15. Pruebas orientadas a objetos

Índice

- Referencias
- Introducción
- Estrategias de pruebas orientadas a objetos
  - Introducción
  - Pruebas de unidad.
  - Pruebas de integración.
  - Pruebas de sistema.

Índice

- Depuración
  - Introducción
  - Proceso
- Pruebas de caja blanca
  - Introducción
  - Prueba del camino básico
  - Prueba de la estructura de control

Índice

- Pruebas de caja negra
  - Introducción
  - Basadas en grafos
  - Partición equivalente
  - Análisis valores límite
  - Tabla ortogonal
  - Comparación

Índice

- Automatización de pruebas
- JUnit 5
- JUnit 4
- Conclusiones

Referencias

- Pressman, R.S. *Ingeniería del Software. Un enfoque práctico*. McGraw-Hill, 2005
- Sommerville I., *Ingeniería del software. 7ª edición*. Pearson Educación, 2005
- Vogella GmbH, *Unit Testing with JUnit-Tutorial* (2016), http://www.vogella.com/tutorials/JUnit/article.html

Introducción

- El objetivo de las pruebas es encontrar el mayor número posible de errores con una cantidad razonable de esfuerzo, aplicado sobre un lapso de tiempo realista
- La construcción de software OO comienza con la creación de los modelos de análisis y diseño

Introducción

- No vamos a dar mecanismos concretos para revisar estos modelos, ya que dichos mecanismos podrían variar en función de la notación utilizada (sobre todo en análisis)
- Lo que sí vamos a pedir es una *consistencia* entre el modelo de análisis y el modelo de diseño

Introducción

- Por lo demás, ambos modelos deben ser incluidos como ECSs
- Por lo tanto, en algún momento del proceso de desarrollo, pasarán sus RTFs correspondientes
- Evidentemente, cuanto antes descubramos errores, menos costosa será su reparación

Estrategias de pruebas OO Introducción

- La estrategia clásica para la prueba de software comienza con *probar lo pequeño* y continúa hacia fuera *probando lo grande*
- En términos de IS, las pruebas son:
  - De unidad.
  - De integración.
  - De validación del sistema.

Estrategias de pruebas OO
Introducción

- Las *pruebas de unidad* se centran en las unidades de programa compilables más pequeñas
- Después, estas unidades se *integran* en una estructura de programa y se prueban en conjunto
- Finalmente se prueba el *sistema*

Estrategias de pruebas OO Introducción

- Veamos cómo se traduce todo esto en el contexto de pruebas OO

Estrategias de pruebas OO
Pruebas de unidad

- En el contexto OO la unidad más pequeña es la clase
- De cada clase hay que probar cada una de sus operaciones
- Debido al polimorfismo, las operaciones pueden variar en el contexto de una jerarquía de clase

Estrategias de pruebas OO
Pruebas de unidad

- Para cada operación hay que comprobar:
  - Interfaz de la operación.
  - Estructuras de datos locales.
  - Condiciones límite.
  - Caminos independientes.
  - Caminos de manejo de errores.

Estrategias de pruebas OO
Pruebas de integración

- Existen dos estrategias diferentes para las pruebas de integración de sistemas OO:
  - Pruebas basadas en hilos.
  - Pruebas basadas en usos.
- Las pruebas *basadas en hilos* integran el conjunto de clases requeridas, para responder una entrada o suceso del sistema

Estrategias de pruebas OO
Pruebas de integración

- Cada *hilo* se integra y prueba individualmente
- En este contexto *hilo* es encadenamiento de mensajes, al estilo diagrama de secuencia, no es *thread*
- Además, se aplican pruebas de regresión

Estrategias de pruebas OO
Pruebas de integración

- La *prueba de regresión* consiste en volver a ejecutar un subconjunto de pruebas que se han llevado a cabo anteriormente para asegurarse que los cambios que se derivan al integrar módulos probados independientemente no propaguen efectos colaterales no deseados

Estrategias de pruebas OO
Pruebas de integración

- Las pruebas *basadas en el uso*, comienzan la construcción del sistema probando aquellas clases (*clases independientes*), que utilizan muy pocas o ninguna clases servidoras
- Después de probar las clases independientes se continúa una secuencia de pruebas por capas de las clases dependientes hasta que se construye el sistema completo.

Estrategias de pruebas OO
Pruebas de sistema

- Al nivel de sistema, los detalles de conexiones de clases desaparecen
- Las principales pruebas del sistema son las de *validación*, y se centran en las acciones visibles al usuario y salidas reconocibles desde el sistema

Estrategias de pruebas OO
Pruebas de sistema

- Para llevar a cabo las pruebas de validación, deben utilizarse los casos de uso
- Se llevan a cabo pruebas de *caja negra*
- Es decir, nos interesa el comportamiento observable del sistema, no la organización interna del mismo

Estrategias de pruebas OO
Pruebas de sistema

- Con respecto a las pruebas de validación:
  - La prueba se concentra en las acciones visibles para el usuario y en la salida del sistema que este puede reconocer
  - Como ya vimos en el Tema 9, consisten en comprobar si el sistema cumple con los requisitos del cliente y usuario

Estrategias de pruebas OO
Pruebas de sistema

- En la práctica es muy difícil que el desarrollador prevea como utilizará el usuario el programa
- Por eso surgen las pruebas alfa y beta
- Las pruebas *alfa* las llevan a cabo usuarios finales en el lugar de trabajo del desarrollador, con el desarrollador
- Las *pruebas beta* se llevan a cabo en el lugar de trabajo de los usuarios finales, sin el desarrollador

Estrategias de pruebas OO
Pruebas de sistema

- Aparte de las pruebas de validación, hay otras pruebas de sistema
- Muchos sistemas de computadora deben ser capaces de recuperarse ante fallos, otras veces deben obviarlos
  - La *prueba* de *recuperación* fuerza el fallo del software de muchas formas y verifica que la recuperación se lleva a cabo de forma satisfactoria

Estrategias de pruebas OO
Pruebas de sistema

- Muchos sistemas deben proteger sus datos de accesos y manipulaciones inadecuadas
  - La *prueba de seguridad* intenta verificar que los mecanismos de protección incorporados en el sistema lo protegen de accesos indebidos

Estrategias de pruebas OO
Pruebas de sistema

- Muchas veces los sistemas se utilizan más allá de su capacidad normal
  - La *prueba de resistencia* ejecuta un sistema de forma que demande recursos en cantidad, frecuencia o volúmenes anormales

Estrategias de pruebas OO
Pruebas de sistema

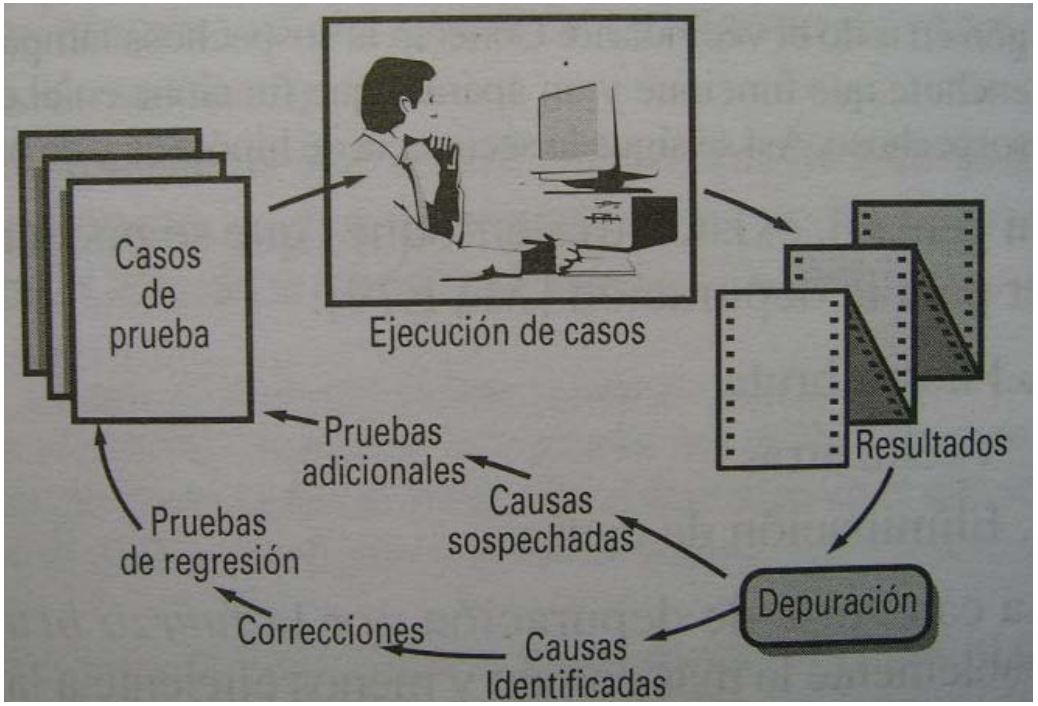
- Los sistemas de tiempo real y los sistemas empotrados tienen fuertes requisitos de rendimiento
  - La *prueba de rendimiento* está diseñada para probar el rendimiento del software en tiempo de ejecución dentro de un sistema
  - Se da incluso a nivel unidad, pero en el contexto del sistema es definitiva

Depuración
Introducción

- La prueba del software produce la *depuración* del mismo
- La depuración consiste en eliminar un error, que presumiblemente, aparecerá como resultado de la prueba

Depuración
Proceso

- La depuración no es una prueba, pero es resultado de la misma



Proceso de depuración del software

Depuración
Proceso

- La depuración puede arrojar dos resultados:
  - Se encuentra y corrige la causa del problema.
  - No se localiza, en cuyo caso habrá que generar nuevos casos de prueba para intentar localizarla
- La depuración suele ser bastante exigente desde el punto de vista psicológico

Pruebas de caja blanca Introducción

- Estas pruebas son aplicables a sistemas orientados a objetos o no
- La prueba de caja blanca usa la estructura de control descrita como parte del diseño para derivar los casos de prueba

Pruebas de caja blanca Introducción

- De esta forma se obtienen casos de prueba que:
  - Garantiza que se recorren todos los caminos independientes de cada módulo.
  - Ejercitan todas las decisiones lógicas en su vertientes *verdadera* y *falsa.*
  - Ejerciten todos los bucles en sus límites.
  - Ejerciten las estructuras internas de datos.

Pruebas de caja blanca
Introducción

- Estas pruebas tienen sentido porque:
  - Los errores lógicos y las suposiciones incorrectas son inversamente proporcionales a la probabilidad de que se ejecute un camino del programa.
  - A menudo creemos que un camino lógico tiene pocas posibilidades de ejecutarse, cuando realmente puede ejecutarse de forma normal

Pruebas de caja blanca Introducción

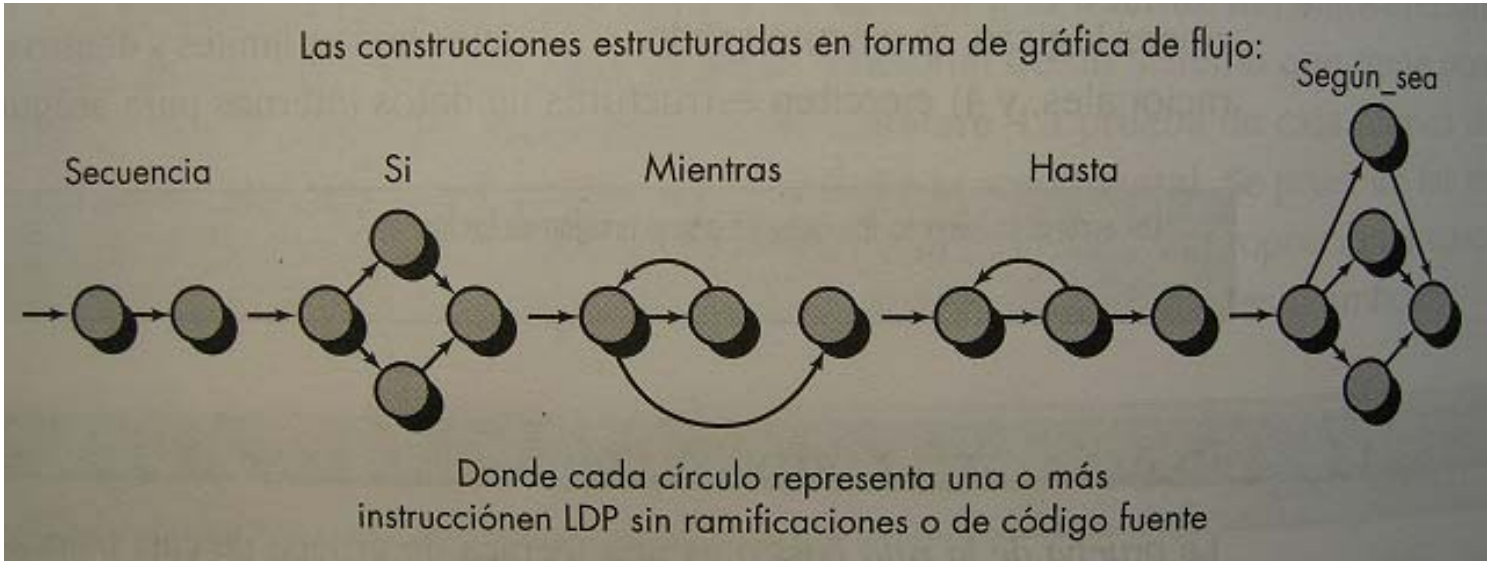
  - Los errores tipográfico son aleatorios
- Hay dos pruebas de caja blanca por excelencia:
  - La prueba del camino básico
  - La prueba de la estructura de control

Pruebas de caja blanca
Prueba del camino básico

- El método del camino básico permite obtener una medida de la complejidad lógica de un diseño procedimental, y usarla como guía para la definición de un conjunto básico de caminos de ejecución.
- Los casos de prueba obtenidos del conjunto básico garantizan que durante la prueba se ejecuta cada sentencia del programa.

Pruebas de caja blanca
Prueba del camino básico

- Notación de grafo de flujo



Notación de grafo de flujo

Pruebas de caja blanca
Prueba del camino básico

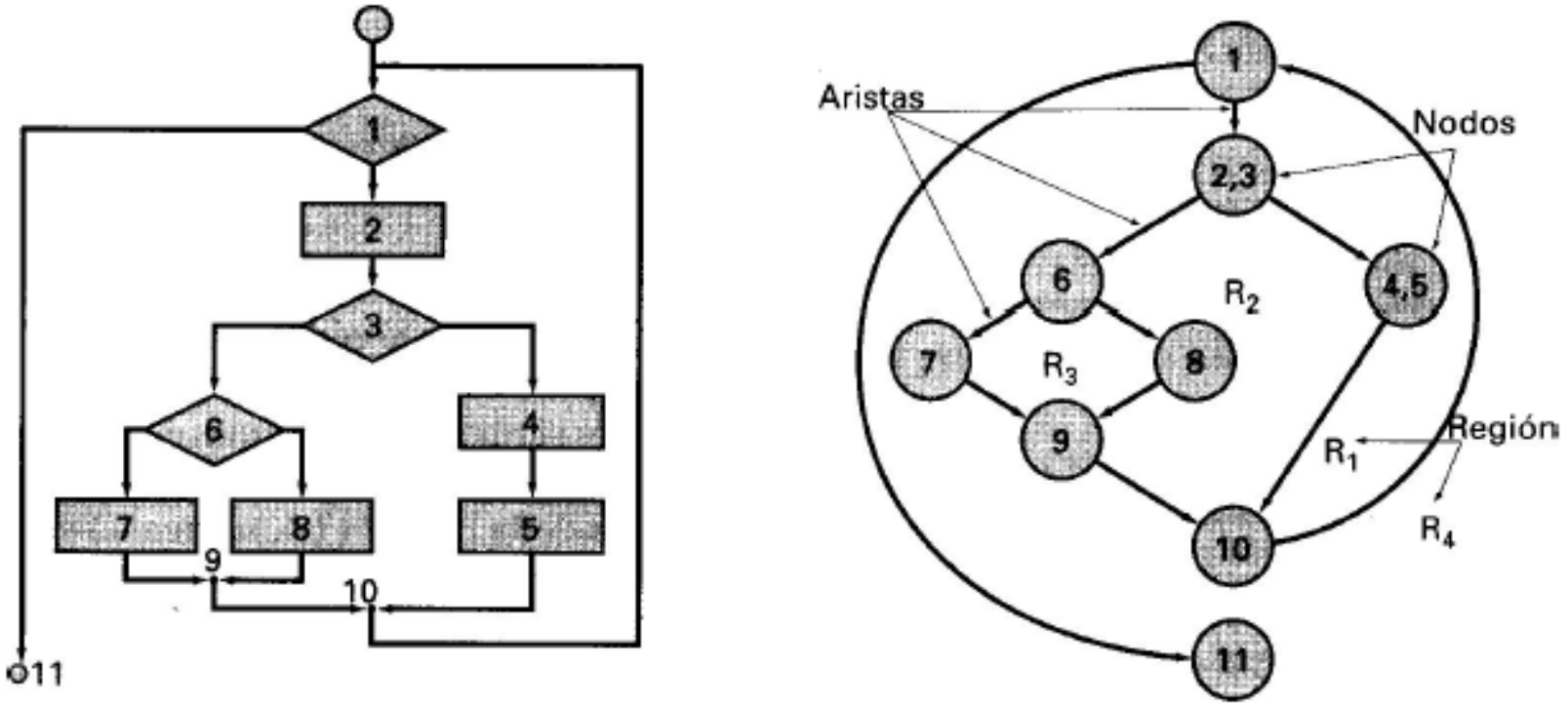


Diagrama de flujo y grafo de flujo asociado

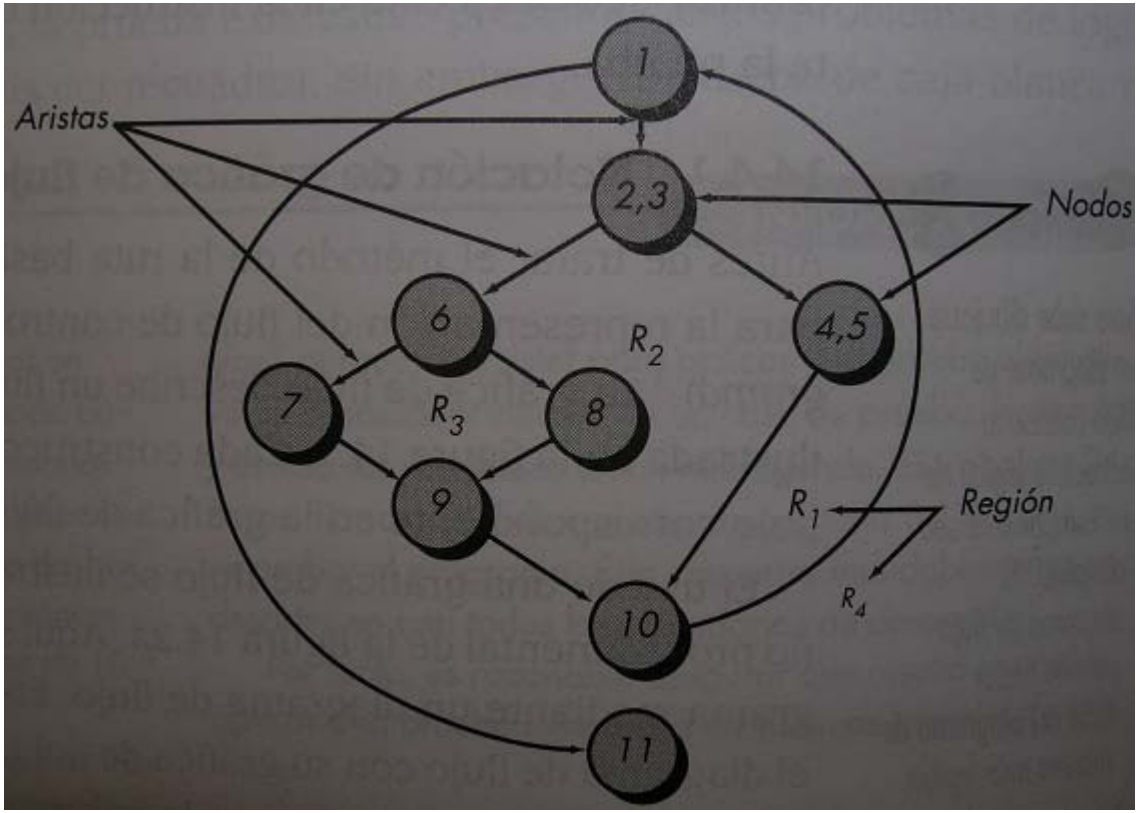
Pruebas de caja blanca
Prueba del camino básico

- La *complejidad ciclomática* es una métrica del software que proporciona una medición cuantitativa de la complejidad lógica de un programa.
  - Define el número de caminos independientes del conjunto básico de un programa.
  - Proporciona un límite superior para las pruebas a realizar que garanticen ejecutar cada sentencia.

Pruebas de caja blanca
Prueba del camino básico

- Un *camino independiente* es cualquier camino del programa que introduce por lo menos un nuevo conjunto de sentencias de proceso o una nueva condición.
- En términos del grafo de flujo, un camino independiente está constituido por lo menos una arista que no haya sido recorrida anteriormente a la definición del camino

Pruebas de caja blanca
Prueba del camino básico



Camino 1: 1-11
Camino 2: 1-2-3-4-5-10-1-11
Camino 3: 1-2-3-6-8-9-10-1-11
Camino 4: 1-2-3-6-7-9-10-1-11
No camino: 1-2-3-4-5-10-1-2-3-6-8-9-10-1-11

Pruebas de caja blanca
Prueba del camino básico

- Los caminos 1-4 componen un *conjunto básico* para el grafo de flujo anterior
- Así, si se pueden diseñar pruebas que fuercen la ejecución de los caminos del conjunto básico se garantizará:
  - Que se habrá ejecutado al menos una vez cada sentencia del programa
  - Que cada condición se habrá ejecutado en sus vertientes verdadera y falsa

Pruebas de caja blanca
Prueba del camino básico

- El número de caminos a buscar lo determina la *complejidad ciclomática*, la cual puede calcularse de tres formas:
  - El número de regiones del grafo
  - V(G) = #aristas - #nodos +2
  - V(G) = #nodos predicados + 1
- Evidentemente, hay que seleccionar casos de prueba que fuercen cada camino del conjunto básico

Pruebas de caja blanca
Prueba del camino básico

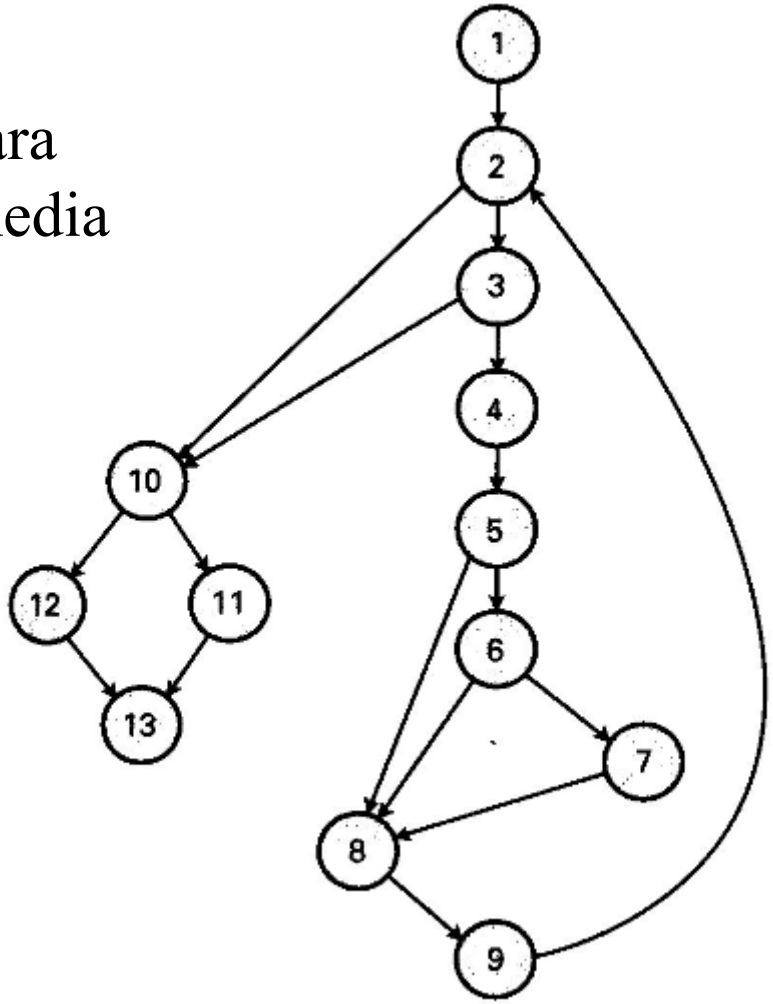
– Ejemplo:

```
PROCEDURE media;
    * Este procedimiento calcula la media de 100 o menos
      números que se encuentren entre unos limites;
      también calcula el total de entradas y el total
      de números válidos.

      INFERFACE RETURNS media, total. entrada, total. válido;
      INTERFACE ACEPTS valor, mínimo, máximo;

      TYPE valor [1:100] IS SCALAR ARRAY;
      TYPE media, total. entrada, total. válido;
         Mínimo, máximo, suma IS SCALAR:
      TYPE i IS INTEGER;
      i = 1;
      total, entrada =total, válido = 0;
      suma =0;
      DO WHILE VALOR [i] <> –999 and total entrada < 100
         Incrementar total entrada en 1;
         IF valor [i] >= mínimo  AND  valor [i] <= maximo
            THEN incrementar total.válido en 1;
                 suma = suma +valor [i];
            ELSE ignorar
         ENDIF
         Incrementar i en 1;
      ENODO
      If total valido > 0
         THEN media =suma/total valido,
         ELSE media = –999,
      ENDIF
END media
```

Procedimiento media

Pruebas de caja blanca
Prueba del camino básico

Grafo de flujo para
procedimiento media



Pruebas de caja blanca
Prueba del camino básico

- La complejidad ciclomática es 6
- El conjunto básico es:

camino 1: 1-2-10-11-13
camino 2: 1-2-10-12-13
camino 3: 1-2-3-10-11-13
camino 4: 1-2-3-4-5-8-9-2-...
camino 5: 1-2-3-4-5-6-8-9-2-...
camino 6: 1-2-3-4-5-6-7-8-9-2- ...

Pruebas de caja blanca
Prueba del camino básico

- Los casos de prueba son:

Caso de prueba del camino 1:
valor *(k)* = entrada válida, donde *k* < i definida a continuación
valor (i) = –999, donde 2 <= i <= 100
resultados esperados: media correcta sobre los *k* valores y totales adecuados.
Nota: el camino 1 no se puede probar por sí solo; debe ser probado como parte de las pruebas de los caminos 4, 5 y 6.

Caso de prueba del camino 2:
valor (1) = –999
resultados esperados: media = –999; otros totales con sus valores iniciales

Caso de prueba del camino 3:
intento de procesar 101 *o* más valores
los primeros 100 valores deben ser válidos
resultados esperados: igual que en el caso de prueba 1

Caso de prueba del camino 4:
valor (i) = entrada válida donde i < 100
valor *(k)* < mínimo, para *k* < i
resultados esperados: media correcta sobre los **k** valores y totales adecuados

Caso de prueba del camino 5:
valor (i) = entrada válida donde i < 100
valor *(k)* > máximo, para *k* <= i
resultados esperados: media correcta sobre los *n* valores y totales adecuados

Caso de prueba del camino 6:
valor (i) = entrada válida donde i < 100
resultados esperados: media correcta sobre los *n* valores y totales adecuados

Pruebas de caja blanca
Prueba de la estruct. de control

- La técnica de prueba del camino básico es una de las técnicas de prueba de la *estructura de control*
- Hay distintas pruebas de la estructura de control:
  - Prueba de condición.
  - Prueba de flujo de datos.
  - Prueba de bucles

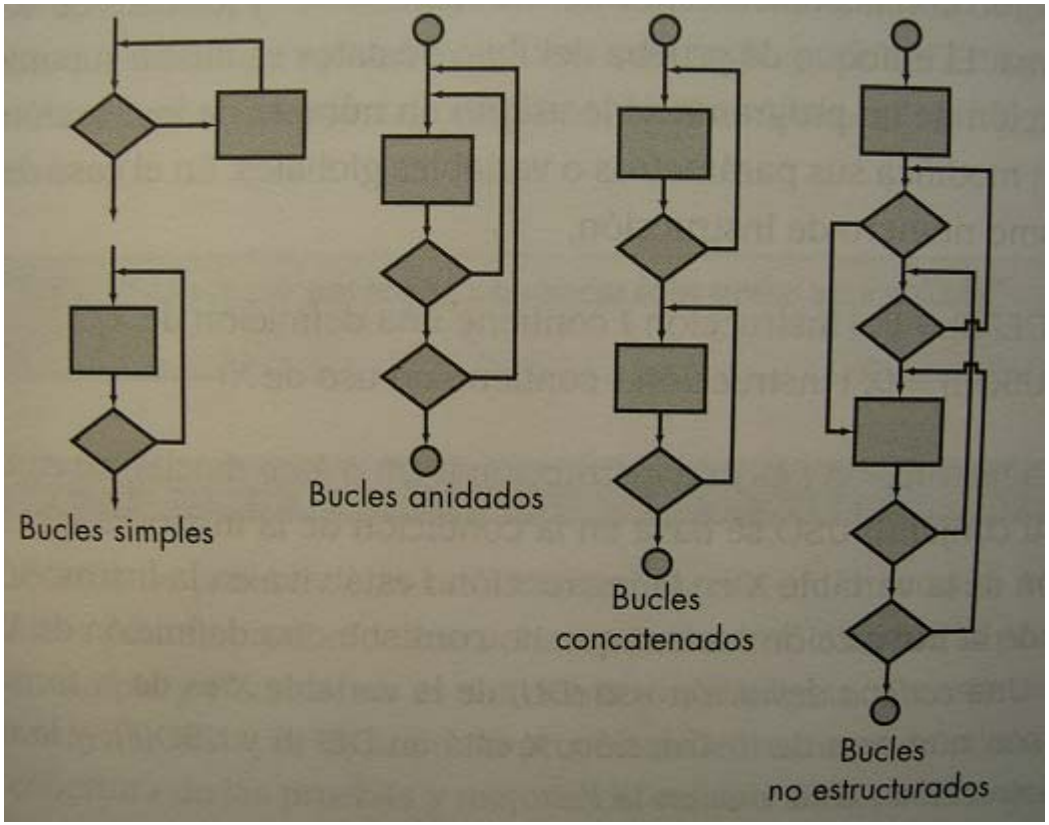
Pruebas de caja blanca
Prueba de la estruct. de control

- La *prueba de condición* ejercita las condiciones lógicas contenidas en el módulo de un programa
- La *prueba de flujo de datos* selecciona caminos de prueba de un programa de acuerdo con la ubicación de las definiciones y los usos de las variables del programa

Pruebas de caja blanca
Prueba de la estruct. de control

- La *prueba de bucles* se centra en probar los bucles
  - Los simples con *n* pasos máximos se probarán:
    - Sin ejecutarlo.
    - Pasando una vez.
    - Pasando dos veces.
    - Pasando *m (<n)* veces
    - Pasando *n-1*, *n* y *n+1* veces

Pruebas de caja blanca
Prueba de la estruct. de control



Tipos de bucles

Pruebas de caja negra Introducción

- Las pruebas de caja negra se centran en los requisitos funcionales del software
- Estas pruebas permiten obtener conjuntos de condiciones de entrada para ejercitar todos los requisitos funcionales de un programa
- Son complementarias a las pruebas de caja blanca

Pruebas de caja negra
Métodos basados en grafos

- El *método basado en grafos* crea un grafo de elementos importantes y sus relaciones, y después se diseña una serie de pruebas que cubran el grafo
  - En el contexto de la POO, dicho grafo podría considerarse como un diagrama de comunicación entre subsistemas

Pruebas de caja negra
Partición equivalente

- La *partición equivalente* divide el dominio de entrada de un programa en clases de datos de los que se puede derivar casos de prueba
  - Para cada entrada se consideran clases de equivalencia, y se prueban sus representantes

Pruebas de caja negra
Partición equivalente

- Por ejemplo:
  - Podemos hacer particiones en base a tipos
    - Enteros
    - Caracteres
    - Caracteres alfanuméricos
  - Podemos hace particiones en base a entidades
    - Empleados a tiempo completo
    - Empleados a tiempo parcial

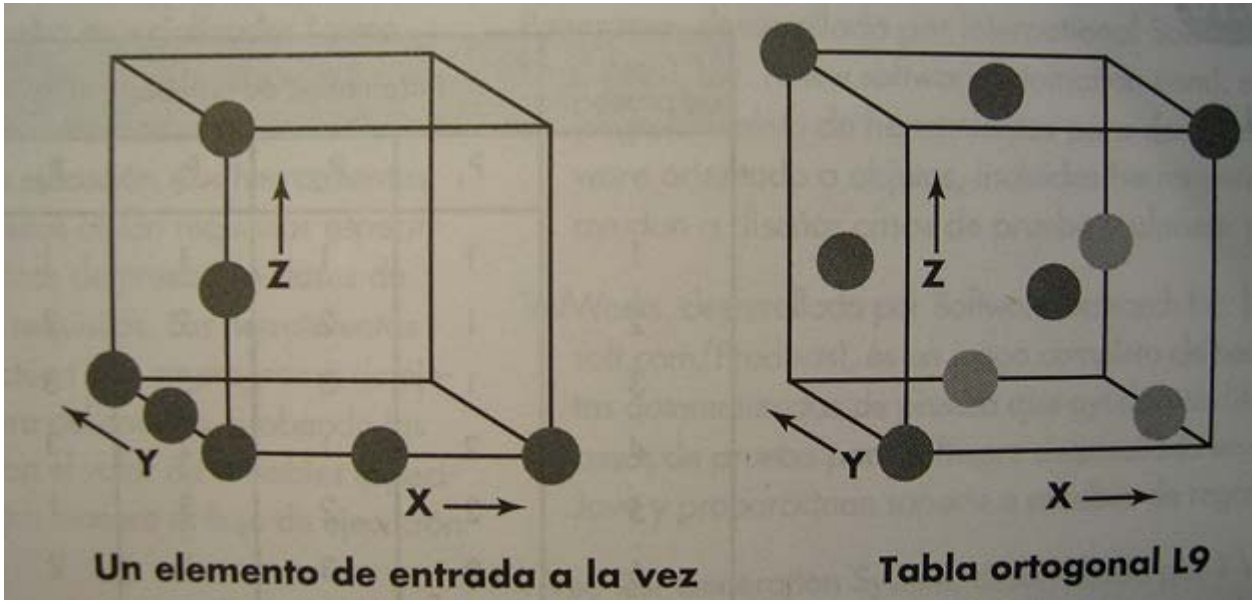
Pruebas de caja negra
Análisis de valores límite

- La *prueba de análisis de valores límite* considera clases de equivalencia, y prueba sus valores límite

Pruebas de caja negra
Tabla ortogonal

- La prueba de tabla ortogonal se aplica cuando el conjunto de parámetros de entrada es pequeño y su dominio finito, pero suficientemente grande como para no poder probar todas sus permutaciones
  - Entonces se prueban tuplas de entrada distribuidas entre todos los parámetros

Pruebas de caja negra
Tabla ortogonal



Prueba tabla ortogonal

Pruebas de caja negra Comparación

- La *prueba de comparación* se basa en desarrollar sistemas software en paralelo, probarlos simultáneamente, y observar la salida
  - Si coinciden, es posible que no haya errores (aunque tampoco es seguro).
  - Si difieren, hay errores.
  - Suele aplicarse en sistemas críticos.

Automatización de pruebas

- Existen marcos que sirven para automatizar el proceso de prueba, como por ejemplo, JUnit*
- Básicamente, comparan objetos obtenidos con objetos esperados
- Otros como JMeter# son para aplicaciones web y pueden medir otros factores (e.g. carga)

* http://www.junit.org/

#http://jmeter.apache.org/

JUnit 5

- JUnit permite automatizar las pruebas
- Para ello se definen clases Java con `@anotaciones` que hacen referencia al marco JUnit
- La principal es `@Test` que denota un test
- Estas anotaciones permiten comparar valores esperados con valores obtenidos

JUnit 5

- En general, utilizaremos un paquete `test`, al mismo nivel que `src` y que replica la estructura de los paquetes a probar en `src`
- JUnit 5 necesita la versión Oxygen para funcionar con Eclipse
- Tenemos JUnit 4 para usar con otros Eclipses

JUnit 5

- JUnit 5 utiliza varias funciones para hacer las pruebas:
  - `fail()`: permite que falle cuando hay éxito
  - `assertTrue()`: comprueba que una condición booleana es cierta
  - `assertFalse()`: comprueba que una condición booleana es falsa
  - `assertEquals()`: comprueba que dos valores coinciden

JUnit 5

- `assertNull()`: comprueba que un objeto es `null`
- `assertNotNull()`: comprueba que un objeto no es `null`
- `assertSame()`: comprueba si dos variables se refieren al mismo objeto
- `assertNotSame()`: comprueba si dos variables no se refieren al mismo objeto

JUnit 5

- Ejemplo:

```
package hola;
public class Hola {
	public String saludo()
	{
		return "hola";
	}
}
```

JUnit 5

```
package hola;

import org.junit.jupiter.api.Test;
import static
org.junit.jupiter.api.Assertions.assertEquals;

import hola.Hola;
```

JUnit 5

```
public class HolaTest {
      @Test
      void exito()
      { Hola h= new Hola();
        assertEquals("hola", h.saludo()); }

     @Test
     void fallo()
     { Hola h= new Hola();
     assertEquals("holaa", h.saludo());}
```

JUnit 5

- También podemos probar excepciones:

```
public String excepcion(int x) throws Exception
      {
            if (x==0) throw new
                              Exception("x era 0");

            return "hello";
       }
```

JUnit 5

```
@Test
void excepcion1Exito()
{     Hola h= new Hola();
      try {
            h.excepcion(0);
      } catch (Exception e) {
      assertEquals("x era 0", e.getMessage());
                              }
}
```

JUnit 5

```
@Test
void excepcion1Fallo()
{     Hola h= new Hola();
      try {
                  h.excepcion(0);
            } catch (Exception e) {
                        fail(e.getMessage());
                              }
}
```

JUnit 5

```
@Test
void excepcion2()
{
      Hola h= new Hola();
      assertThrows(Exception.class,
                        ()-> h.excepcion(0));
}
```

JUnit 5

- También permite hacer test parametrizados:

```
@ParameterizedTest
@ValueSource(ints= {0, 1, 2})
void param(int x)
{     Hola h= new Hola();
      try {
      assertEquals("hello", h.excepcion(x));
} catch (Exception e) { fail(e.getMessage());}
}
```

JUnit 5

- La comparación entre objetos compara direcciones de memoria
- Por tanto, si queremos comparar objetos (incluidos aquellos que tienen colecciones de objetos) lo más sencillo es utilizar la comparación basada en la implementación de `equals()`o de `toString()`

JUnit 4

- JUnit 4 es similar (salvo librerías concretas del marco)
- Las únicas diferencias significativas con los visto están en el tratamiento de excepciones y en los test parametrizados
- Además, los métodos `@Test` deben ser públicos

JUnit 4

```
@Rule
public ExpectedException thrown =
ExpectedException.none();
@Test
public void excepcion2() throws Exception {
      Hola h= new Hola();
      thrown.expect(Exception.class);
      thrown.expectMessage("x era 0");
      h.excepcion(0);
}
```

JUnit 4

```
@RunWith(Parameterized.class)
public class HolaTestParam {
      //valores de los parámetros
      @Parameters
      public static Collection<Object[]> data() {
      return Arrays.asList(new Object[][]{
                                    {0},
                                    {1},
                                    {2}});
      }
```

JUnit 4

```
//parámetro
private int x;

//constructor
public HolaTestParam(int x)
{
this.x= x;
}
```

JUnit 4

```
    @Test
    public void param()
    {
        Hola h= new Hola();
    try {
        assertEquals("hello", h.excepcion(x));
} catch (Exception e) { fail(e.getMessage());}
}
}
```

Conclusiones

- Objetivo global: encontrar el máximo número de errores con el mínimo esfuerzo
- Estrategia de pruebas
  - Unidad.
  - Integración.
    - Hilos
    - Uso
  - Sistema.

Conclusiones

- Depuración
- Pruebas de caja blanca
- Pruebas de caja negra
- Automatización de pruebas