Validación y Verificación de Software

Preliminares: Calidad | Validación y Verificación | Especificaciones y V&V | Análisis estático y dinámico | Inspecciones | Testing | Procesos de desarrollo | Terminología |Clasificación

Calidad en el Desarrollo de Software

Uno de los objetivos principales en el desarrollo de software es conseguir productos de alta calidad.

La calidad involucra confiabilidad, mantenibilidad, interoperabilidad, etc.

- La confiabilidad es uno de los atributos de calidad más importantes.

Una de las medidas de calidad mejor establecidas es la cantidad de defectos por líneas de código.

- Mayor cantidad de defectos hace al software menos confiable (de menor calidad).

Validación y Verificación

Las tareas de validación y verificación ayudan a mostrar que el software cubre las expectativas para las cuales fue construido.

Ayuda a garantizar calidad (en particular, confiabilidad).

Esto no significa garantizar que el software será completamente libre de defectos.

Debe ser “lo suficientemente bueno” para el uso que se le pretende dar (y el uso determina el grado de confianza que se requiere del software)

Validación vs. Verificación

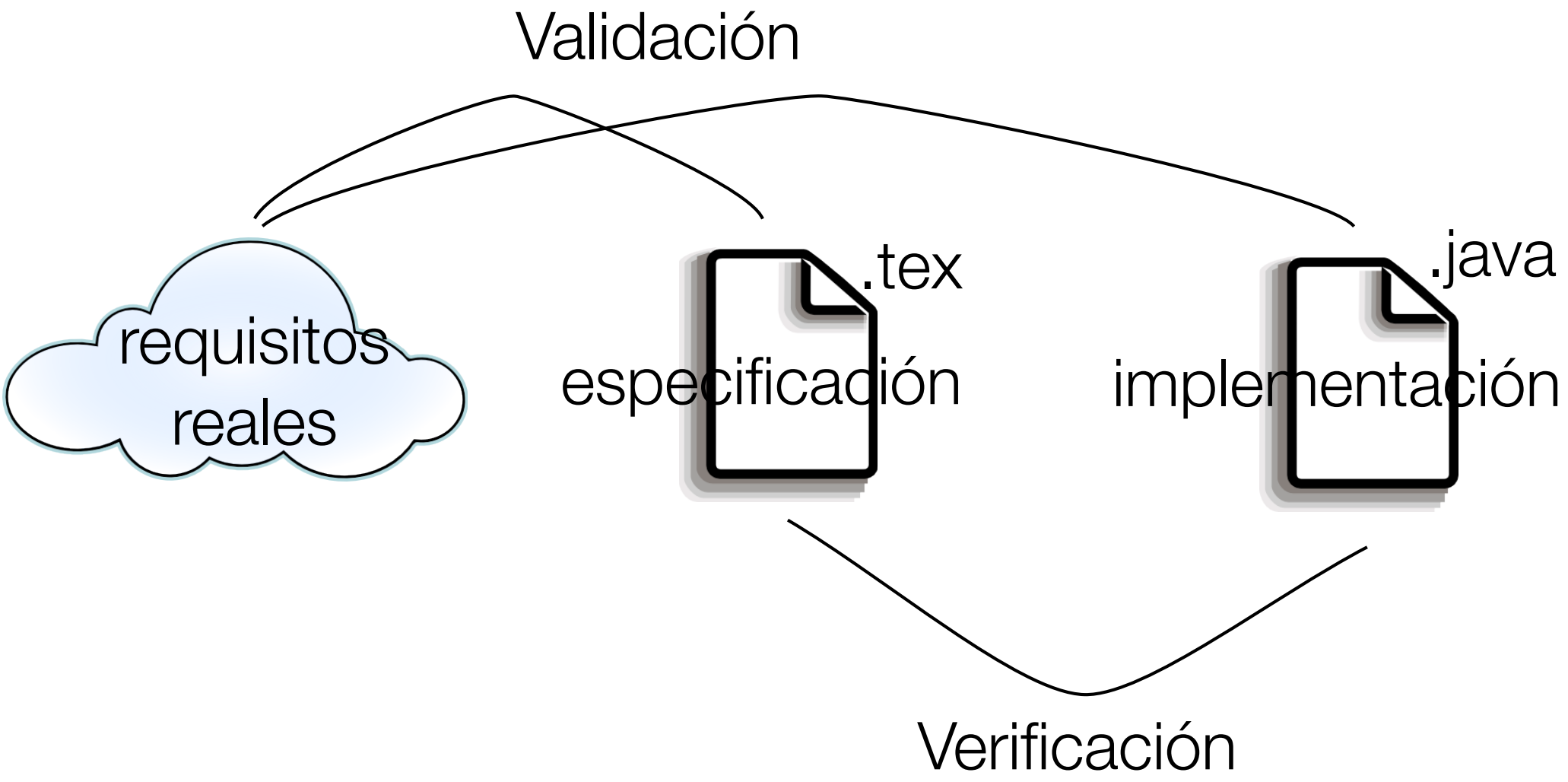
Verificación: el software debería realizar lo que su especificación indica:

¿construimos el producto correctamente?

Validación: El software debería hacer lo que el usuario requiere de él:

¿construimos el producto correcto?

Validación vs. Verificación



Especificaciones en V&V

Las especificaciones son fundamentales como mecanismo para poder razonar sobre la corrección de un programa, es decir, para las actividades de Validación y Verificación

La especificación de un programa es una descripción de qué es lo que se supone que el programa debe hacer

Existen muchas formas diferentes de expresar especificaciones de programas

- una particularmente adecuada es la basada en aserciones de corrección

Contratos como Especificación de Software

El Diseño por Contratos es una metodología de desarrollo de software (orientado a objetos) basado en la posibilidad de especificar las funcionalidades asociadas a las partes de un sistema (clases, pero también unidades de menor granularidad como las rutinas).

Estas especificaciones deben permitir, de manera precisa, indicar cuál es la relación correcta entre una clase y los clientes de la misma, y hacen uso intensivo de aserciones, y elementos de especificaciones formales.

Las aserciones forman parte de los denominados contratos, y constituyen la base esencial del Diseño por Contratos.

Aserciones de Corrección

Las aserciones de corrección de programas, en particular para programas imperativos, suelen venir en el siguiente formato, de pre- y post-condición:

$$\{P\}\ A\ \{Q\}$$

donde A es un programa, P y Q son fórmulas (de estado). El significado de una aserción como la anterior es:

“Siempre que se inicie la ejecución del programa A en un estado que satisfaga P, el programa termina en un estado que satisface Q.”

Las aserciones de corrección son una notación matemática, y generalmente no son parte de nuestro lenguaje para construir software.

Contratos, derechos y obligaciones

Las especificaciones de rutinas mediante pre- y post-condiciones establecen derechos y obligaciones para cliente y proveedor:

	Precondición	Postcondición
Cliente	debe asegurarse que se cumple (antes de la ejecución)	puede suponer que se cumple (luego de la ejecución)
Proveedor	puede suponer que se cumple (antes de la ejecución)	debe asegurarse que se cumple (luego de la ejecución)

Ejemplo

```
/**
@pre. A != null, 0 <= i <= j < |A|
@post. - A no cambia.
       - Si |A|=0, resultado = (0,0), sino resultado.fst() es el mínimo de A en el
         segmento [i..j] y resultado.snd() es el máximo de A en el segmento [i..j]
*/
public static IntPair minMaxArray(int[] A, int i, int j) {

   if (i>j) {
       IntPair res = new IntPair(0,0);
       return res;
   }
   else {
       IntPair res_sub = new IntPair(A[i],A[i]);
       int k = i;
       while (k<=j) {
           if (A[k]<res_sub.fst()) {
             res_sub.changeFst(A[k]);
         }
           if (A[k]>res_sub.snd()) {
             res_sub.changeSnd(A[k]);
         }
         k++;
       }
       return res_sub;
  }
}
```

Invariantes de Clase

Las precondiciones y postcondiciones establecen propiedades de rutinas particulares. Existe en general la necesidad de expresar otras propiedades globales de las instancias de una clase. Esto se logra a través de los invariantes de clase (también llamados invariantes de representación).

Los invariantes de clase forman parte del contrato de una clase. Establecen propiedades que deben mantenerse a lo largo de la ejecución de una clase. Es decir,

todos los constructores deben establecer, al finalizar, la propiedad

todas las rutinas de la clase deben preservar la propiedad

Los Métodos y el Invariante de Representación

Los métodos de una clase deben ser acompañados por especificaciones en términos de pre- y post-condiciones.

El contrato de una clase está compuesto por:

las pre- y post-condiciones de cada uno de los métodos de la clase,

el invariante de representación de la clase.

Se dice que una clase respeta su contrato si y sólo si:

Los constructores de la clase garantizan la validez del invariante de representación en su terminación.

Cada método m satisface que:

$$\{\texttt{pre-m} \wedge \texttt{repOK()}\}$$
$$\texttt{m}$$
$$\{\texttt{post-m} \wedge \texttt{repOK()}\}$$

Ejemplo

```
public class Fecha {

 private int dia;
 private int mes;
 private int anho;

 ...

 /*
  Invariante:
  - mesValido: 1<=mes y mes<=12
  - diaValido: 1<=dia y dia<=31 y
  (si (mes=4 o mes=6 o mes=9 o mes=11) entonces dia<=30) y
  (si mes=2 entonces dia<=29)
  - anhoValido: 1900<=anho
  */

}
```

en realidad es un invariante aproximado
Se puede escribir algo más preciso
controlando bisiestos!

Violaciones de Aserciones

Las violaciones de aserciones nos llevan a una de las reglas fundamentales del diseño por contratos

La violación en tiempo de ejecución de una aserción de corrección es la manifestación de un bug en el software.

De acuerdo a los derechos y obligaciones enunciados anteriormente, la violación de una precondición es un error del cliente, mientras que la violación de una post-condición es un error del proveedor.

Esto permite identificar responsables de bugs reflejados como violaciones a contratos.

El proceso de validación y verificación

V&V debería aplicarse en cada instancia del proceso de desarrollo.

Tiene dos objetivos principales:

- descubrir defectos en el sistema, y
- medir si el sistema es “usable” en situaciones de operación del mismo.

Una forma de realizar tareas de V&V es a través de análisis (de programas, modelos, especificaciones, documentos, etc.).

- En particular para código, tenemos análisis estático y análisis dinámico.

Análisis dinámico

Analiza las propiedades de programas mediante su ejecución:

- extrae información de las corridas de programas,
- es sumamente “preciso”, pero intrínsecamente incompleto:
  - testing,
  - profiling de consumo de recursos (e.g., memoria),
  - chequeo en tiempo de ejecución de propiedades de programas.

Análisis estático

Se infieren propiedades de programas mediante el análisis del texto del mismo:

generalmente es impreciso:

inspección manual,

chequeo de tipos (en compilación).

Inspección manual

Involucra personas que examinan el código fuente, para intentar descubrir defectos, y otros problemas de calidad “internos”: (e.g., código no apropiadamente documentado, mal estructurado, que no respeta estándares, etc.)

No requiere la ejecución del sistema (incluso se aplica a otras representaciones del sistema además del código: diseño, por ejemplo.)

Técnica muy efectiva para descubrir defectos, aunque difícilmente exhaustiva.

Testing

Esencialmente, consiste en comprobar el comportamiento de los programas en un conjunto de situaciones particulares (e.g., para algunas entradas particulares).

> Puede revelar la presencia de errores pero rara vez puede garantizar su ausencia.

Es la técnica de verificación funcional por excelencia.

Debe usarse en combinación con otras técnicas de V&V estáticas (e.g., inspección de código) para dar mejores resultados.

Debe realizarse de manera planificada.

Testing en procesos de desarrollo clásicos

Los modelos de proceso de desarrollo clásicos involucran generalmente las siguientes actividades:

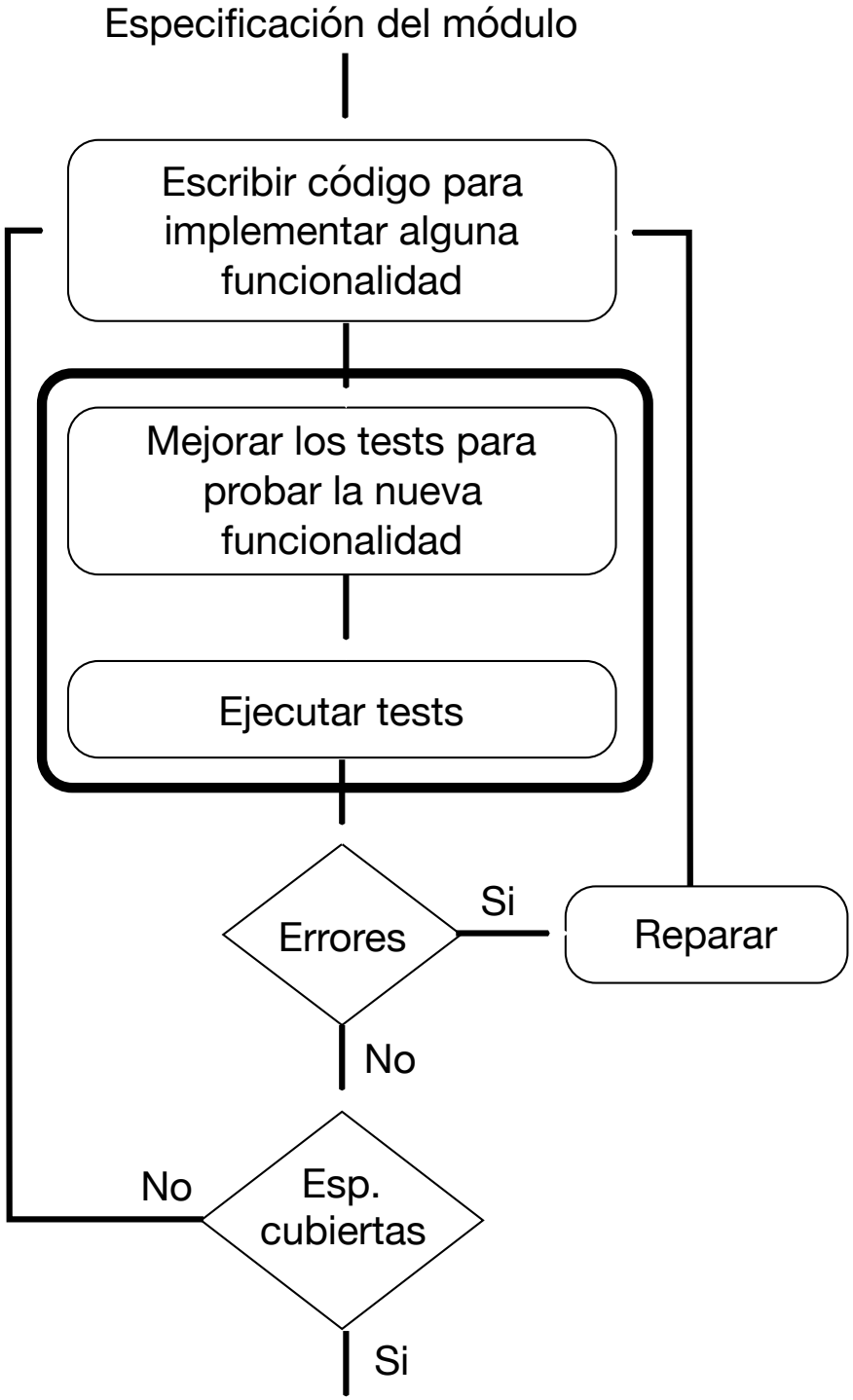
- Fase de requisitos
- Fase de especificación
- Fase de planeamiento
- Fase de diseño
- Fase de implementación
- Integración y Testing
- Mantenimiento

Testing es una actividad realizada en este caso posteriormente a la implementación.

Testing en procesos de desarrollo ágiles

En los modelos de proceso de desarrollo “ágiles”, el testing está presente en varias fases del proceso de desarrollo, principalmente como parte de la codificación.

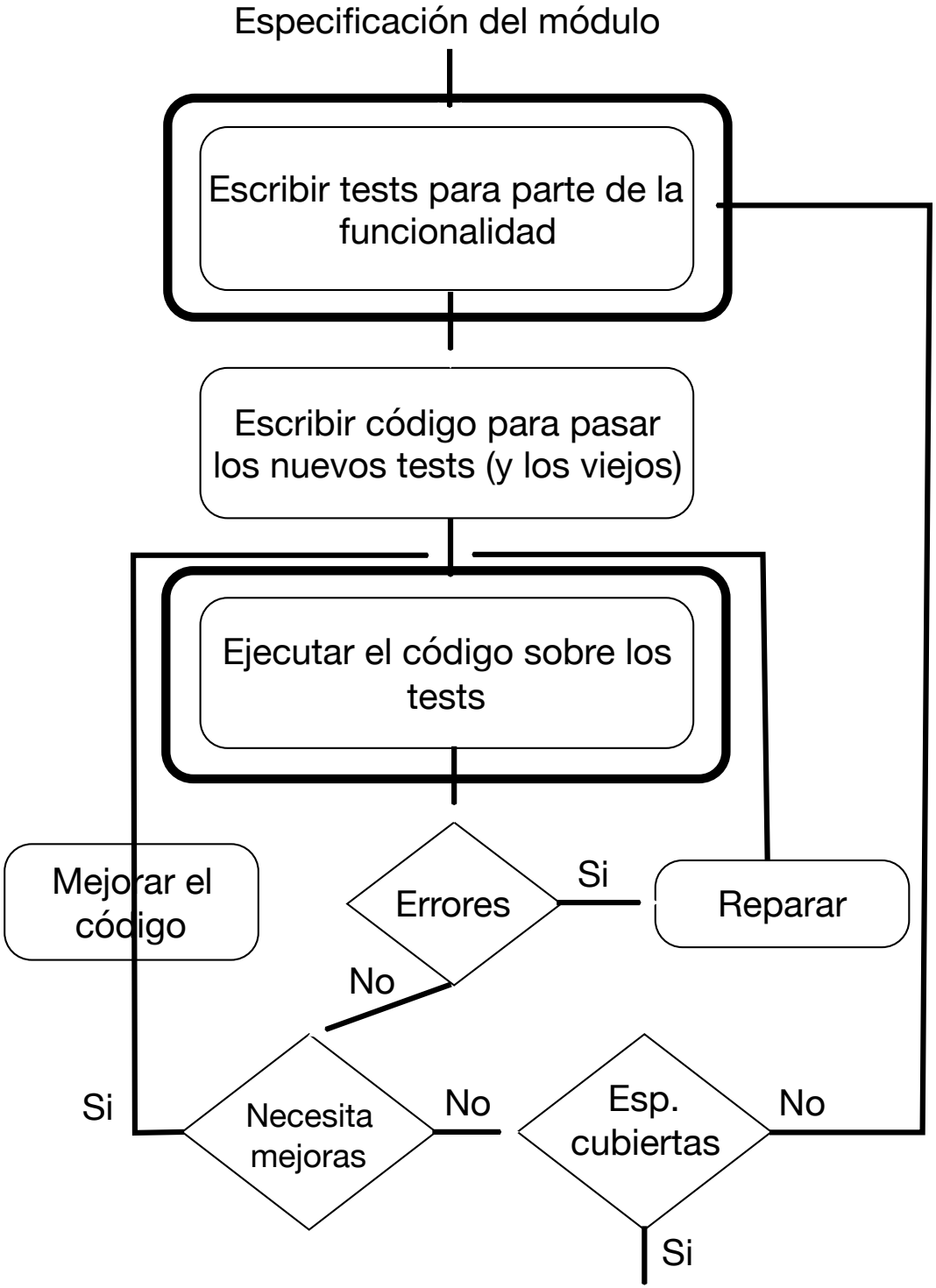
En el desarrollo de software guiado por funcionalidades, el testing se realiza como parte de la implementación de cada funcionalidad.



Testing como especificaciones

Algunos procesos “ágiles” ponen un énfasis aún mayor en testing.

En Test Driven Development (TDD), los tests se escriben antes de la codificación, y sirven como especificación de funcionalidades, y criterio de aceptación de código



Testing: terminología básica

Defecto (Fault): Línea(s) de código defectuosas en el software

error: Un estado interno incorrecto que es la manifestación de algún defecto.

Falla (failure): comportamiento incorrecto externo con respecto a los requerimientos o alguna otra descripción del comportamiento esperado.

Un ejemplo concreto

```
public static int numZero (int [ ] arr)
{  // Effects: If arr is null throw NullPointerException
   // else return the number of occurrences of 0 in arr
   int count = 0;
   for (int i = 1; i < arr.length; i++)
   {
      if (arr [ i ] == 0)
      {
         count++;
      }
   }
   return count;
}
```

Defecto: Debería comenzar la búsqueda en 0, no 1

Test 1
[2, 7, 0]
Esperado: 1
Obtenido: 1

Error: i es 1, no 0, en la primera iteración
Falla: ninguna

Test 2
[0, 2, 7]
Esperado: 1
Obtenido: 0

Error: i es 1, no 0
El error se propaga a la variable count
Falla: count es 0 al retornar

Ejemplo

```
public int findLast (int[] x, int y) {
//Effects: If x==null throw NullPointerException
//   else return the index of the last element
//   in x that equals y.
//   If no such element exists, return -1
   for (int i=x.length-1; i > 0; i--)
   {
     if (x[i] == y)
     {
       return i;
     }
   }
   return -1;
}
```

Cuál es el defecto en findLast?

Existe alguna entrada que no ejecute el defecto?

Y alguna que ejecute el defecto sin producir una falla?

Y una entrada que sí produzca una falla?

*del libro Introduction to Software Testing, de Amman y Offutt

Ejemplo

```
public int findLast (int[] x, int y) {
//Effects: If x==null throw NullPointerException
//   else return the index of the last element
//   in x that equals y.
//   If no such element exists, return -1
   for (int i=x.length-1; i > 0; i--)
   {
     if (x[i] == y)
     {
       return i;
     }
   }
   return -1;
}
```

Cuál es el defecto en findLast?

i > 0, Solución: i>=0

Existe alguna entrada que no ejecute la defecto?

x = null, y = 0

Y alguna que ejecute el defecto sin producir una falla?

x = [1,2,3], y = 2

resultado 2, esperado 2

Y una entrada que sí produzca una falla?

x = [1,2,3], y = 1

resultado -1, esperado 0

*del libro Introduction to Software Testing, de Amman y Offutt

Más definiciones

Testing: Evaluar el software observando su ejecución

Debugging: El proceso de buscar un defecto dada una falla

Un test es una prueba de software, compuesta usualmente de:

- Entradas
- Código a ejecutar (testear)
- Descripción de los resultados esperados

Test existoso: Ejecución de un test no revela una falla

Test fallido: Ejecución de un test descubre una falla

Testing: clasificaciones básicas

Existen diferentes tipos de testing, de acuerdo a las características de sus partes. Algunos de estos tipos son los siguientes:

Sistema: el bloque a testear es todo el sistema.

Integración: el bloque a testear es la composición de varios módulos, y la condición de aceptación corresponde a propiedades de la ejecución combinada de los módulos.

Regresión: la condición de aceptación es preservar el comportamiento de versiones anteriores del software.

Diferencial: la condición de aceptación es mantener un comportamiento similar a otro software con el mismo propósito que el testeado.

Lectura recomendada

Capítulo 1 del libro Introduction to Software Testing de Ammann y Offutt