Sesión S11: Análisis de pruebas

Uso de métricas para análisis de pruebas

Cobertura de código

Herramienta automática de análisis: Cobertura

Vamos al laboratorio...

María Isabel Alfonso Galipienso
Universidad de Alicante eli@ua.es

EL PROCESO DE PRUEBAS

SEGÚN ISQTB FOUNDATION LEVEL SYLLABUS (2011)

Definición del proceso de pruebas

Las pruebas son un conjunto de actividades conducentes a conseguir alguno de estos objetivos:

* Encontrar **defectos**
* Evaluar el nivel de **calidad** del software
* Obtener información para la toma de decisiones
* Prevenir defectos

(ISQTB Foundation Level Syllabus -2011)

Actividades del proceso de pruebas

- **Planificación** y **control** de las pruebas
- **Diseño** de las pruebas
- **Implementación** y **ejecución** de las pruebas
- **Evaluar** si hemos alcanzado las metas definidas y emitir un **informe**

Definimos los objetivos de las pruebas (en cada nivel de pruebas tenemos objetivos diferentes). Establecemos QUÉ, CÓMO y CUÁNDO vamos a realizarlas

Es el proceso más importante para cumplir con los objetivos marcados. Básicamente consiste en decidir, de forma sistemática, con qué datos de entrada concretos vamos a probar el código. En cada nivel de pruebas el proceso de diseño es diferente.

La idea es poder ejecutar las pruebas diseñadas de forma automática. En cada nivel de pruebas usaremos "herramientas" diferentes.

Como resultado de la ejecución de las pruebas obtendremos información sobre el proceso realizado. En cada nivel de pruebas obtendremos informes diferentes

?

- ¿Qué información podemos obtener de las pruebas?
- ¿Para qué es útil dicha información?

Métricas y análisis de pruebas

- Para analizar y extraer conclusiones sobre las pruebas realizadas sobre nuestro proyecto software, necesitamos "cuantificar" el proceso y resultado de las mismas, es decir, utilizar métricas que nos permitan conocer con la mayor objetividad diferentes características de nuestras pruebas.
- Una **métrica** se define como una medida cuantitativa del grado en el que un **sistema**, **componente** o **proceso**, posee un determinado atributo
  - Si no podemos medir, no podemos saber si estamos alcanzando nuestros objetivos, y lo más importante, no podremos "controlar" el proceso software ni podremos mejorarlo
  - Tiene que haber una relación entre lo que podemos medir, y lo que queremos "conocer".
- ¿Qué podemos medir? Casi cualquier cosa: líneas de código probadas, número de errores encontrados, número de pruebas realizadas, número de clases probadas, número de horas invertidas en realizar las pruebas,...

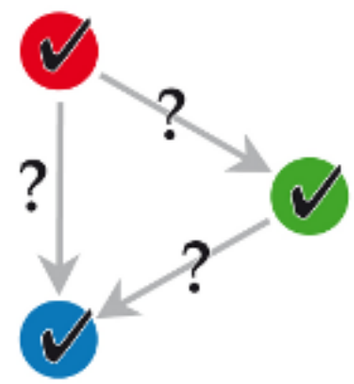
Análisis de las pruebas

- independientemente del objetivo concreto de nuestras pruebas, siempre buscamos **efectividad** (no ser efectivo implica no cumplir nuestro objetivo)
- Básicamente, hay dos causas fundamentales que pueden provocar que nuestras pruebas no sean efectivas
  - Podemos ejecutar el código, pero con un mal diseño de pruebas, de forma que los casos de prueba sean tales que nuestro código esté "pobremente" probado o no probado de ninguna manera
  - Podemos, de forma "deliberada", dejar de probar partes de nuestro código
- La primera cuestión es bastante complicada de detectar de forma automática
- Por otro lado, está claro que si parte de nuestro código no se ejecuta durante las pruebas, es que no está siendo probado
  - Vamos a detenernos en esta cuestión: en el problema de la COBERTURA de código, es decir en analizar cual es la EXTENSIÓN de nuestras pruebas
- La COBERTURA de código es la característica que hace referencia a cuánto código estamos probando con nuestros tests (porcentaje de código probado)
  - IMPORTANTE: NO proporciona un indicador la calidad del código, ni la calidad de nuestras pruebas, sino de la **extensión** de nuestros tests

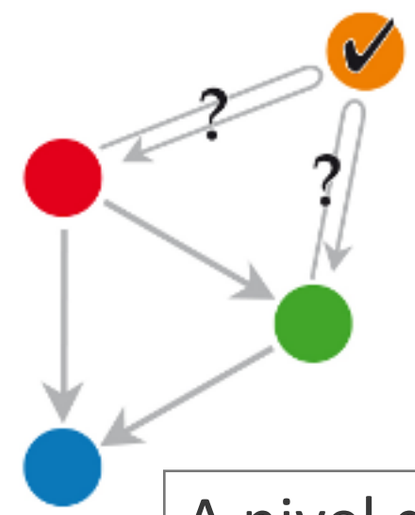
Cobertura de código

- El análisis de la cobertura de código se lleva a cabo mediante la "exploración", de forma dinámica, de diferentes de flujos control del programa

```
a = calcular(valor);
if ((a || b) && c) {
    contador = 0;
    a = calcular(b);
}
contador++;
```



A nivel de bloque




```
EvaluateA( ... )
{
}
```

A nivel de función

```
a = calcular(valor);
if ((a || b) && c) {
    contador = 0;
    a = calcular(b);
}
contador++;
```

A nivel de decisiones

```
a = calcular(valor);
if ((a || b) && c) {
    contador = 0;
    a = calcular(b);
}
contador++;
```

A nivel de condiciones

... entre otras

Análisis de la cobertura de código (I)

Pragmatic Software Testing. Rex Black. Chapter 21

- Hay **siete** formas principales para cuantificar la cobertura de código:
  - Statement coverage: un 100% significa que hemos ejecutado cada sentencia (línea)
  - Branch (or decision) coverage: un 100% significa que hemos ejecutado cada rama o DECISIÓN en sus vertientes verdadera y falsa.
    - Una decisión es una expresión booleana formada por condiciones y cero o más operadores booleanos
    - Para sentencias if, necesitamos asegurar que la expresión que controla las "ramas" se evalúa a cierto y a falso
    - Para sentencias "switch" necesitamos cubrir cada caso especificado, así como al menos un caso no especificado (o el caso por defecto)
    - Un 100% de cobertura de ramas implica un 100% de cobertura de líneas
  - Condition coverage: un 100% significa que hemos ejercitado cada CONDICIÓN. Cuando tenemos expresiones con múltiples condiciones, para conseguir el 100% de cobertura de las condiciones tenemos que evaluar el comportamiento para cada una de dichas condiciones en sus vertientes verdadera y falsa
    - Una condición es el elemento "mínimo" de una expresión booleana (no puede descomponerse en una expresión booleana más simple)
    - Por ejemplo, para la sentencia if ((A>0) && (B>0)), necesitamos probar que (A>0) sea cierto y falso, y (B>0) sea cierto y falso. Esto lo podemos conseguir con dos tests: true && true, y false && false

Análisis de la cobertura de código (II)

- Multicondition coverage: un 100% de cobertura significa que hemos ejercitado todas las posibles combinaciones de condiciones.
  - Siguiendo con el ejemplo anterior, para la sentencia if ((A>0) && (B>0)), necesitamos probar las cuatro posibles combinaciones: true && true, true && false, false && true y false && false

Nota: los símbolos "&&" hacen referencia a una operación lógica AND, no necesariamente tiene que referirse al operador "&&" de Java

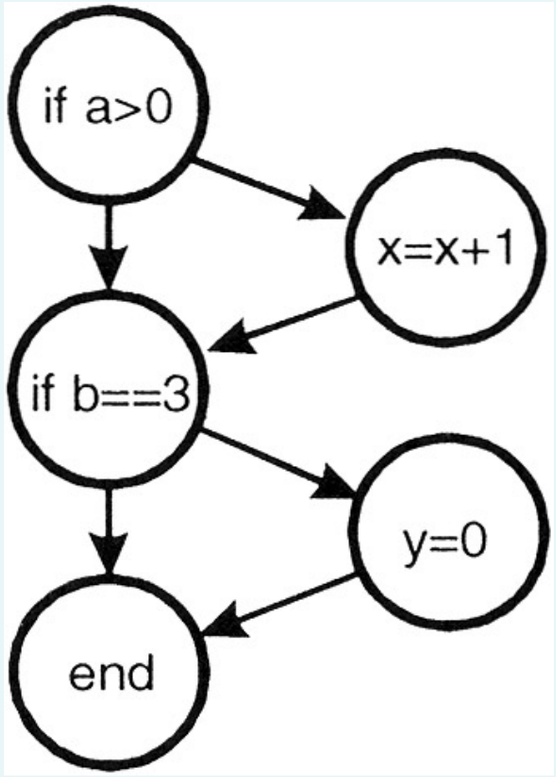
- Condition decision coverage: en lenguajes como C++ y Java, en donde las condiciones "siguientes" se evalúan dependiendo de las condiciones que las preceden, un 100% de cobertura de condiciones múltiples puede no tener sentido
  - Por ello ejercitaremos cada condición en el programa (cada una toma todos sus posibles valores al menos una vez), y cada decisión en el programa (cada una toma todos sus valores posibles al menos una vez)
  - Para la sentencia java if ((A>0) && (B>0)), probaremos con las combinaciones true && true, true && false y false && true. La combinación false && false no es necesaria debido a que la segunda condición no puede influenciar la decisión tomada por la expresión de la sentencia if
- Loop coverage: un 100% de cobertura implica probar el bucle con 0 iteraciones, una iteración y múltiples iteraciones
- Path coverage: un 100% de cobertura implica que se han probado todos los possibles caminos de control. Es muy difícil de conseguir cuando hay bucles. Un 100% de cobertura de caminos implica un 100% de cobertura de ramas

Niveles de cobertura de código

statement coverage

- **NIVEL 1**: cobertura de líneas (100%)

```
if (a>0) {
    x=x+1;
}
if (b==3) {
    y=0;
}
```



- Para este código podemos conseguir un 100% de cobertura de líneas con un único caso de prueba (por ejemplo a=6, y b=3). Sin embargo estamos dejando de probar 3 de los cuatro caminos posibles.
- Un 100% de cobertura de líneas no suele ser un nivel aceptable de pruebas. Podríamos clasificarlo como de NIVEL 1
- A pesar de constituir el nivel más bajo de cobertura, en la práctica puede ser difícil de conseguir

Realmente hay un **NIVEL 0** , el cual se define como: “test whatever you test; let the users test the rest” [Lee Copeland, 2004]

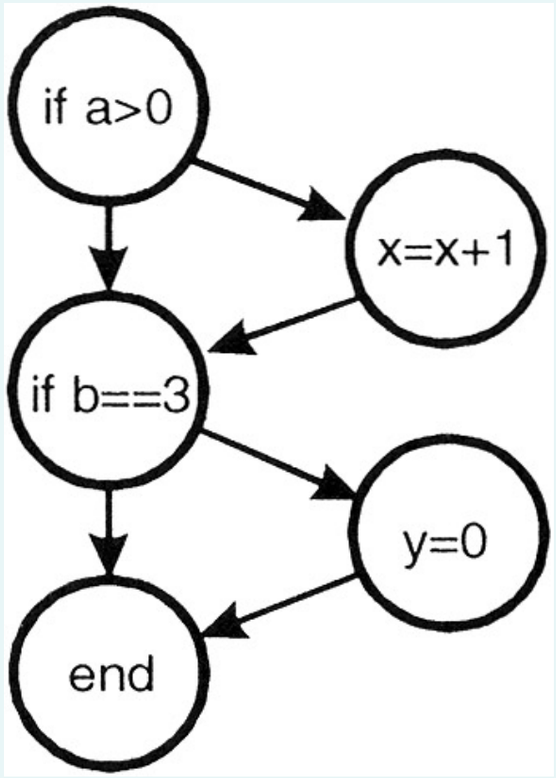
Boris Beizer, en una ocasión escribió: "testing less than this [100% statement coverage] for new software is unconscionable and should be criminalized. ... In case I haven't made myself clear, ... untested code in a system is stupid, shortsighted, and irresponsible." [Beizer, 1990]

Niveles de cobertura de código

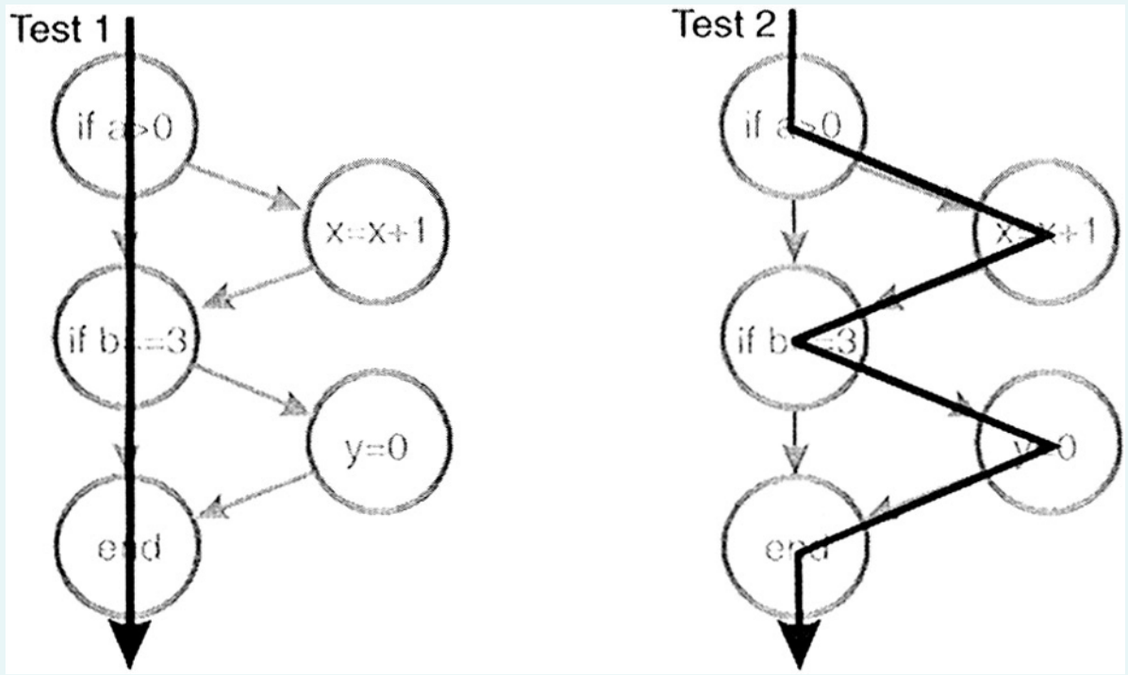
branch (decision) coverage

- **NIVEL 2**: cobertura de ramas (100%)

```
if (a>0) {
    x=x+1;
}
if (b==3) {
    y=0;
}
```



- Para este código podemos conseguir un 100% de cobertura de ramas con dos casos de prueba (por ejemplo a=0, b=2; y a=4, b=3). Sin embargo estamos dejando de probar 2 de los cuatro caminos posibles.



Un 100% de cobertura de ramas (decisiones) implica un 100% de cobertura de líneas

- Observa que para conseguir la cobertura de ramas (el 100%), necesitamos diseñar casos de prueba de forma que cada DECISIÓN que tenga como resultado true/false sea evaluada al menos una vez

Niveles de cobertura de código

condition coverage

- **NIVEL 3**: cobertura de condiciones (100%)

```
if (a>0 && c==1) {
    x=x+1;
}
if (b==3 || d<0) {
    y=0;
}
```

- Para que la primera sentencia sea cierta, a tiene que ser >0 y c = 1. La segunda requiere que b= 3 ó d<0
- En la primera sentencia, si el valor de a lo fijamos a 0 para hacer las pruebas, probablemente la segunda condición (c==1) no será probada (dependerá del lenguaje de programación)

&& no es el operador java!!

- Podemos conseguir el nivel 3 con dos casos de prueba:
  - (a=7, c=1, b=3, d = -3)
  - (a=-4, c=2, b=5, d = 6)

Un 100% de cobertura de condiciones NO garantiza un 100% de cobertura de líneas

- La cobertura de condiciones es mejor que la cobertura de decisiones debido a que cada CONDICIÓN INDIVIDUAL es probada (en sus vertientes verdadera y falsa) al menos una vez, mientras que la cobertura de decisiones puede conseguirse sin probar cada condición

Niveles de cobertura de código

condition + decision coverage

- **NIVEL 4**: cobertura de condiciones (100%) y decisiones (100%)

```
if(x && y) {
   sentenciaCond;
}
// && es un AND lógico
```

- En este ejemplo podemos conseguir el nivel 3 (cobertura de condiciones), con dos casos: (x =TRUE, y = FALSE) y (x=FALSE, y =TRUE). Pero observa que en este caso "sentenciaCond" nunca será ejecutada. Podemos ser más completos si buscamos también una cobertura de decisiones

- Podemos conseguir el nivel 4 creando casos de prueba para cada condición y cada decisión:
  - (x=TRUE, y=FALSE),
  - (x=FALSE, y=TRUE),
  - (x=TRUE, y=TRUE)

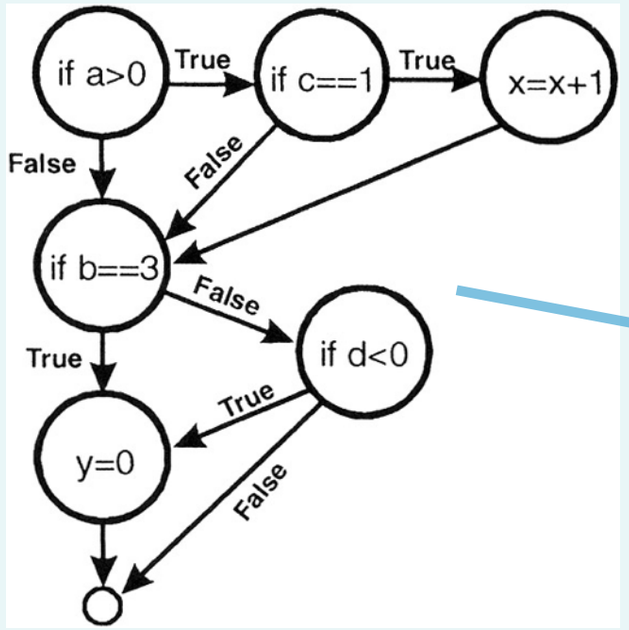
Niveles de cobertura de código

multicondition coverage

- **NIVEL 5**: cobertura de condiciones múltiples (100%)

```
if (a>0 & c==1) {
    x=x+1;
}
if (b==3 | d<0) {
  y=0;
}
```

operador && de java



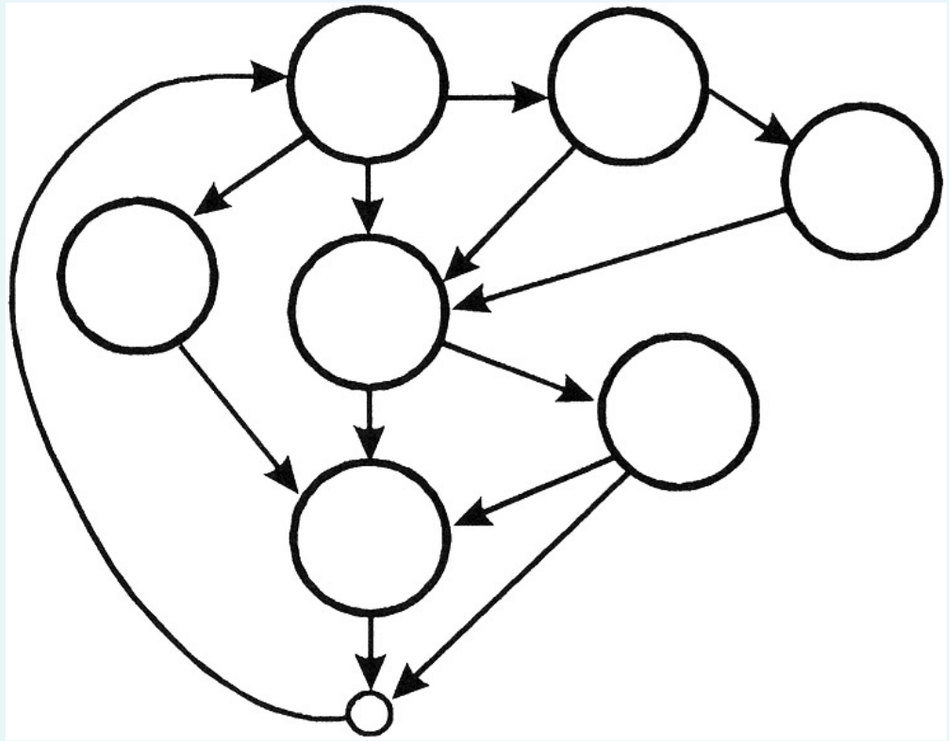
- En este ejemplo con código Java (sin cortocircuito), este nivel de cobertura podemos conseguirlo con cuatro casos de prueba:
  - a= 5, c=1, b=3, d= -3
  - a=-4, c=1, b=3, d= 7
  - a= 5, c=2, b=5, d= -4
  - a=-1, c=2, b=5, d=0
- Si el lenguaje evalúa las condiciones en "cortocircuito" (&&, ||) podemos conseguir un 100% de cobertura de condiciones múltiples, considerando cómo el compilador evalúa realmente las condiciones múltiples de una decisión.
  - a= 5, c=1, b=5, d=0
  - a= 5, c=2, b=5, d= -4
  - a= -1, c=2, b=3, d= -3
- Si conseguimos un 100% de cobertura de condiciones múltiples, también conseguimos cobertura de decisiones, condiciones y condiciones+decisiones
- Una cobertura de condiciones múltiples no garantiza una cobertura de caminos

Un 100% de cobertura de condiciones múltiples implica un 100% de cobertura de condiciones, decisiones, y condiciones/decisiones

Niveles de cobertura de código

path coverage

- **NIVEL 7**: cobertura de caminos (100%)
  - Si conseguimos una cobertura del 100% de caminos garantizamos que recorremos todas las posibles combinaciones de caminos de control
  - Para códigos de módulos sin bucles, generalmente el número de caminos es lo suficientemente "pequeño" como para que pueda construirse un caso de prueba para cada camino. Para módulos con bucles, el número de caminos puede ser "enorme", de forma que el problema se haga intratable



Un 100% de cobertura de caminos implica un 100% de cobertura de bucles, ramas y sentencias

Un 100% de cobertura de caminos NO garantiza una cobertura de condiciones múltiples (ni al contrario)

Niveles de cobertura de código

loop coverage

- **NIVEL 6**: cobertura de bucles (100%)
  - Cuando un módulo tiene bucles, de forma que el número de caminos hacen impracticable las pruebas, puede conseguirse una reducción significativa de esfuerzo limitando la ejecución de los bucles a un pequeño número de casos
    - Ejecutar el bucle 0 veces
    - Ejecutar el bucle 1 vez
    - Ejecutar el bucle n veces, en donde n es un número que representa un valor típico
    - Ejecutar el bucle en su máximo número de veces m (adicionalmente se pueden considerar la ejecución (m−1) y (m+1))

Analizadores automáticos de cobertura

- Los analizadores de código que utilizan las herramientas de cobertura se basan en la instrumentación de dicho código.
  - Instrumentar el código consiste en añadir código adicional para poder obtener una métrica de cobertura (el código adicional añade algún contador de código que devuelve un resultado después de la ejecución del mismo)
- Si hablamos de Java, podemos encontrar tres categorías:
  - Inserción de código de instrumentación en el código fuente
  - Inserción de código de instrumentación en el byte-code de Java
    - Esta aproximación es la que utiliza Cobertura
  - Ejecución de código en una máquina virtual de Java modificada
- Vamos a ver como ejemplo una herramienta automática que analiza la cobertura de código:
  - COBERTURA

Métricas que utiliza Cobertura

- Cobertura instrumenta el bytecode de Java de forma que cuando se ejecuta dicho código, se ponen en funcionamiento los analizadores de Cobertura
  - Está basada en JCoverage (herramienta comercial)
  - Puede utilizarse desde línea de comandos, integrada con Ant, o también con Maven
- Cobertura genera un informe con 3 métricas concretas:
  - Line coverage
    - ¿Cuántas líneas se ejecutan?
  - Branch coverage
    - ¿Se ejecuta cada condición en sus vertientes verdadera y falsa?
    - ¡CUIDADO!, a pesar de que hemos definido la cobertura de ramas como cobertura de DECISIONES, la herramienta cobertura, cuando calcula la métrica "branch coverage" está considerando el porcentaje de condiciones cubiertas frente al total de condiciones en el programa.
  - Complejidad ciclomática
    - ¿Cuántos casos de prueba son necesarios para cubrir todos los caminos linealmente independientes?

Informes que genera Cobertura

Coverage Report

Packages

All
ppss.cobertu

All Packages

Classes

App *(83%)*

Coverage Report - All Packages

Package	# Classes	Line Coverage		Branch Coverage		Complexity
All Packages	1	83%	5/6	50%	2/4	3
ppss.cobertura.example	1	83%	5/6	50%	2/4	3

Report generated by Cobertura 1.9.4.1 on 23/11/10 11:42.

Métricas

Coverage Report

Packages

All
org.is2.cobertu

All Packages

Classes

App *(83%)*

Coverage Report - org.is2.cobertura.example.App

Classes in this File	Line Coverage		Branch Coverage		Complexity
App	83%	5/6	50%	2/4	3

```
1     package ppss.cobertura.example;
2
3     /**
4      * Ejemplo de uso de cobertura con Maven
5      *
6      */
7
8  1  public class App {
9
10            public void two_ifs(int i) {
11 1                    if (i<20) {
12 1                            System.out.println("Soy un numero menor que 20");
13                      }
14 1                    if (i % 2 == 0) {
15 0                            System.out.println("Soy numero par");
16                      }
17 1          }
18
19    }
```

Report generated by Cobertura 1.9.4.1 on 23/11/10 11:42.

Las líneas ejecutables se muestran en verde

Muestra el número de veces que se ejecuta cada línea

Resultado de ejecutar `two_ifs(11)`

Grafo de flujo de un programa

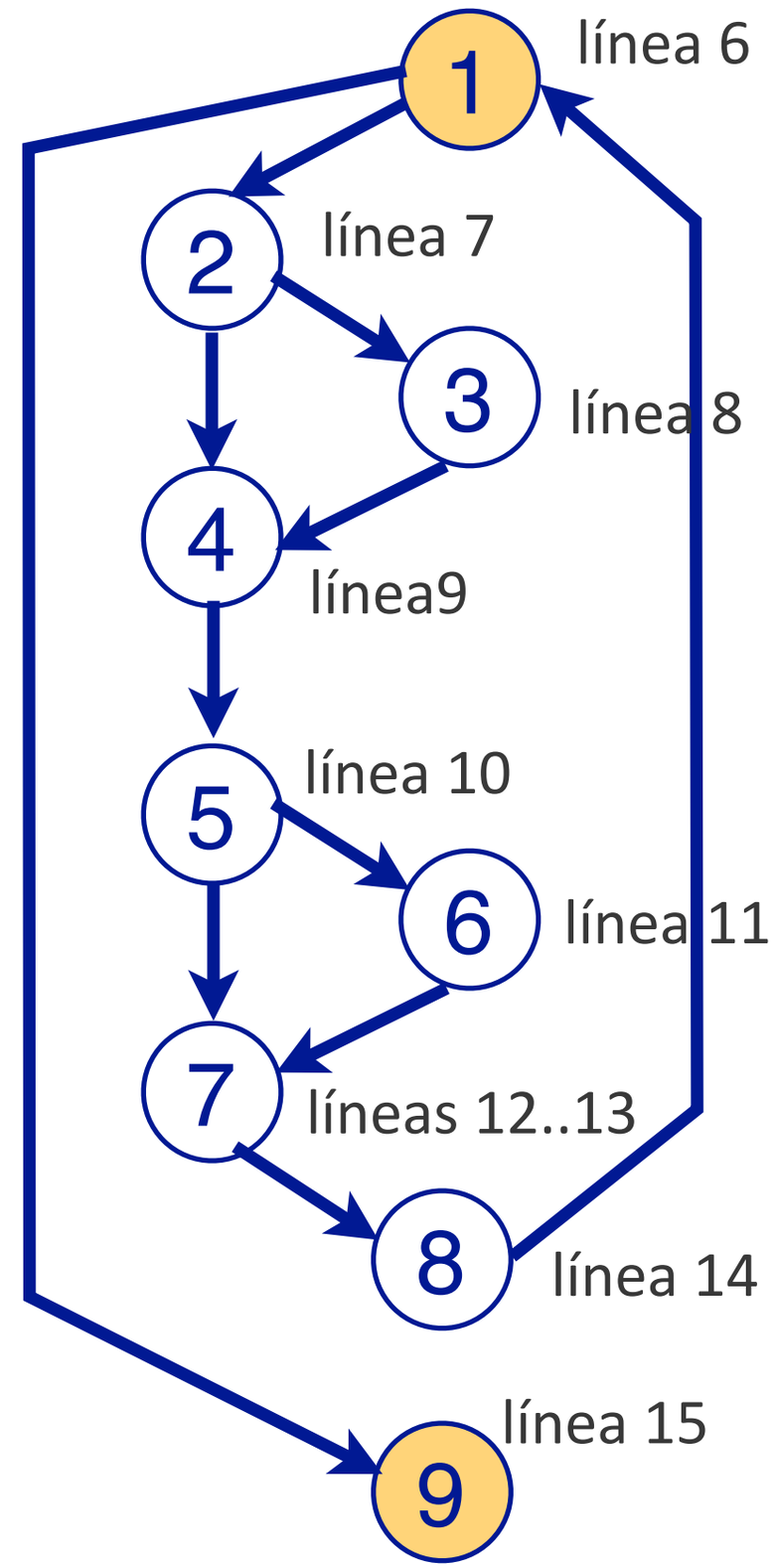
TODAS las sentencias deben estar representadas en el grafo

Cada NODO representa como máximo una condición y/o cualquier número de sentencias secuenciales

Example.java

```
1  public class Example {
2    boolean var= false;
3
4    public void nothing(int number) {
5
6      while (var == false) {
7        if (number %2 == 0)
8          System.out.println("Even number");
9
10       if (number >20)
11         System.out.println("Greater than 20");
12
13       var = true;
14     }
15   }
16 }
```

Cada ARISTA representa el flujo de control de las sentencias

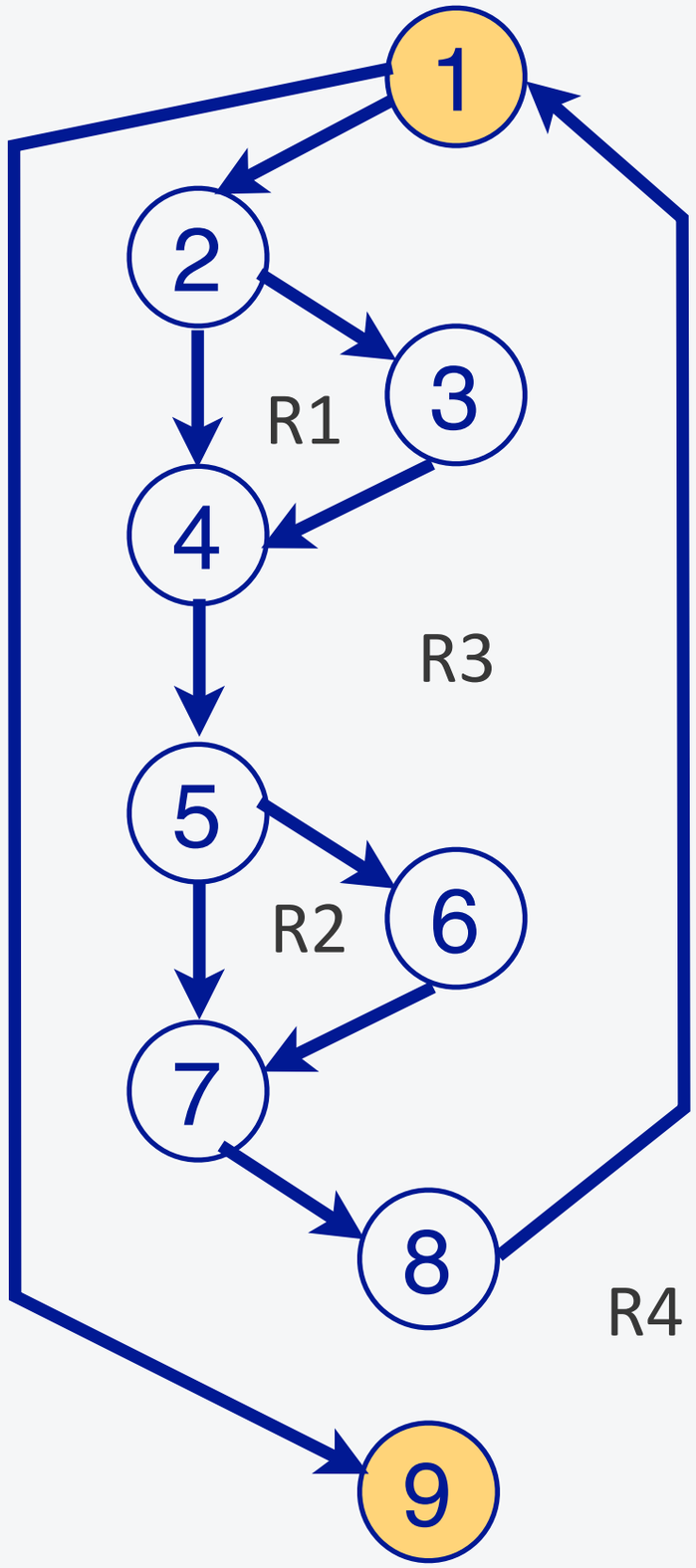


Grafo de flujo y cálculo de CC

- La complejidad ciclomática (CC) es una cota superior del número de caminos linealmente independientes
- Hemos visto que se puede calcular de varias formas:
  - (1) CC = arcos – nodos + 2 = 11–9+2
  - (2) CC = número de condiciones +1 = 3+1
  - (3) CC = número de regiones cerradas en el grafo, incluyendo la externa = 4

Las alternativas (2) y (3) SÓLO se pueden utilizar si el programa NO tiene saltos incondicionales

- Cobertura calcula la CC utilizando una cuarta alternativa:
  - CC = número de condiciones +número de salidas (siendo este número como mínimo 1)



¿Para qué sirve la CC?

- Mide la complejidad lógica del código:
  - Cuanto mayor sea su valor el código resulta mucho más difícil de mantener (y de probar), con el consiguiente incremento de coste así como el riesgo de introducir nuevos errores
  - Si el valor es muy alto (> 15) puede llevarnos a tomar la decisión de refactorizar el código para mejorar la mantenibilidad del mismo
- Nos da una cota superior del número máximo de tests a realizar de forma que se se garantice la ejecución de TODAS las líneas de código al menos una vez, y también nos garantiza que TODAS las condiciones se ejecutarán en sus vertientes verdadera y falsa (cuidado: cada nodo del grafo debe representar como máximo una única condición)
- La herramienta Cobertura calcula de forma automática el valor de CC

Plugin de Cobertura para Maven

http://www.mojohaus.org/cobertura-maven-plugin/

Goal	Report?	Description
cobertura:check	No	Check the coverage percentages for unit tests from the last instrumentation, and optionally fail the build if the targets are not met. To fail the build you need to set `configuration/check/haltOnFailure=true` in the plugin's configuration.
cobertura:check-integration-test	No	Check the coverage percentages for unit tests and integration tests from the last instrumentation, and optionally fail the build if the targets are not met. To fail the build you need to set `configuration/check/haltOnFailure=true` in the plugin's configuration.
cobertura:clean	No	Clean up the files that Cobertura Maven Plugin has created during instrumentation.
cobertura:cobertura	Yes	Instrument the compiled classes, run the unit tests and generate a Cobertura report.
cobertura:cobertura-integration-test	Yes	Instrument the compiled classes, run the unit tests and integration tests and generate a Cobertura report.
cobertura:instrument	No	Instrument the compiled classes.
datafile		
cobertura:help	No	Display help information on cobertura-maven-plugin. Call `mvn cobertura:help -Ddetail=true -Dgoal=<goal-name>` to display parameter details.

Instrumentación e Informe de Cobertura

- La goal cobertura: instrument
  - Modifica el bytecode (ficheros .class) de Java para incluir un contador del número de veces que se ejecuta cada línea de código. Dicha información se "acumula" en cada ejecución en un fichero denominado **cobertura.ser** (en el directorio ${dir_proyecto}/target/cobertura). La goal "clean" "borra" (pone los contadores a cero) los resultados acumulados de ejecuciones previas
  - Por defecto NO se instrumentan las clases de test y el resultado (clases instrumentadas) se almacena en el directorio: ${dir_proyecto}/target/generated-classes/cobertura
- El informe de cobertura se genera en el directorio ${dir_proyecto}/target/site/cobertura . Por defecto se genera en formato html. Podemos **obtener el informe**:
  - Utilizando las goals **cobertura:cobertura**, **cobertura:cobertura-integration-test**
    - Incluiremos el plugin de cobertura en la sección <build> de nuestro pom.
    - Podremos configurar la instrumentación de cobertura y los porcentajes de cobertura a alcanzar (que se comprobarán después de ejecutar los tests)
  - Utilizando la fase **site**
    - Incluiremos el plugin de cobertura dentro de la sección <reporting> del pom. En este caso usaremos la configuración básica del plugin.

Ejemplo: clase a probar y clase de Test

- Nuestra clase a probar tiene un método denominado two_ifs

src/main/java/ppss/App.java

```java
package ppss;

public class App
{
    public int two_ifs(int i) {
     if (i<20) {
       System.out.println("Soy un numero menor que 20");
       return 1;
     }
     if (i % 2 == 0) {
       System.out.println("Soy numero par");
       return 2;
    }
    return 0;
  }
}
```

- Escribimos un primer test que únicamente crea un objeto de tipo App

src/test/java/ppss/AppTest.java

```java
package ppss;

import org.junit.Test;

public class AppTest
{
    @Test
    public void testSimpleTwo_ifs() {
        //Inicialmente la prueba únicamente crea una instancia de App
        App app = new App();

     }

}
```

Generación de informes (I)

ALTERNATIVA 1

- Podemos generar el informe de cobertura ejecutando la goal cobertura:cobertura (o la goal cobertura:cobertura-integration-test)
  - `mvn cobertura:cobertura` ← Instrumenta, ejecuta los tests unitarios, y genera un informe de cobertura

Configuración del proyecto

```
<build>
    <plugins>
      <plugin>
         <groupId>org.codehaus.mojo</groupId>
         <artifactId>cobertura-maven-plugin</artifactId>
         <version>2.7</version>
      </plugin>
    </plugins>
</build>
```

  - Esta goal invoca la ejecución de la fase del ciclo de vida "test" antes de ejecutarse a sí misma
  - Se ejecuta en su propio ciclo de vida (instrumenta el código, ejecuta los tests y genera un informe)
  - La información de la **instrumentación** se almacena en:
    - `target/cobertura/cobertura.ser`
  - Las **clases instrumentadas** se generan en:
    - `target/generated-classes/cobertura`
  - El **informe** de cobertura se genera en:
    - `target/site/cobertura`

Generación de informes (II)

ALTERNATIVA 2

- Podemos generar el informe de cobertura ejecutando la fase site:
  - `mvn site`

Configuración del proyecto

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-project-info-reports-plugin</artifactId>
      <version>3.0.0</version>
    </plugin>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-site-plugin</artifactId>
      <version>3.7.1</version>
    </plugin>
  </plugins>
</build>
<reporting>
  <plugins>
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>cobertura-maven-plugin</artifactId>
      <version>2.7</version>
    </plugin>
  </plugins>
</reporting>
```

maven-project-info-reports-plugin: se usa durante la fase site para generar informes del proyecto. Necesitamos la versión 3.0.0 (Si no lo añadimos, se usará la 2.8)

maven-site-plugin: se usa para generar el "sitio web" del proyecto. Queremos usar la versión 3.7.1 (Si no lo añadimos, usaremos la 3.3)

maven-site-plugin: La etiqueta <reporting> permite gestionar los informes generados en el sitio web del proyecto durante la fase site.

- El **sitio web** de nuestro proyecto se genera en:
  - `target/site/` (y se accede a él desde target/site/index.html)
- Se generan dos informes de cobertura:
  - cobertura y cobertura-integration-test (con los tests unitarios y de integración, respectivamente. Ambos en formato html)

Informe generado por Cobertura

- Independientemente de si hemos generado el informe con cualquiera de las dos formas anteriores, si abrimos el fichero: **target/site/cobertura/index.html**, veremos lo siguiente:

Informe del **PROYECTO**

Packages

All
ppss

All Packages

Classes

App *(12%)*

Coverage Report - All Packages

Package	# Classes	Line Coverage		Branch Coverage		Complexity
All Packages	1	12%	1/8	0%	0/4	5
ppss	1	12%	1/8	0%	0/4	5

Report generated by Cobertura 2.1.1 on 26/04/19 23:39.

Informe del paquete **ppss**

Packages

All
ppss

ppss

Classes

App *(12%)*

Coverage Report - ppss

Package	# Classes	Line Coverage		Branch Coverage		Complexity
ppss	1	12%	1/8	0%	0/4	5

Classes in this Package	Line Coverage		Branch Coverage		Complexity
App	12%	1/8	0%	0/4	5

Report generated by Cobertura 2.1.1 on 26/04/19 23:39.

Informe inicial para la clase App

- El informe generado para la clase App.java muestra lo siguiente:

Coverage Report - ppss.App

Classes in this File	Line Coverage		Branch Coverage		Complexity
App	12%	1/8	0%	0/4	5

```
1        package ppss;
2
3
4   1    public class App {
5
6          public int two_ifs(int i) {
7   0        if (i<20) {
8   0          System.out.println("Soy un numero menor que 20");
9   0          return 1;
10           }
11  0        if (i % 2 == 0) {
12  0          System.out.println("Soy numero par");
13  0          return 2;
14           }
15  0      return 0;
16         }
17     }
```

Report generated by Cobertura 2.1.1 on 27/04/19 16:17.

Estamos probando un 12% del código

Sólo se ejecuta esta línea

Hay 8 líneas ejecutables: 4, 7, 8, 9, 11, 12, 13, 15

Modificamos nuestras pruebas

- Editamos nuestras pruebas y añadimos la llamada al método two_ifs() con un número impar como parámetro:

Nuevo código de pruebas

```
6   public class AppTest {
7
8           @Test
9           public void testSimpleTwo_ifs() {
10                  App app = new App();
11                  //Ejecutamos un metodo de la clase
12                  int n = app.two_ifs(11);
13                  Assert.assertEquals(1,n);
14          }
15  }
```

- Inicializamos los contadores y volvemos a generar el informe:
  - `mvn cobertura:clean site`
- PREGUNTA: ¿Qué pasa si no ejecutamos "cobertura:clean"?
- PREGUNTA: ¿Podemos utilizar "clean" en lugar de "cobertura:clean"? ¿Cuál es la diferencia?

Nuevo Informe generado por Cobertura

- Si abrimos el fichero: **target/site/cobertura/index.html**, veremos lo siguiente:

Informe del **PROYECTO**

Packages

All
ppss

All Packages

Classes

App *(50%)*

Coverage Report - All Packages

Package	# Classes	Line Coverage		Branch Coverage		Complexity
All Packages	1	50%	4/8	25%	1/4	5
ppss	1	50%	4/8	25%	1/4	5

Report generated by Cobertura 2.1.1 on 27/04/19 16:37.

Informe del paquete **ppss**

Coverage Report - ppss

Package	# Classes	Line Coverage		Branch Coverage		Complexity
ppss	1	50%	4/8	25%	1/4	5

Classes in this Package	Line Coverage		Branch Coverage		Complexity
App	50%	4/8	25%	1/4	5

Report generated by Cobertura 2.1.1 on 27/04/19 16:37.

Nuevo informe para la clase App

- El informe generado para la clase App.java muestra lo siguiente:

Coverage Report - ppss.App

Classes in this File	Line Coverage	Branch Coverage	Complexity
App	50% 4/8	25% 1/4	5

```
1      package ppss;
2
3
4   1  public class App {
5
6        public int two_ifs(int i) {
7   1        if (i<20) {
8   1          System.out.println("Soy un numero menor que 20");
9   1          return 1;
10         }
11  0        if (i % 2 == 0) {
12  0          System.out.println("Soy numero par");
13  0          return 2;
14         }
15  0      return 0;
16       }
17     }
```

Line 7: Conditional coverage 50% (1/2).

Report generated by Cobertura 2.1.1 on 27/04/19 16:37.

Hemos probado el 50% del código

Hemos cubierto el 25% de las ramas posibles

La línea 7 se muestra en "rojo" porque la cobertura de ramas no es del 100%

Configuración de la Instrumentación

- Por defecto NO se instrumentan las clases de test. Podemos cambiar el comportamiento por defecto configurando la instrumentación en el plugin de cobertura (dentro de la etiqueta <build> del pom)

Configuración del proyecto

```
<build>
  <plugins>
      ...
      <plugin>
          <groupId>org.codehaus.mojo</groupId>
          <artifactId>cobertura-maven-plugin</artifactId>
          <version>2.7</version>
          <configuration>
              <instrumentation>
                  <ignores>
                      <ignore>com.example.boringcode.*</ignore>
                  </ignores>
                  <excludes>
                      <exclude>com/example/dullcode/**/*.class</exclude>
                      <exclude>com/example/**/*Test.class</exclude>
                  </excludes>
              </instrumentation>
          </configuration>
          <executions>
              <execution>
                  <goals>
                      <goal>clean</goal>
                  </goals>
              </execution>
          </executions>
      </plugin>
  </plugins>
</build>
```

Se IGNORAN las llamadas a estos métodos

Estas clases NO serán instrumentadas

Forzar un mínimo de cobertura

- Cobertura puede utilizarse no sólo para obtener informes, sino que puede integrarase en el ciclo de desarrollo, de forma que si no cubrimos nuestro código mínimamente por las pruebas, podemos impedir que se genere una versión del proyecto (se detiene la construcción)
- Para ello tendremos que configurar la ejecución de la goal check (o check-integration-test) del plugin en el fichero pom.xml
  - Estas goals están asociadas por defecto a la fase **verify**
- Podemos configurar, entre otros, los siguientes parámetros:
  - branchRate, lineRate : nivel de clase
  - packageBranchRate, packageLineRate : nivel de paquete
  - totalBranchRate, totalLineRate: nivel de proyecto
  - haltOnFailure: valor por defecto: true

Configuración del proyecto

```
<project>
  ...
  <build>
    <plugins>
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>cobertura-maven-plugin</artifactId>
        <version>2.7</version>
        <configuration>
          <check>
            <branchRate>85</branchRate>
            <lineRate>85</lineRate>
            <haltOnFailure>true</haltOnFailure>
            <totalBranchRate>85</totalBranchRate>
            <totalLineRate>85</totalLineRate>
            <packageLineRate>85</packageLineRate>
            <packageBranchRate>85</packageBranchRate>
          </check>
        </configuration>
        <executions>
          <execution>
              <goals>
                 <goal>clean</goal>
                <goal>check</goal>
              </goals>
            </execution>
          </executions>
      </plugin>
    </plugins>
  </build>
  ...
</project>
```

mvn verify

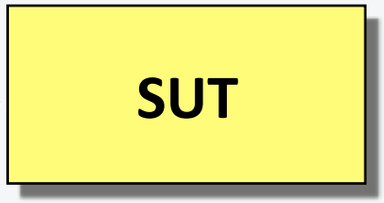
Y AHORA VAMOS AL LABORATORIO...

Practicaremos la generación y análisis de informes de cobertura de nuestros tests

Cobertura: instrumenta el código, ejecuta los drivers y genera un informe

Camino	Datos Entrada	Resultado Esperado	Resultado Real
C1	d1=... d2=... ...	r1	
..			
CM	d1=... d2=... ...	rM	

instrumentamos el código (.class)



Informe de cobertura de la ejecución de los tests

Packages

All
org.apache.maven.shared.i
org.apache.maven.shared.i
org.apache.maven.shared.i
org.apache.maven.shared.i

org.apache.maven.s

Classes

DefaultDownloadManager (
DownloadFailedException (
DownloadManager *(N/A)*

Coverage Report - org.apache.maven.shared.io.download

Package	# Classes	Line Coverage		Branch Coverage		Complexity
org.apache.maven.shared.io.download	3	95%	62/65	100%	6/6	3.556

Classes in this Package	Line Coverage		Branch Coverage		Complexity
DefaultDownloadManager	95%	55/58	100%	6/6	6.75
DownloadFailedException	100%	7/7	N/A	N/A	1
DownloadManager	N/A	N/A	N/A	N/A	1

Report generated by Cobertura 1.8 on 8/22/06 8:30 AM.

Referencias bibliográficas

- "So you think you're covered?" (Keith Gregory)
  - http://www.kdgregory.com/index.php?page=junit.coverage
- Pragmatic Software Testing. Rex Black. John Wiley & Sons. 2007
  - Capítulo 21
- A practitioner's guide to software test design. Lee Coopeland. Artech House. 2004
  - Capítulo 10