado2 = 2, lado3 = 2. **Resultado** = equilátero.

¿Por qué estos casos?

- Al menos probé un caso para cada respuesta posible del programa (error, escaleno, isósceles, equilátero)
- Más adelante veremos técnicas para seleccionar casos interesantes

Ejemplo – más casos de prueba

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

Tipos de defectos en el software

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

Tipos de defectos en el software

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

Tipos de defectos en el software

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

Tipos de defectos en el software

- De estrés o sobrecarga
  - Exceder el tamaño máximo de un área de almacenamiento intermedio
  - Ejemplos
    - El sistema funciona bien con 100 usuarios, pero no con 110
    - Sistema que funciona bien al principio del día y se va degradando paulatinamente el desempeño hasta ser espantoso al caer la tarde. Falta: había tareas que no liberaban memoria
- De capacidad o de borde
  - Más de lo que el sistema puede manejar
  - Ejemplos
    - El sistema funciona bien con importes <1000000
    - Año 2000: sistema trata bien fechas hasta el 31/12/99

Tipos de defectos en el software

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

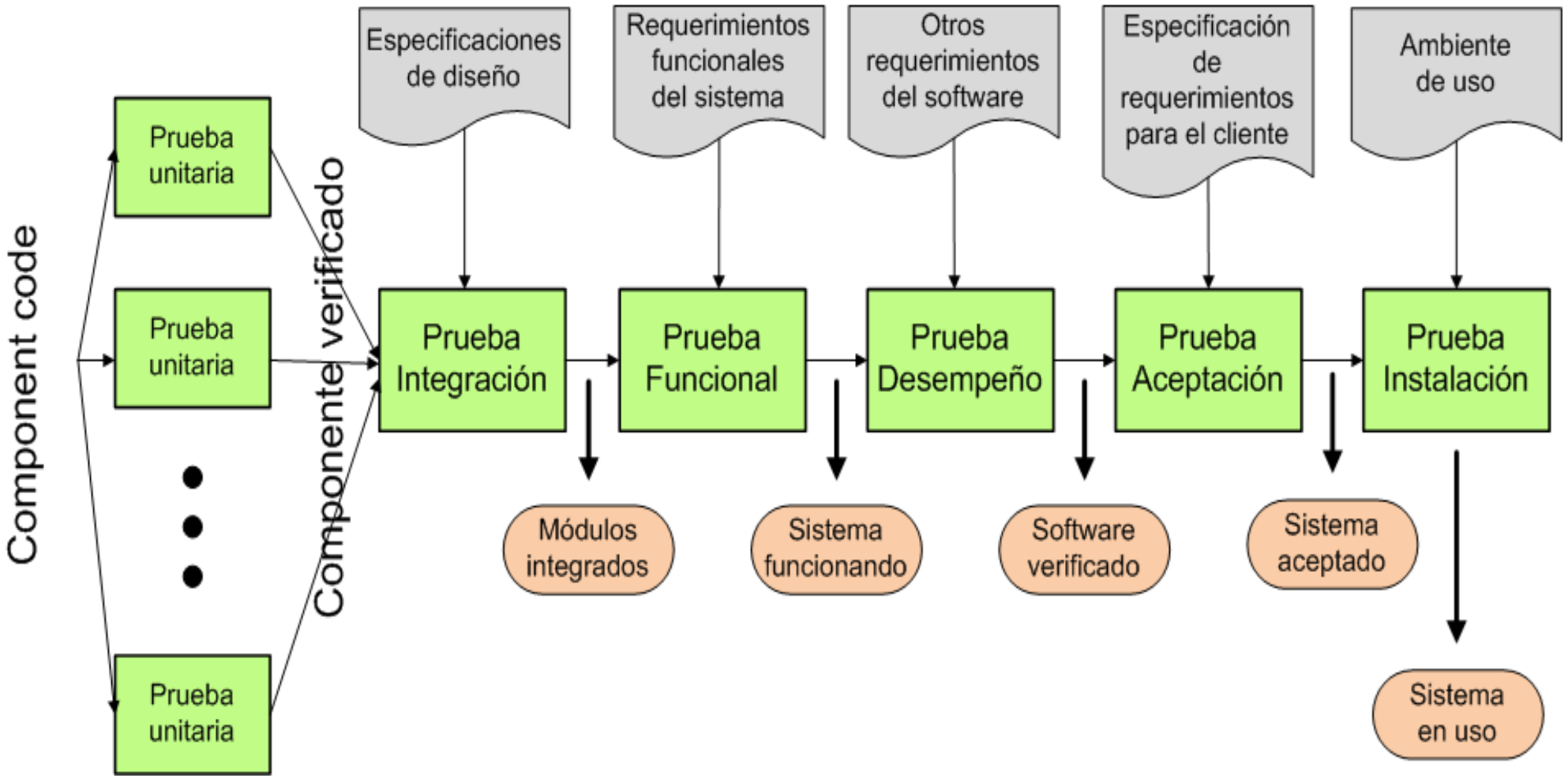
Tipos de defectos en el software

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

Clasificación de defectos

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

Proceso de V&V



Pruebas en el proceso

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

Pruebas en el proceso

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

¿Quién verifica?

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

¿Quién verifica?

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

Conflictos personales

- Conflictos posibles entre
  - Equipo de verificación (encontrar defectos)
  - Equipo de desarrollo (mostrar las bondades de su obra)
- Soluciones:
  - Trabajo en equipo (roles distintos, igual objetivo)
  - Se evalúa al producto (no la persona)
  - Voluntad de mejora (personal y del equipo)

Técnicas de verificación unitaria

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

Técnicas estáticas

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

Algunas definiciones

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

Visión de los objetos a probar

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

Conceptos básicos

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

La prueba (test) demuestra la presencia de faltas y nunca su ausencia (Dijkstra)

- Es necesario elegir un subconjunto de las entradas del programa para testear
  - Conjunto de prueba (test set)

Conceptos básicos

- El test debe ayudar a localizar faltas y no solo a detectar su presencia

- El test debe ser repetible
  - En programas concurrentes es difícil de lograr

- ¿Cómo se hacen las pruebas?
  - Se ejecuta el caso de prueba y se compara el resultado esperado con el obtenido.

Entonces... ¿qué estamos buscando?

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

Caja negra

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

Partición en clases de equivalencia

- Técnica de pruebas de caja negra.
- Se utiliza en conjunto con análisis de valores límite.
- Se divide el dominio de entrada del programa/componente/sistema a probar en clases de equivalencia.
  - La suposición es que cada elemento de dicha clase se comporta de «igual» forma en el programa.
  - Entonces, basta con tomar un elemento de cada clase de equivalencia (y sus límites).

Partición en clases de equivalencia
Ejecución de la técnica

1. Dividir el dominio de entrada en clases de equivalencia considerando
   - El dominio de entrada en sí mismo
   - El dominio de salida
2. Definir los casos de prueba para cada clase de equivalencia tomando al menos un representante de la clase y valores límite de ella
   - Escribir un nuevo caso de prueba que cubra tantas clases de equivalencia válidas (no cubiertas) como sea posible.
   - Escribir un caso de prueba para cubrir una y solo una clase de equivalencia para cada clase de equivalencia inválida (evita cubrimiento de errores por otro error).

Partición en clases de equivalencia

- Identificación de las clases de equivalencia.
- *Ejemplos:*
  - «La numeración es de 1 a 999»:
    - clase válida: 1 <= núm. <= 999,
    - 2 clases inválidas: núm. < 1 y núm. > 999
  - «El primer carácter debe ser una letra»:
    - clase válida: el primer carácter es una letra
    - clase inválida: el primer carácter no es una letra
- Si se cree que ciertos elementos de una clase de equivalencia no son tratados de forma idéntica por el programa, dividir la clase de eq. en clases de eq. menores

Valores límite

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

Valores límite (ejemplos)

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

USAR EL INGENIO PARA ENCONTRAR CONDICIONES LÍMITE

Ejercicio I

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

Ejercicio I (solución)

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

Ejercicio I (solución)

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

Ejercicio 2

- Un programa/método recibe un entero representando la edad de una persona y devuelve true, si es mayor de edad, y false si es menor de edad.
- Identificar clases de equivalencia.

Ejercicio 2 (solución)

- ¿Qué es ser mayor de edad?
  - Problema en la especificación que se puede detectar al momento de intentar generar casos de prueba (en otros momentos también).
- Supongamos que con 18 años o más se es mayor de edad
  - Dividir en clases válidas e inválidas (y considerar límites)
    - Alfanuméricos (si se permite su ingreso)
    - Edades negativas
    - Edades positivas
    - Menores que 18
    - Iguales o mayores a 18
    - ¿Mayores que 140?

Ejercicio 3

- Un programa recibe un vector que contiene números reales entre 0 y 1.000 como elementos. El programa devuelve el promedio de estos números reales. La cantidad de elementos del vector debe ser mayor o igual a 4 y menor o igual a 8.
  - ¿Cuáles son los dominios de entrada?
  - ¿Cómo se partirían en clases de equivalencia?

Ejercicio 4

- Un programa recibe tres números enteros mayores que cero
- Realice una partición en clases de equivalencia

Encontrar el defecto

- Se ha provocado una falla mediante una prueba
- Ahora hay que encontrar el defecto
  - Revisar el código o
  - Hacer *debug* del código
  - Basándose en los datos de entrada del caso de prueba que falló
- Luego de encontrar el defecto
  - Corregir
  - Realizar pruebas de regresión

Encontrar el efecto y corregirlo

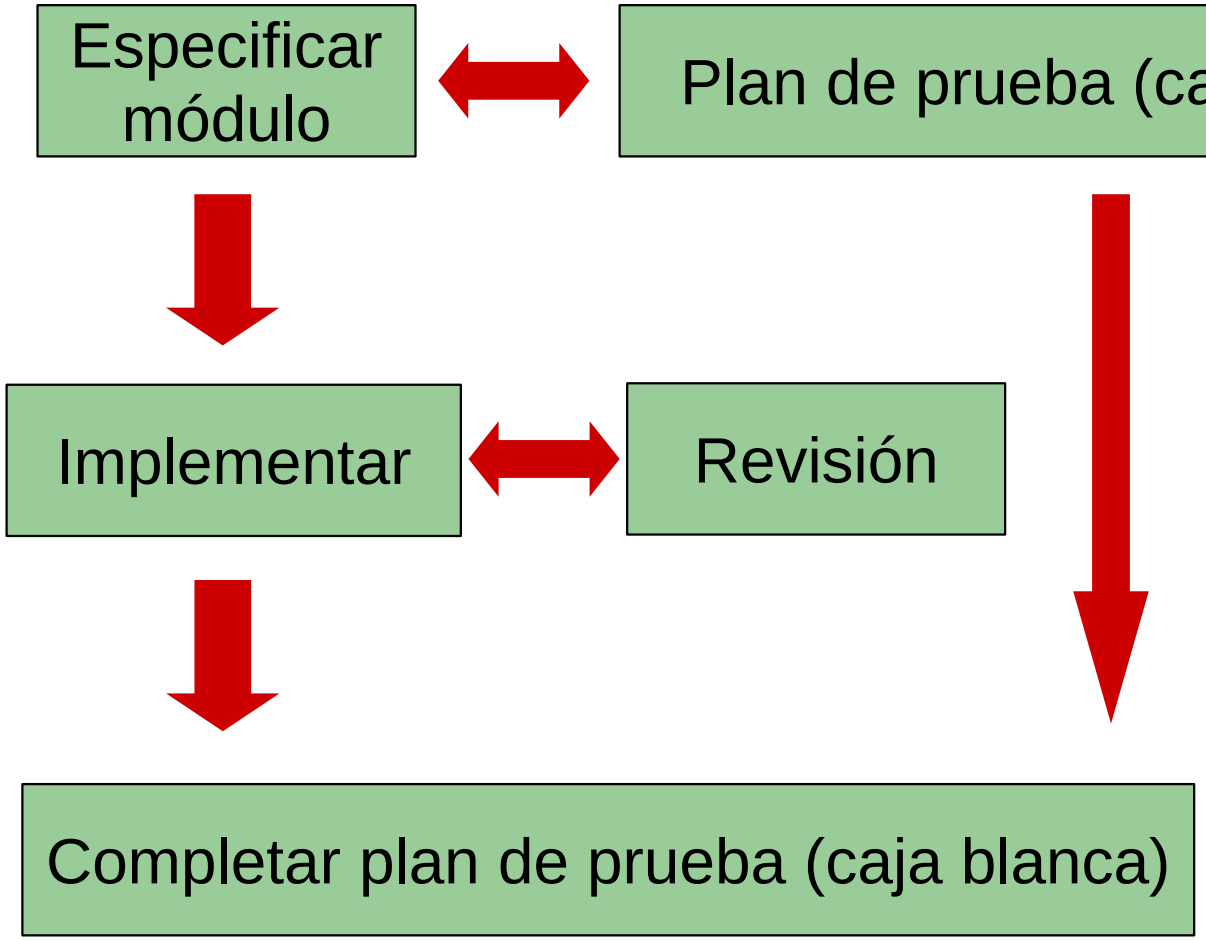
```
public class Operaciones {
    public int[] merge( int[ ] a, int [ ] b) {
        int i= 0;
        int j = 0;
        int k = 0;
        int c[] = new int[a.length + b.length];
        while( i < a.length && j < b.length ){
            if( a[i] < b[j]){
                c[k]=a[i];
                k++;
                i++;
            }else{
                c[k]=a[j];
                k++;
                j++;
            }
        }
        for (int iter=i;iter<a.length;iter++){
            c[k]=a[iter];
            k++;
        }
        for (int iter=j;iter<b.length;iter++){
            c[k]=b[iter];
            k++;
        }
        return c;
    }
}
```

Error al hacer la asignación
Se cambia
c[k] = a[j]; por c[k]= b[j];

Sugerencias

- Revisar el código
  - Es cuando se detecta la mayor cantidad de defectos
  - Es bueno detectarlos tempranamente
- Construir casos de caja negra
  - La funcionalidad del software es lo que hay que asegurar
- Ver el cubrimiento alcanzado
  - Conocer qué tan buenos son mis casos
- Asegurar cubrimiento
  - Complementar con casos faltantes para asegurar cierto criterio de caja blanca

Proceso para un módulo



Sugerencia:

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

Más sugerencias

- ¿Qué hacer cuando queda poco tiempo?
  - Revisar, construir casos de caja negra, luego ejecutar, corregir (varias veces), ejecutar regresión (varias veces), ver cubrimiento alcanzado, construir nuevos casos, ejecutar, corregir (varias veces), ejecutar regresión (varias veces), ver cubrimiento alcanzado…
  - **Lleva mucho tiempo, mejor sólo codificar.**
  - ¿Cuál es el costo del retrabajo?
    - No se olviden de que los componentes que no verifico hoy son los que van a fallar mañana. Incluso, van a fallar al momento de ejecutar otro componente que lo «usa» y va a ser difícil y muy costoso ubicar el defecto.

Comparación de las técnicas

Estáticas (análisis)

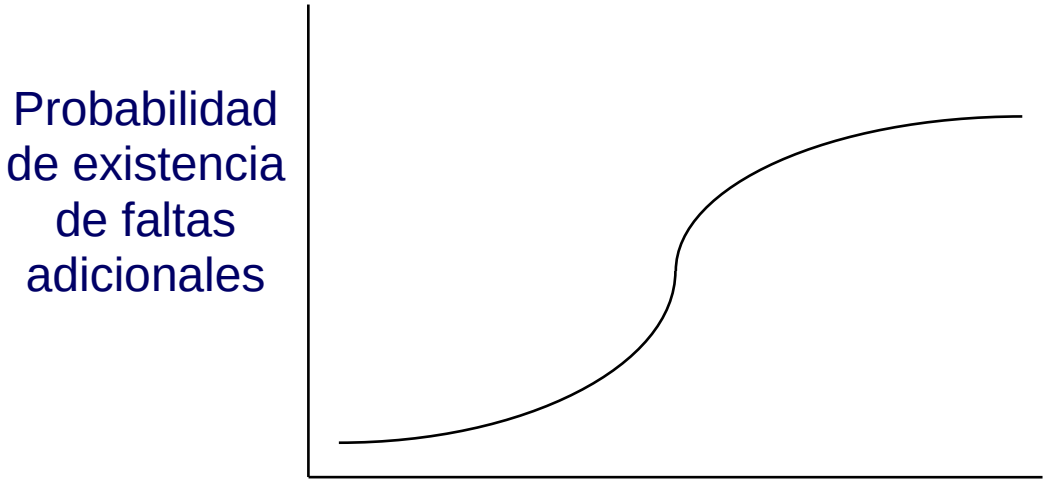
- ✓ Efectivas en la detección temprana de defectos
- ✓ Sirven para verificar cualquier producto (requisitos, diseño, código, casos de prueba, etc.)
- ✓ Conclusiones de validez general
- ✗ Sujeto a los errores de nuestro razonamiento
- ✗ Normalmente basadas en un modelo del producto y no en el producto
- ✗ No se usan para validación (solo verificación)
- ✗ Dependen de la especificación
- ✗ No consideran el hardware o el software de base

Comparación de las técnicas

Dinámicas (pruebas)

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

Más faltas… más por encontrar



Estudio de Myers en 1979

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

Criterios de terminación

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

Criterios de terminación

- Criterios usados (pero contraproducentes):
- Terminar la prueba cuando:

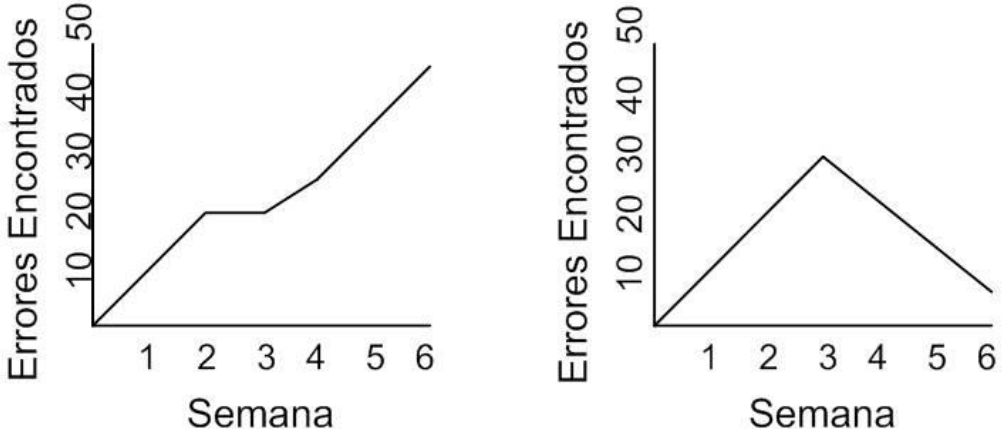
1. El tiempo establecido para esta ha expirado.
   - Este criterio es inútil, puesto que puede ser satisfecho sin haber hecho absolutamente nada. No mide la calidad de las pruebas realizadas.
2. Todos los casos de prueba se han ejecutado sin detectar fallas (es decir, cuando todos los casos de prueba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

Categorías de criterios (Myers)

- Por criterios del diseño de casos de prueba
  - Se definen criterios de terminación según el cumplimiento de un criterio de cubrimiento y que todos esos casos de prueba se ejecuten sin provocar fallas.
- Detención basada en la cantidad de defectos
  - Ejemplos:
    - Detectar 3 defectos por módulo y 70 en la prueba funcional.
    - Tener detectado el 90 % de los defectos del sistema.
  - Necesito estimar la cantidad total de defectos. De esta manera puedo saber cuándo parar la prueba.

Categorías de criterios (Myers)

- Basada en la cantidad de fallas detectadas por unidad de tiempo, durante las pruebas
  - Se registra la cantidad de fallas que se detectan durante la prueba por unidad de tiempo. Luego se grafican estos valores.
  - Examinando la forma de la curva se puede determinar cuándo detener las pruebas.



Una combinación

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

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

Próxima clase:

Pruebas de software

(continuación)