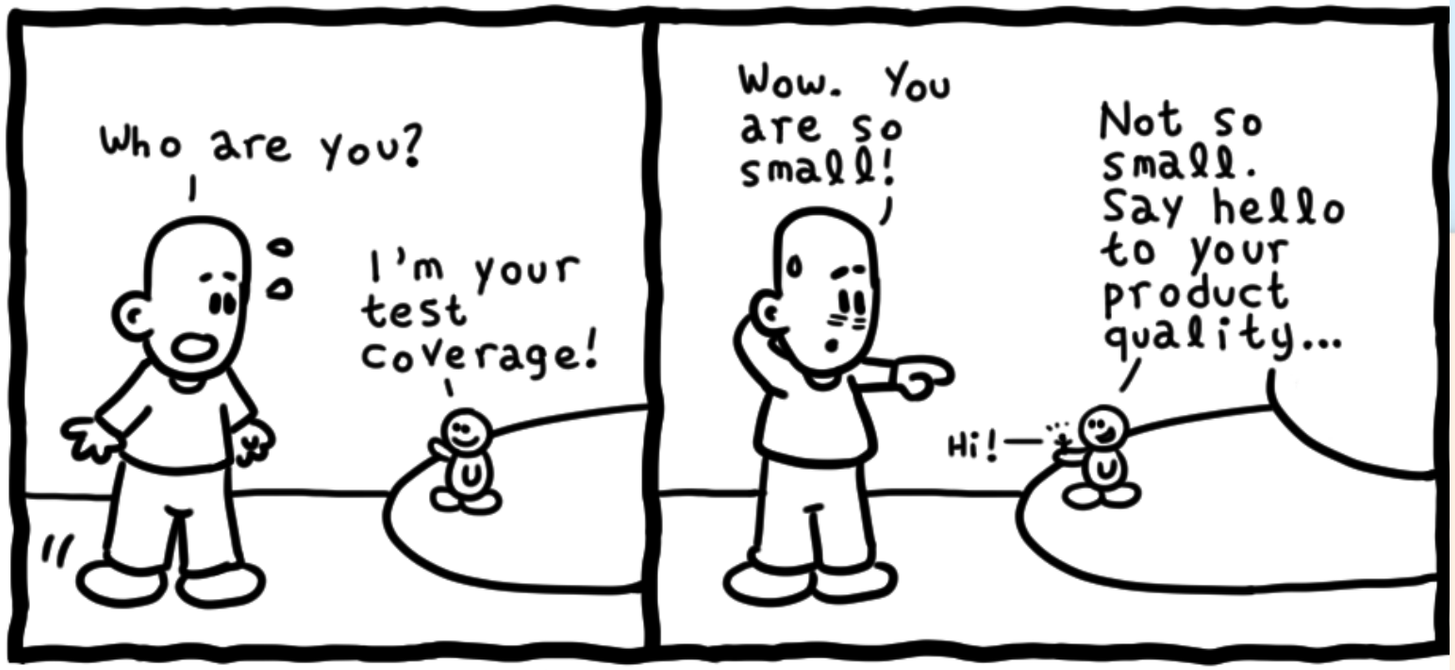
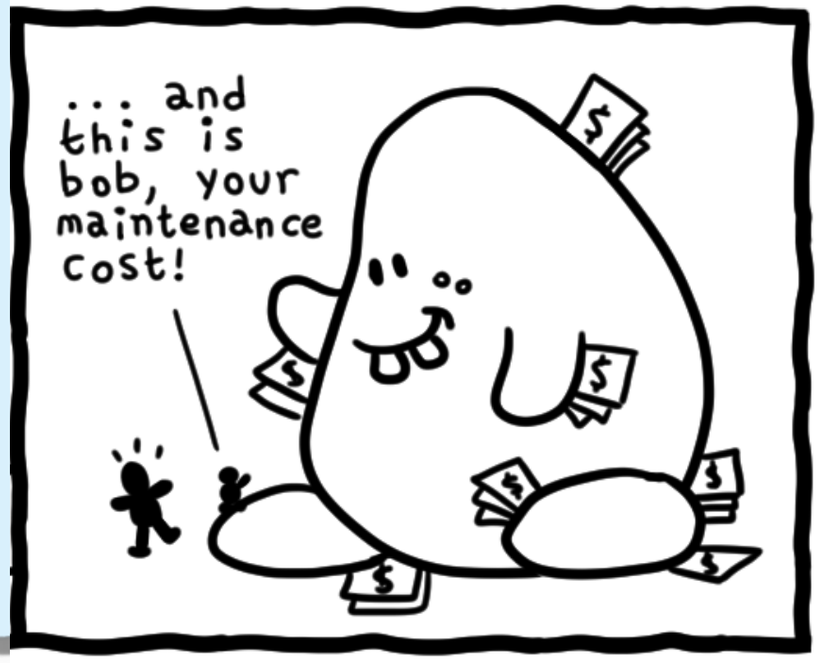
PpSs Planificación y Pruebas de Sistemas Software

Curso 2021-22





Sesión S10: Análisis de pruebas

Análisis de pruebas

- Uso de métricas para el análisis de pruebas
- Cobertura de código
  - ➡ Formas de cuantificar la cobertura de código
  - ➡ Niveles de cobertura
- Herramienta automática de análisis: JaCoCo
  - ➡ Integración con Maven
  - ➡ Generación de informes

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 sobre las pruebas realizadas?
- ¿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,...

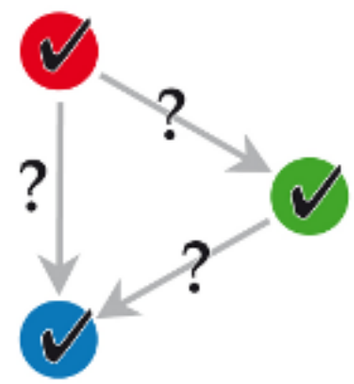
Análisis de las pruebas

- 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

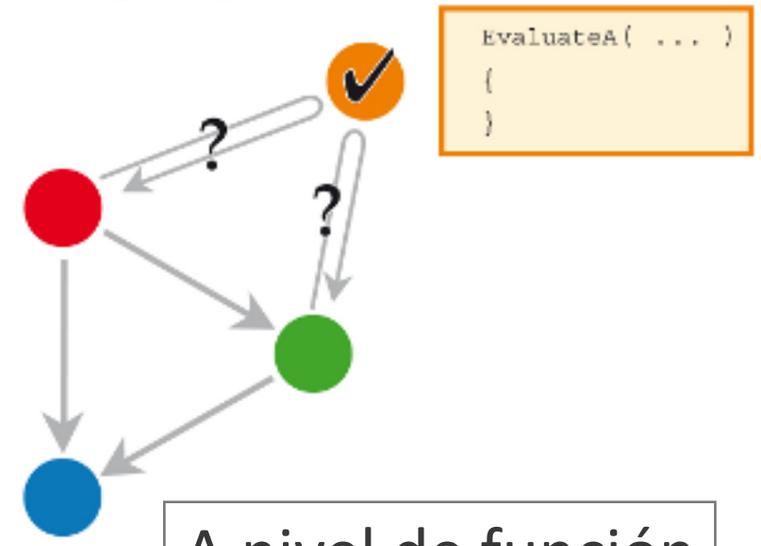
Cobertura de código

- 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



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
- 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

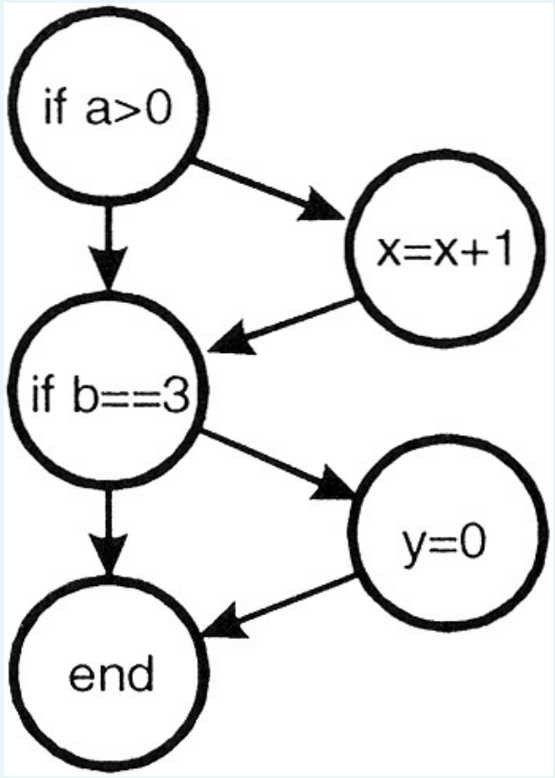
Niveles de cobertura de código

statement coverage

Statement Coverage - Georgia Tech - Software Development Process - Youtube

- **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]

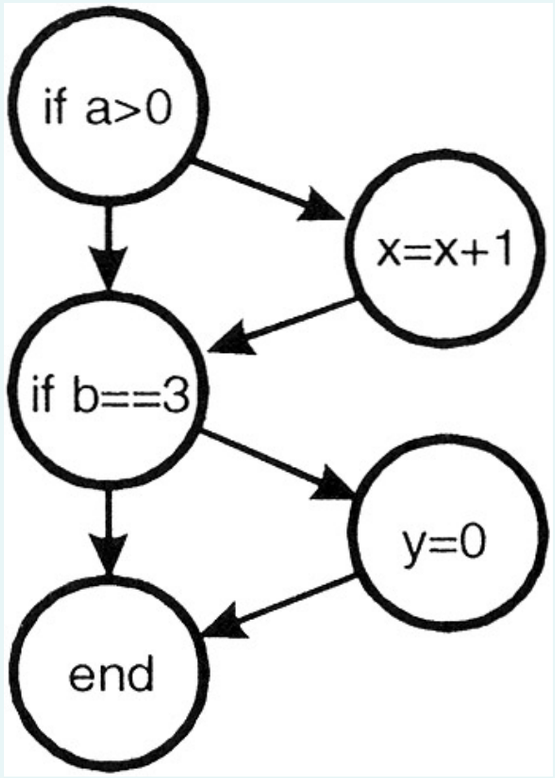
Niveles de cobertura de código

branch (decision) coverage

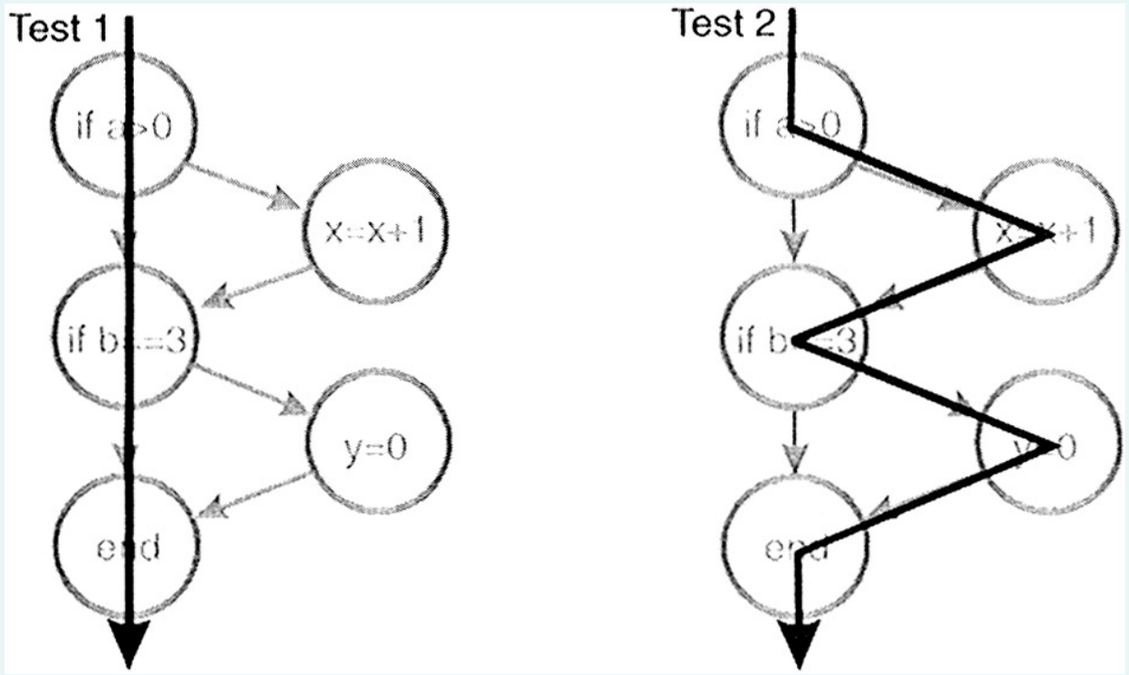
Branch Coverage - Georgia Tech - Software Development Process - Youtube

- **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

Condition Coverage - Georgia Tech - Software Development Process - Youtube

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)

- 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

Branch and Condition Coverage - Georgia Tech - Software Development Process - Youtube

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

```
if(x & y) {
   sentenciaCond;
}
```

- 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

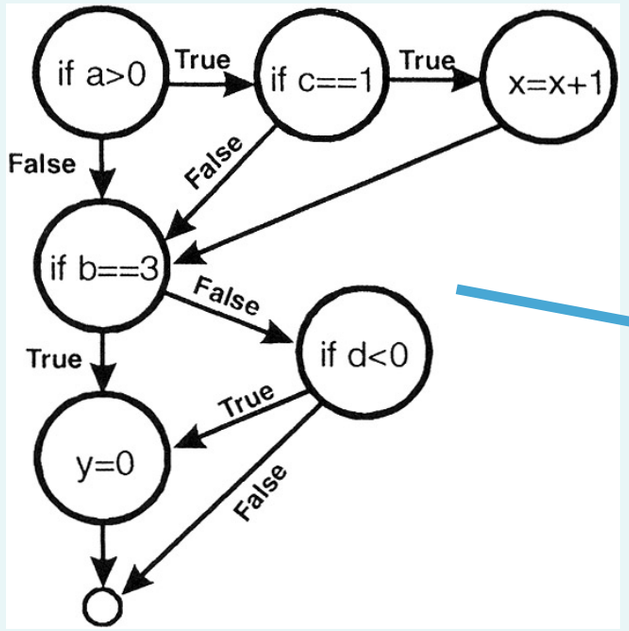
multicondition coverage

NIVEL 5: cobertura de condiciones múltiples (100%)

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

- 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

operador && de java



- 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

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))

Test Criteria Subsumption - Georgia Tech - Software Development Process - Youtube

Other Criteria - Georgia Tech - Software Development Process - Youtube

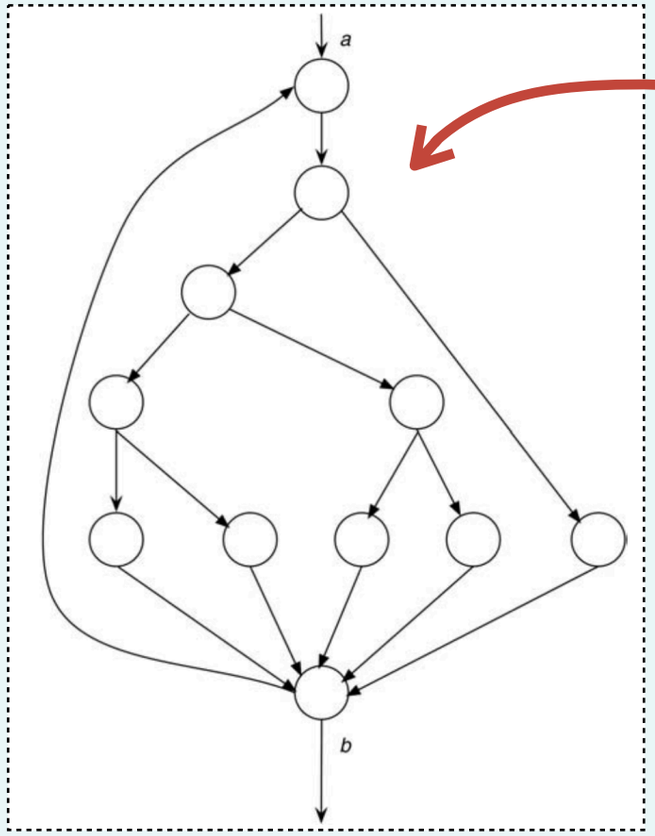
Niveles de cobertura de código

path coverage

Path Coverage - Software Testing - Youtube

- **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

p.ej. para un código con 10 ifs (sin bucles), necesitaríamos $2^{10} = 1024$ tests!!!)



Para este ejemplo con un bucle entre 1 y 20, necesitaríamos $5^{20} + 5^{19} + \ldots + 5^{1}$, donde 5 es el número de camino en el bucle, lo que equivale a 10^{14} 100 trillones de tests!!

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)

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 JaCoCo
  - Ejecución de código en una máquina virtual de Java modificada
- Vamos a ver como ejemplo una herramienta automática que analiza la cobertura de código:
  - JaCoCo

Métricas que calcula JaCoCo

https://www.jacoco.org/jacoco/trunk/doc/counters.html

- JaCoCo instrumenta el bytecode de Java (de forma off-line, y también on-the-fly) de forma que cuando se ejecutan los tests sobre dicho código recopila información sobre la cobertura conseguida
  - Puede utilizarse desde línea de comandos, integrada con Ant, o también con Maven
- JaCoCo genera un informe con las siguiente métricas:
  - **Instructions**
    - Número de instrucciones byte code de Java
  - **Branches** (para cada sentencia if, switch)
    - ¿Se ejecuta cada decisión y condición en sus vertientes verdadera y falsa? CUIDADO!! Jacoco llama branch tanto a una condición como a una decisión. P.ej. Si en un if hay 2 decisiones y 4 condiciones posibles, entonces ese if tiene 6 branches.
    - Las sentencias try…catch no se consideran como condiciones
  - **Complejidad ciclomática** (para cada método no abstracto)
    - ¿Cuántos casos de prueba son necesarios para cubrir todos los caminos linealmente independientes?
  - **Líneas**
    - Una línea se considera ejecutada cuando se ejecuta al menos una de las instrucciones bytecode asociadas a esa línea
  - **Métodos**
    - Cada método no abstracto contiene una instrucción como mínimo. Un método se considera ejecutado, cuando se ejecuta al menos una de sus instrucciones. Los constructores e inicializadores estáticos se cuentan como métodos
  - **Clases**
    - Una clase se considera ejecutada, cuando se ejecuta al menos uno de sus métodos.

Informes que genera JaCoCo

Para cada métrica se muestra la columna "Missed"

Niveles de análisis: proyecto, paquete, clase y método

jacoco-example

jacoco-example

indicación del número de tests que faltan para una cobertura del 100% de condiciones + líneas

Paquete

Element	instructions: Missed Instructions	instructions: Cov.	Branches: Missed Branches	Branches: Cov.	CC: Missed	CC: Cxty	Lines: Missed	Lines: Lines	Methods: Missed	Methods: Methods	Classes: Missed	Classes: Classes
ppss		50%		25%	2	4	4	8	0	2	0	1
Total	11 of 22	50%	3 of 4	25%	2	4	4	8	0	2	0	1

Análisis a nivel de proyecto

jacoco-example > ppss

ppss

Podemos "navegar" por los hiperenlaces

Clase

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
App		50%		25%	2	4	4	8	0	2	0	1
Total	11 of 22	50%	3 of 4	25%	2	4	4	8	0	2	0	1

jacoco-example > ppss > App

App

Método

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
two_ifs(int)		42%		25%	2	3	4	7	0	1
App()		100%		n/a	0	1	0	1	0	1
Total	11 of 22	50%	3 of 4	25%	2	4	4	8	0	2

Informes que genera JaCoCo

Podemos visualizar el código fuente

jacoco-example > ppss > App

App

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
two_ifs(int)		42%		25%	2	3	4	7	0	1
App()		100%		n/a	0	1	0	1	0	1
Total	11 of 22	50%	3 of 4	25%	2	4	4	8	0	2

Constructor por defecto

Resultado de ejecutar `two_ifs(11)`

jacoco-example > ppss > App.java

App.java

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

Se resaltan en VERDE las sentencias ejecutadas

No se han ejecutado todas las "branches" (falta 1)

No se han ejecutado ninguna de las dos"branches"

Se muestran en ROJO las sentencias no ejecutadas

Grafo de flujo de un programa

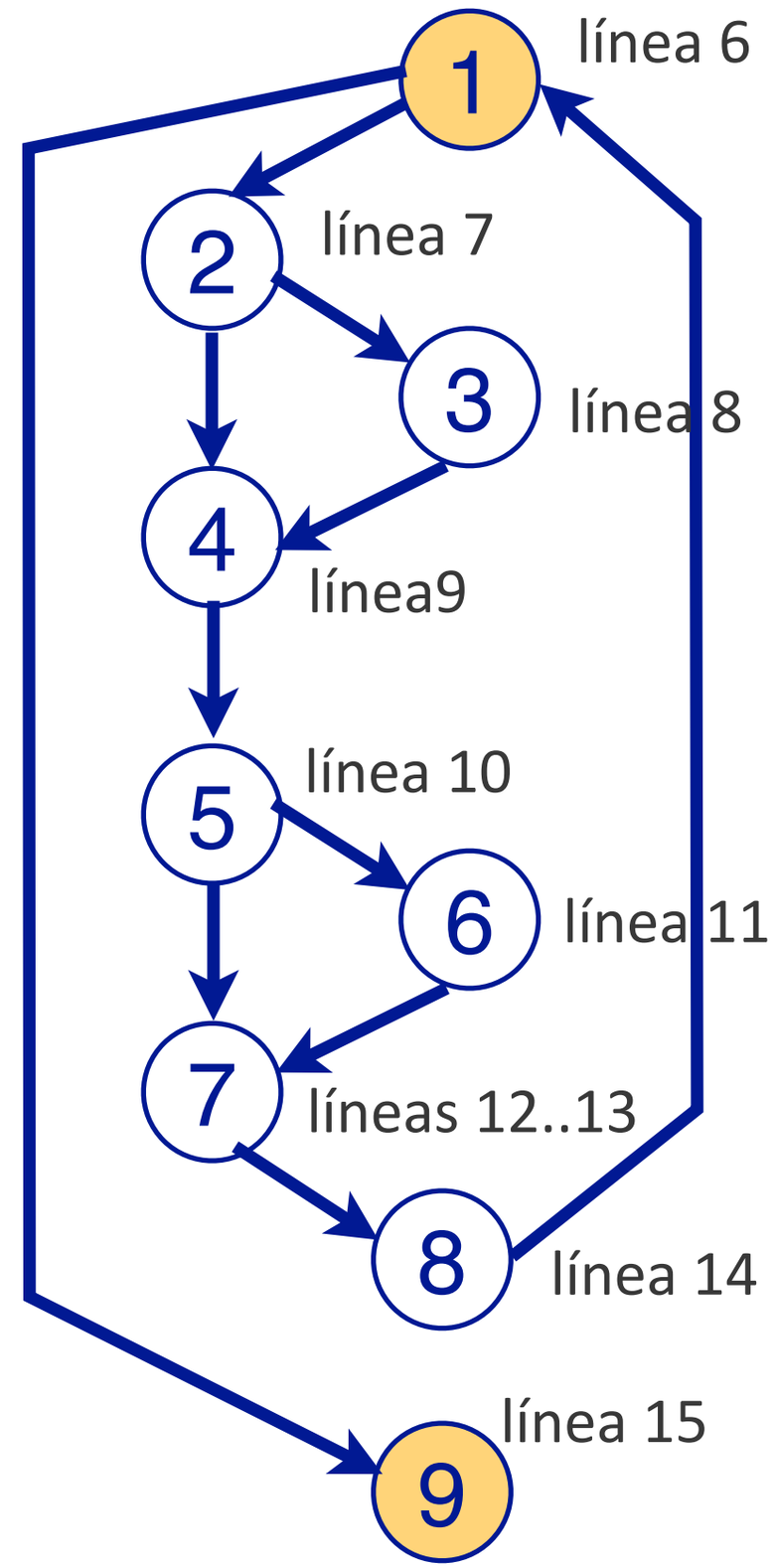
TODAS las sentencias deben estar representadas en el grafo

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

Hay varias formas de calcular la CC

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

- **JaCOCO** calcula la CC utilizando una cuarta alternativa:
  - CC = B −D + 1 = 6 −3 +1 = 4

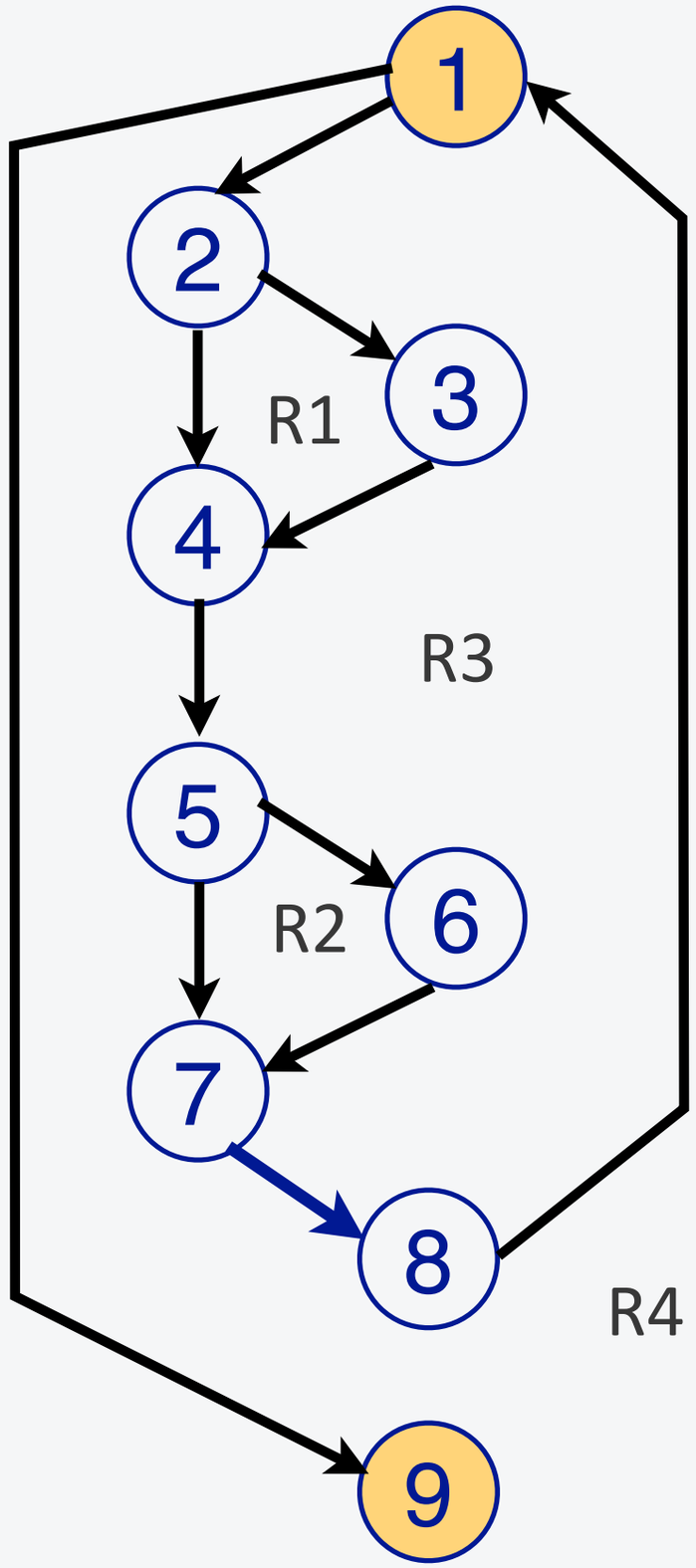
siendo:

B = número de "branches"

Las aristas: 1-9, 1-2, 2-3, 2-4, 5-6, 5-7 son "branches"

D = número de "Decision points"

Los nodos: 1, 2 y 5 son "decision points"

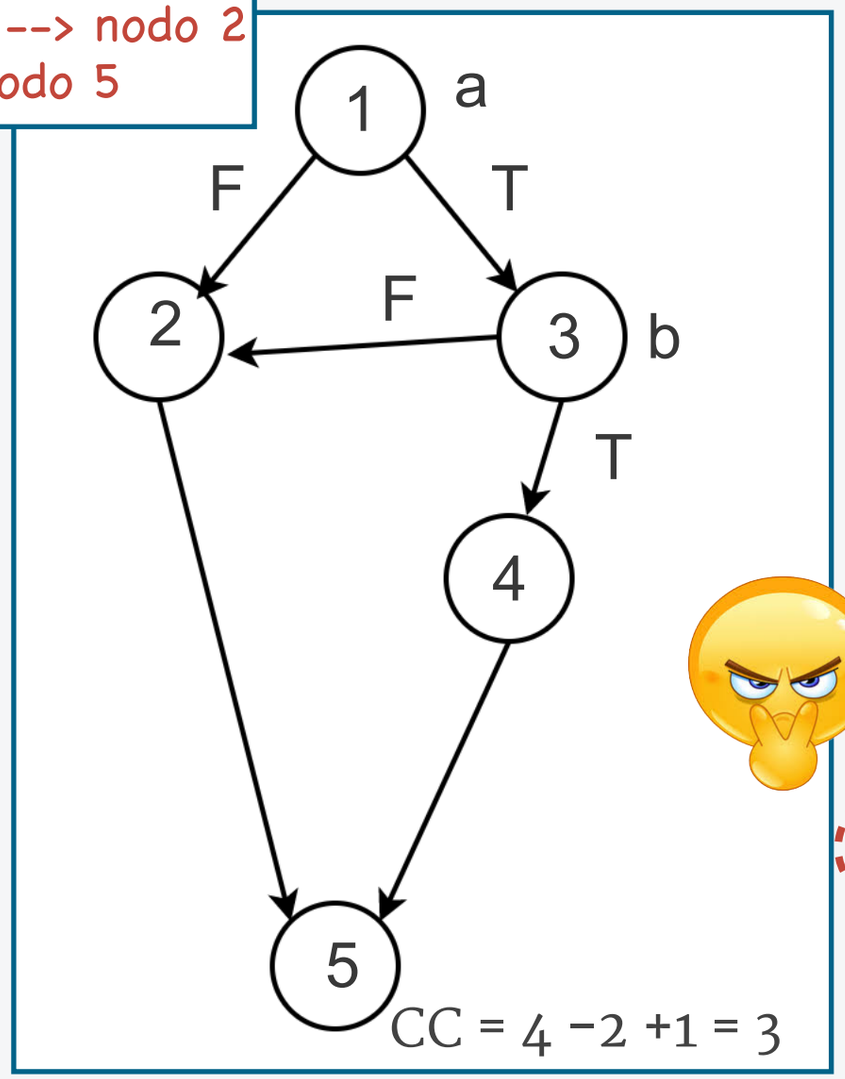


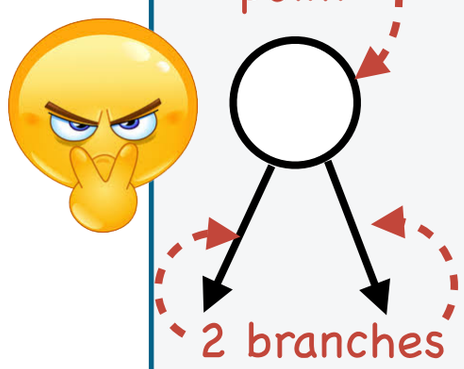
Ejemplos de cálculo de CC con JaCoCo

JaCoCo cuenta Branches y Decision Points para calcular el valor de CC

Operador lógico AND con cortocircuito

```
if (a && b) {
   s1; --> nodo 4
} else {
   s2; --> nodo 2
}--> nodo 5
```

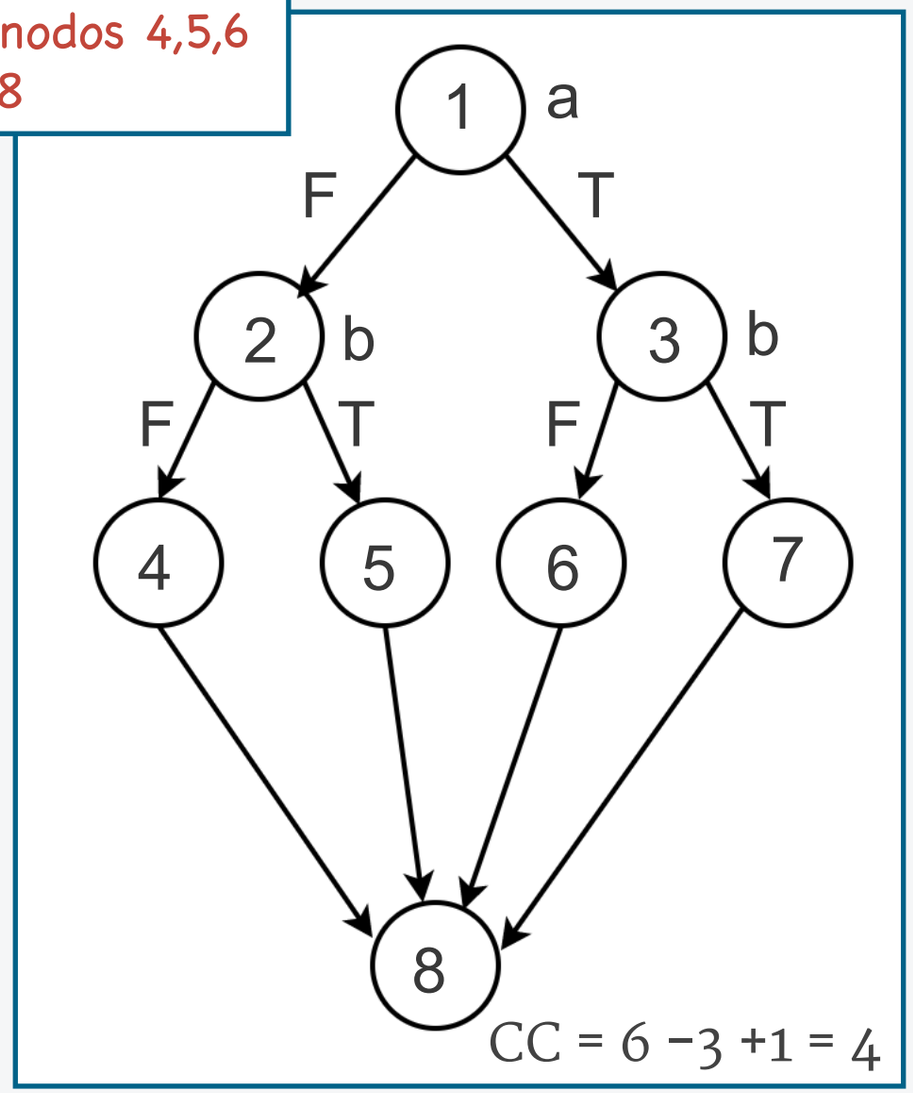




Las aristas: 1−2, 1−3, 3−2, 3−4 son "branches"
Los nodos 1, 3 son "decision points"

Operador lógico AND sin cortocircuito

```
if (a & b) {
   s1; --> nodo 7
} else {
   s2; --> nodos 4,5,6
}--> nodo 8
```



Las aristas: 1−2, 1−3, 2−4, 2−5, 3−6, 3−7 son "branches"
Los nodos 1, 2, 3 son "decision points"

¿Para qué sirve la CC?

CC es una métrica muy útil, no sólo para testing

- 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 JaCoCo calcula de forma automática el valor de CC

Plugin JaCoCo para Maven

https://www.jacoco.org/jacoco/trunk/doc/maven.html

- JaCoCo instrumenta los ficheros .class para obtener los datos de cobertura. La instrumentación tiene lugar "on-the-fly" usando un mecanismo denominado "Java Agent"
  - Los ficheros .class se pre-procesan (instrumentan) cuando la máquina virtual procede a cargar dichas clases.
  - Durante la ejecución de los tests se genera el análisis de cobertura y se guardan en un fichero binario con extensión .exec
- jacoco:prepare-agent
  - Prepara el valor de la propiedad "argLine" para para pasarla como argumento a la JVM durante la ejecución de las pruebas unitarias.
  - Por defecto se asocia a la fase "**initialize**"
  - Entre otras cosas, indica qué clases deben ser instrumentadas, y cuál será el fichero con los datos resultantes del análisis de cobertura (**target/jacoco.exec**)
- jacoco:prepare-agent-integration
  - Es similar a la anterior, pero para las pruebas de integración. Por defecto se asocia a la fase "pre-integration-test", y el fichero con resultados del análisis será **target/jacoco-it.exec**
- jacoco:report
  - A partir de los datos obtenidos con el análisis de cobertura durante la ejecución de las pruebas unitarias (**target/jacoco.exe**), genera un informe en formato html, xml y cvs (en el directorio **target/site/jacoco**). Esta goal está asociada por defecto a la fase "**verify**"
- jacoco:report-integration
  - A partir de los datos obtenidos con el análisis de cobertura durante la ejecución de las pruebas de integración (**target/jacoco-it.exe**), genera un informe en formato html, xml y cvs (en el directorio **target/site/jacoco-it**). Esta goal está asociada por defecto a la fase "**verify**"

Plugin JaCoCo para Maven

- jacoco:check
  - Comprueba si se ha alcanzado un determinado valor de cobertura, y detiene el proceso de construcción si no se alcanza dicho valor
  - Por defecto se asocia a la fase "**verify**", y usa los datos del fichero **jacoco.exec**
  - Es posible configurar la cobertura a diferentes **niveles** (etiqueta **<element>**):
    - BUNDLE, PACKAGE, CLASS, SOURCEFILE o METHOD
    - BUNDLE es a nivel de aplicación, PACKAGE es a nivel de paquete, …
  - Para cada nivel, se puede indicar el **contador** a configurar (etiqueta **<counter>**):
    - INSTRUCTION, LINE, BRANCH, COMPLEXITY, METHOD, CLASS

jacoco-example

jacoco-example

<element>

<counter>

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
ppss		50%		25%	2	4	4	8	0	2	0	1
Total	11 of 22	50%	3 of 4	25%	2	4	4	8	0	2	0	1

<value>

- Para cada contador, se pueden establecer **valores** máximos o mínimos, sobre los diferentes valores calculados (etiqueta **<value>**):
  - TOTALCOUNT, COVEREDCOUNT, MISSEDCOUNT, COVEREDRATIO, MISSEDRATIO

Ejemplo: clase a probar y clase de Test

- Nuestra clase a probar tiene un método denominado two_ifs

```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;
  }
}
```

src/main/java/ppss/App.java

- Escribimos un test unitario para el método two_ifs()

```java
public class AppTest {

        @Test
        public void testSimpleTwo_ifs() {
                App app = new App();
                //Ejecutamos un metodo de la clase
                int n = app.two_ifs(11);
                Assert.assertEquals(1,n);
        }
}
```

src/test/java/ppss/AppTest.java

Análisis de cobertura y generación de informes (I)

```
<plugin>
    <groupId>org.jacoco</groupId>
    <artifactId>jacoco-maven-plugin</artifactId>
    <version>0.8.8</version>

    <executions>
        <execution>
            <id>default-prepare-agent</id>
            <goals>
                <goal>prepare-agent</goal>
            </goals>
        </execution>
        <execution>
            <id>default-report</id>
            <goals>
                <goal>report</goal>
            </goals>
        </execution>
        <execution>
            <id>default-check</id>
            <goals>
                <goal>check</goal>
            </goals>
            <configuration>
                <rules>
                    <rule>
                        <element>BUNDLE</element>
                        <limits>
                            <limit>
                                <counter>COMPLEXITY</counter>
                                <value>COVEREDRATIO</value>
                                <minimum>0.60</minimum>
                            </limit>
                        </limits>
                    </rule>
                </rules>
            </configuration>
        </execution>
    </executions>
</plugin>
```

Preparamos la instrumentación y el análisis de cobertura

Generamos el informe en formato html (a partir del fichero jacoco.exec)

Establecemos unos niveles de cobertura

podemos incluir todas las "reglas" que consideremos necesarias

La construcción se detendrá si, a nivel de proyecto, la complejidad ciclomática no alcanza el 60%

- Comando maven:
  - `mvn verify`
- Obtenemos como resultado:

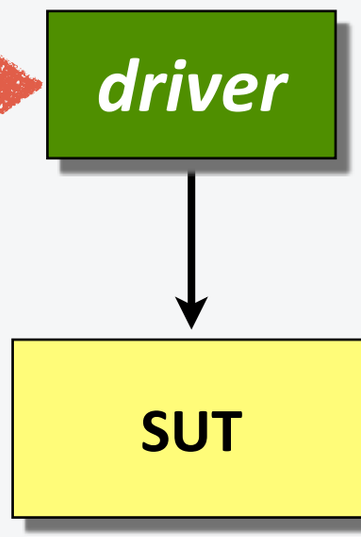
```
[INFO] --- jacoco-maven-plugin:0.8.7:check
(default-check) @ jacoco-example ---
[INFO] Loading execution data file /Users/ppss/
jacoco-example/target/jacoco.exec
[INFO] Analyzed bundle 'jacoco-example' with 1
classes
[WARNING] Rule violated for bundle jacoco-
example: complexity covered ratio is 0.50, but
expected minimum is 0.60
[INFO] ------------------------------------------------------------
[INFO] BUILD FAILURE
[INFO] ------------------------------------------------------------
```

- Hasta que no consigamos los valores de cobertura especificados, nuestra construcción no terminará con éxito.

Y AHORA VAMOS AL LABORATORIO...

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

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



JaCoCo: instrumenta los .class antes de ejecutar los tests.

Analiza la cobertura después de ejecutar los tests y genera un informe

Informe de cobertura

JaCoCo

JaCoCo

instructions | branches | CC | lines | methods | classes

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
org.jacoco.examples		58%		64%	24	53	97	193	19	38	6	12
org.jacoco.core		97%		93%	107	1,388	115	3,347	21	720	2	139
org.jacoco.agent.rt		77%		84%	31	121	62	310	21	74	7	20
jacoco-maven-plugin		90%		81%	35	183	44	407	8	110	0	19
org.jacoco.cli		97%		100%	4	109	10	275	4	74	0	20
org.jacoco.report		99%		99%	4	572	2	1,345	1	371	0	64
org.jacoco.ant		98%		99%	4	163	8	429	3	111	0	19
org.jacoco.agent		86%		75%	2	10	3	27	0	6	0	1
Total	1,355 of 27,352	95%	143 of 2,125	93%	211	2,599	341	6,333	77	1,504	15	294

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
- JaCoCo (https://www.jacoco.org/jacoco/trunk/doc/index.html)