Generación Automática de Tests

Ingeniería del Software 2

Juan P. Galeotti

DEPARTAMENTO DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

ICC Instituto de Ciencias de la Computación

- ANSI/IEEE 1059 standard – "*A process of analyzing a software item to detect the differences between existing and required conditions (i.e., defects) and to evaluate the features of the software item.*"

Software Testing

Expectativa

vs.

Realidad

Resultado

Estímulo

"Tester"

Software


EXPECTATIVA
REALIDAD

Límites del Testing

"Program testing can be used to show the presence of bugs, but never to show their absence!"

Edsger Dijsktra
1930--2002

→ absence cannot be proven

Automatización del Testing de Software

Decide automáticamente si el test pasa o falla

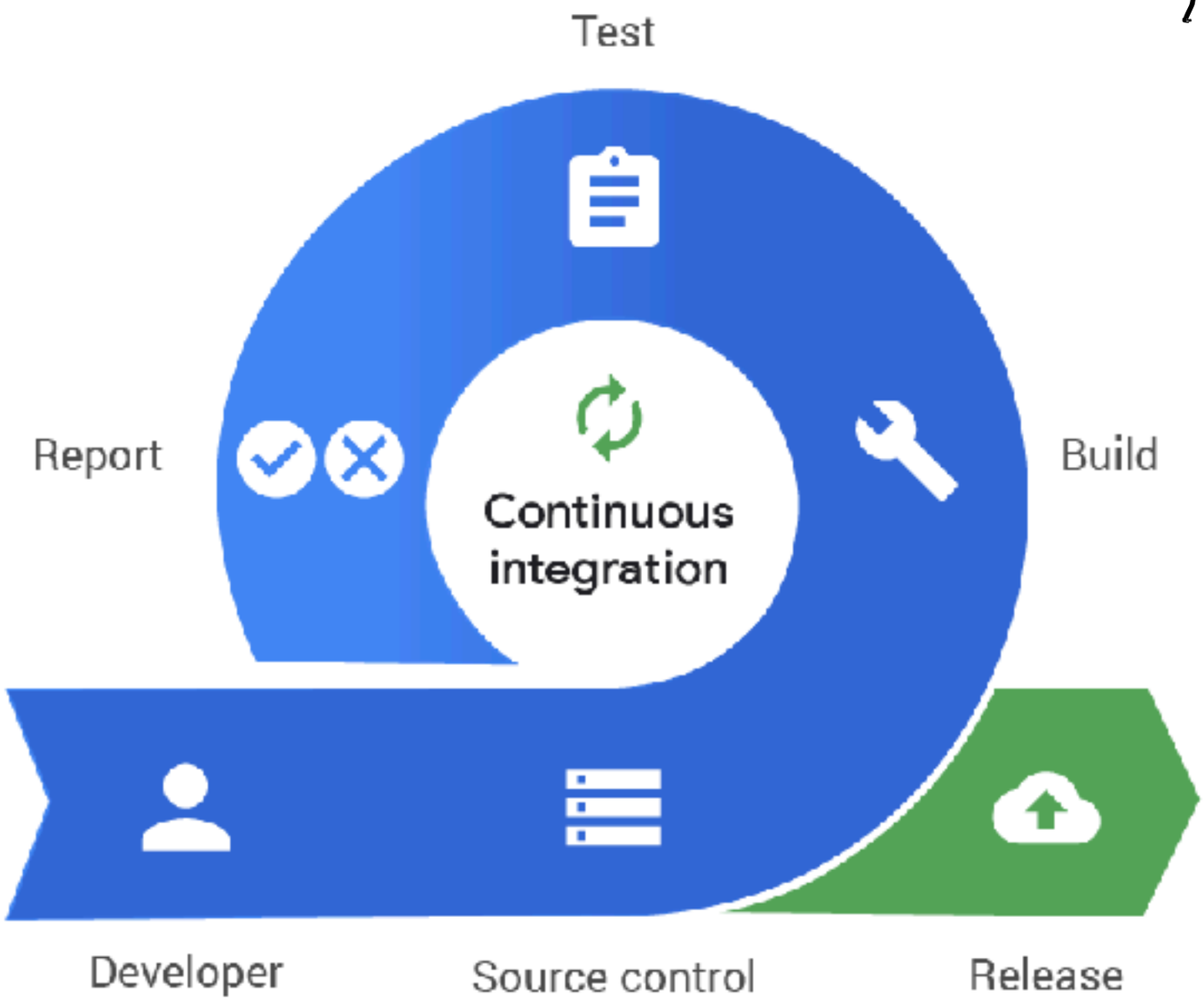
Software bajo Test

Ejecuta

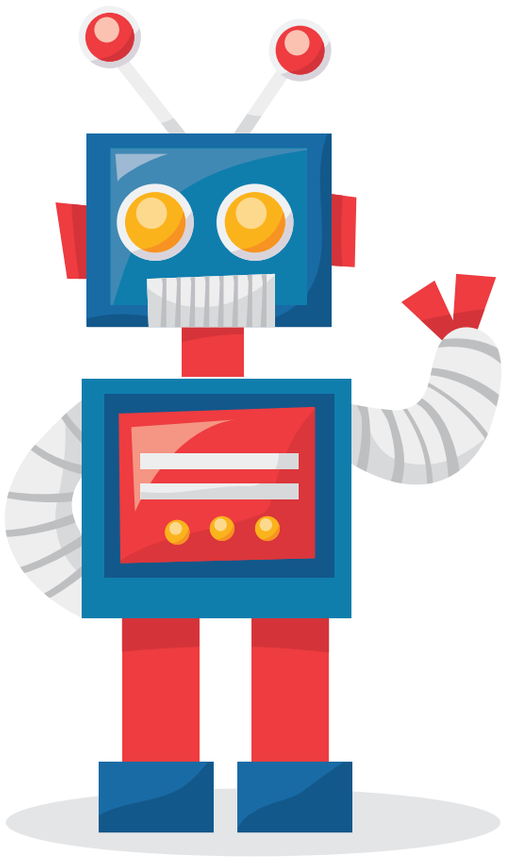
```
@Test
public void test0()  throws Throwable  {
    Foo foo0 = new Foo();
    Bar bar0 = new Bar("baz3");
    bar0.coverMe(foo0);
    assertEquals(0, foo0.getX());
}
```

Tests (i.e. "Test Suite")

Integración Continúa



Ejecución **Automática** de Casos de Tests



JUnit

NUnit

Selenium

```
    @Test
    public void newArrayListsHaveNoElements() {
        assertThat(new ArrayList().size(), is(0));
    }
```

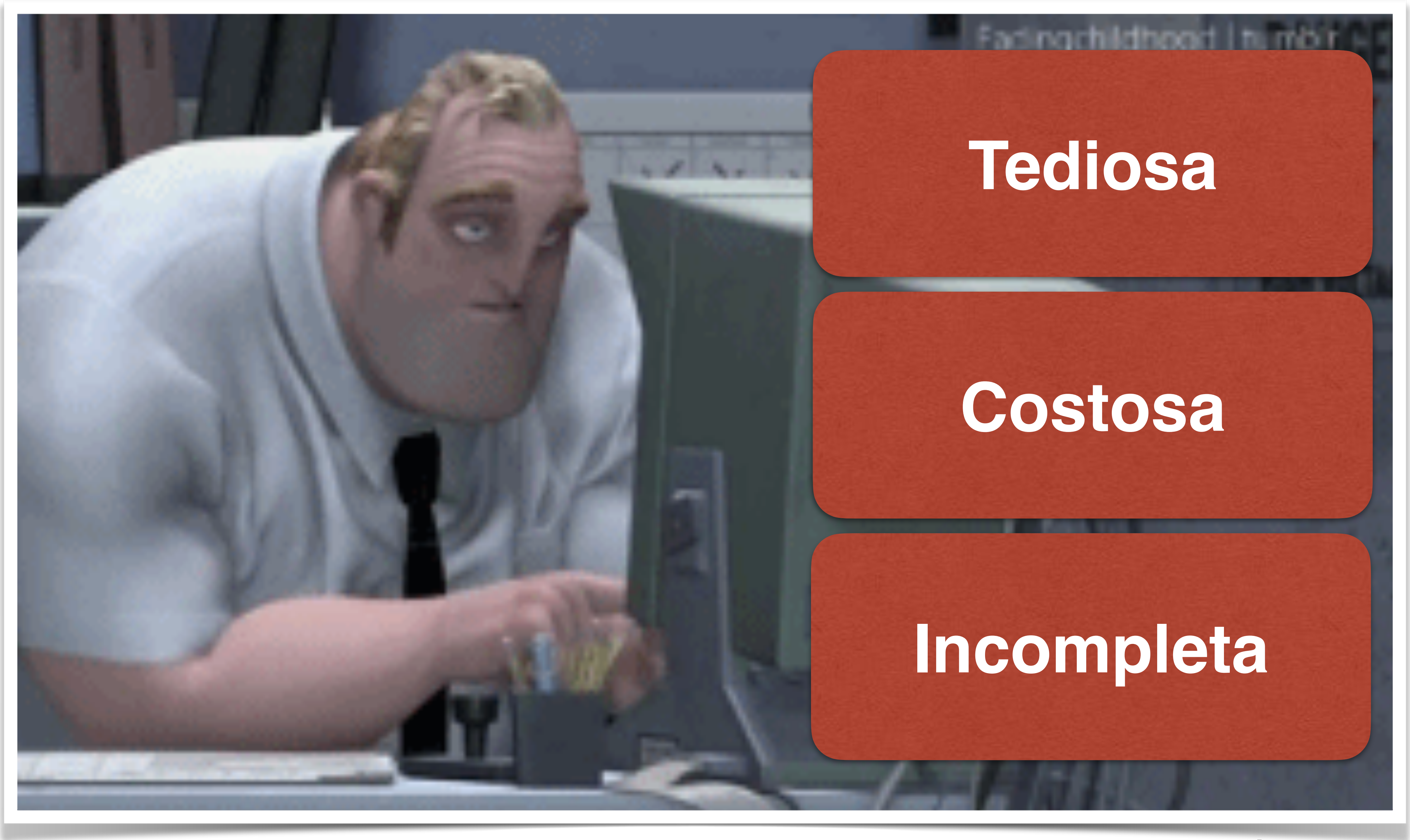
```
    @Test
    public void sizeReturnsNumberOfElements() {
        List instance = new ArrayList();
        instance.add(new Object());
        instance.add(new Object());
        assertThat(instance.size(), is(2));
    }
```

REST-assured

Automatic execution

Manual Creation

Creación **Manual** de Casos de Tests

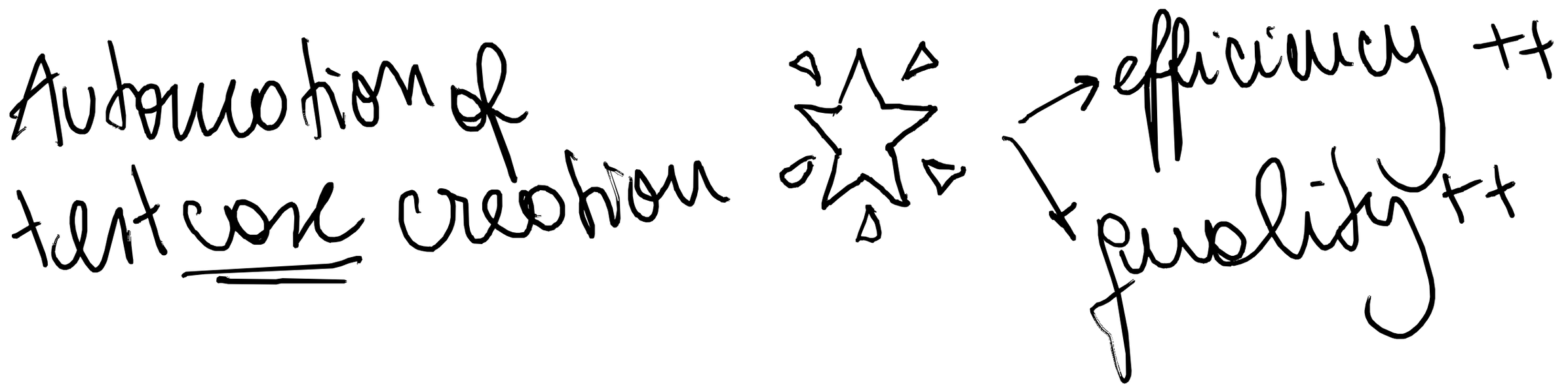


Casos de test → creación manual

Visión

- Aprovechar el actual **poder de cómputo** para
  - Mejorar la **eficiencia** de las personas que programan
  - Aumentar la **calidad** de los programas que escribimos
- ¿Cómo?

Automatizando la <u>creación</u> de los Casos de Tests



Automatic Test Case Generation

Software

Test Suite

Test Generator

```
@Test
public void test0()  throws Throwable
    Foo foo0 = new Foo();
    Bar bar0 = new Bar("baz3");
    bar0.coverMe(foo0);
    assertEquals(0, foo0.getX());
}
```

(software)

how does it work?

AIEE-IRE '62 (Spring) Proceedings of the May 1-3, 1962, spring joint computer conference

A GENERAL TEST DATA GENERATOR FOR COBOL

Lt. Richard L. Sauder

Automation Techniques Branch

Headquarters, Air Force Logistics Command

Wright-Patterson Air Force Base, Ohio

This article discusses the effort being made by the Air Force Logistics Command in developing a method of generating effective program test data. This "Test Data Generator" is designed to operate in conjunction with the COBOL compiler implemented by AFLC. As such, the system not only builds data conforming to descriptions given in the Data Division of a COBOL program but also places in these items necessary data relationships to test the logic of the COBOL program. Both the utilization and the method of operation of the system are discussed in this paper.

Introduction

One of the major underdeveloped areas that

nent criterion is that of controlling the content of these elements.

The method of operation and the information necessary to fulfill these functions will now be described in detail.

Utilization of the COBOL Data Division

Determining the format of data fields results from a thorough interpretation of the Data Division of the COBOL source program being tested. Within this Division, options exist to describe in detail both the structure of every element within a record and the relationships of these elements to one another. Those options

Fault detection

Escenario #1: Detección de fallas

- **Idea**: para poder clasificar una falla necesitamos un **oráculo** ?
- ¿Qué pasa si no tenemos un oráculo?

Oráculos implícitos → general checks on failures/crashes

Robustness

Escenario #2: Testing de Robustez

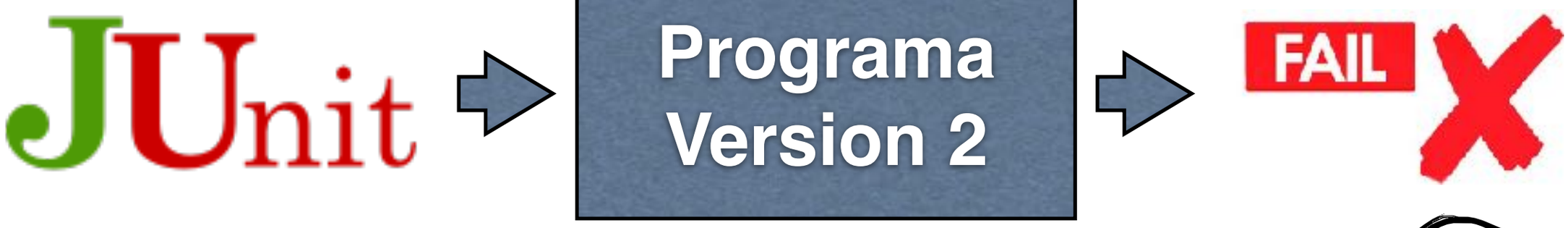
- A veces no disponemos de un oráculo para chequear automáticamente el programa

- Robustness Testing: ¿ocurrió algún crash?
  - ¿Qué es un crash? Oráculos implícitos

oráculo implícito

Regression

Escenario #3: Testing de Regresión



- Regression Testing: ¿hay una diferencia?
  - ¿Es una diferencia esperable?

use tests from 1° code version to test 2° version. acts as oracle.

Niveles de Testing

1. **Test de Sistema**: probar el resultado de usar un sistema. Todo el equipo.
2. **Test de Integración**: probar funcionamiento entre unidades/módulos y programadores
3. **Test de Unidad**: chequear comportamiento de una unidad. Mocks. Único programador.

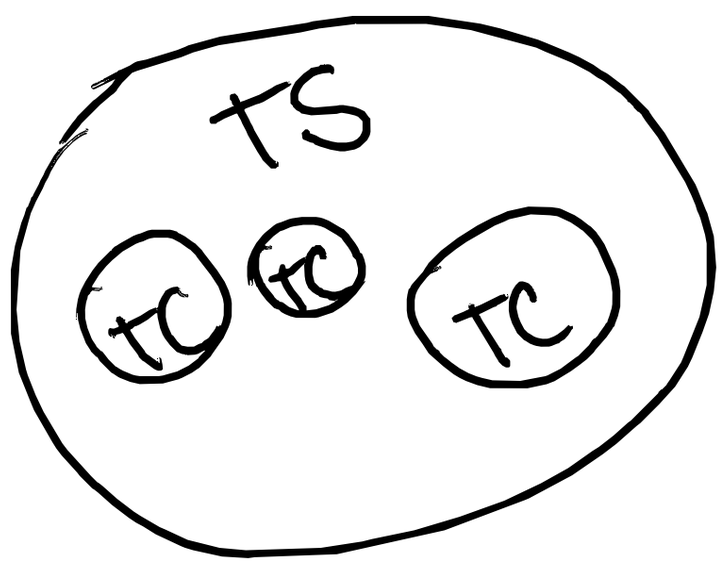
Etapas de un Test

Un test consiste de:

1. el ***setup*** – una etapa de preparación requerida antes de ejercitar el test
2. la ***ejercitación*** – el código que ejercita la funcionalidad bajo test
3. el ***chequeo*** – código que chequea la respuesta contra el resultado esperado (oráculo)

Test Cases vs. Test Suites

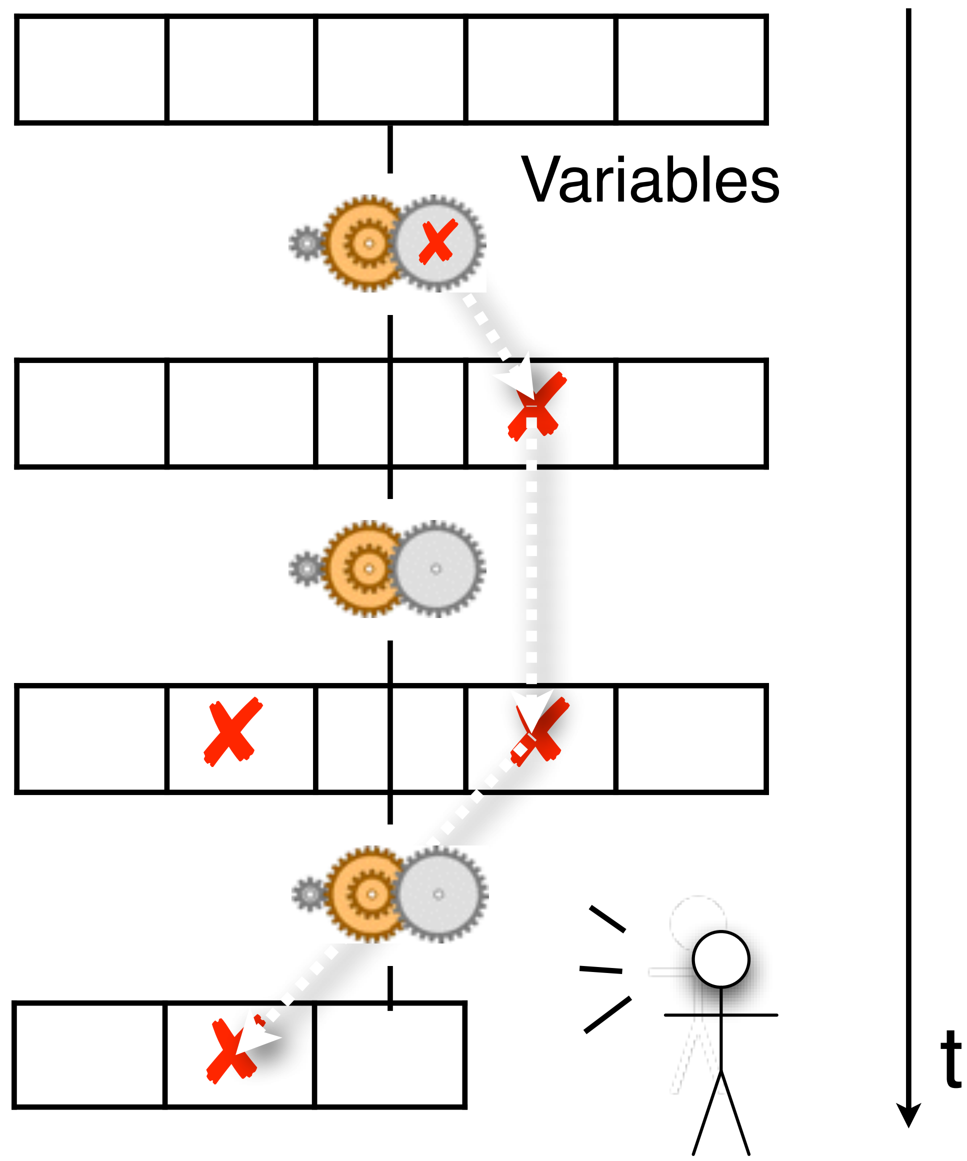
- **Test Case**: compuesto por código para *setup*, *exercise*, *check*
  - requiere de alguna noción de oráculo
- **Test Suite**: un conjunto de test cases
  - sin orden necesariamente



Del Defecto a la Falla

• static
• introduced by programmer
• defect during execution

1. El programador crea un ***defecto*** en el código.
2. Cuando es ejecutado, el defecto crea una ***infección***.
3. La infección se *propaga*.
4. La infección causa una ***falla***.



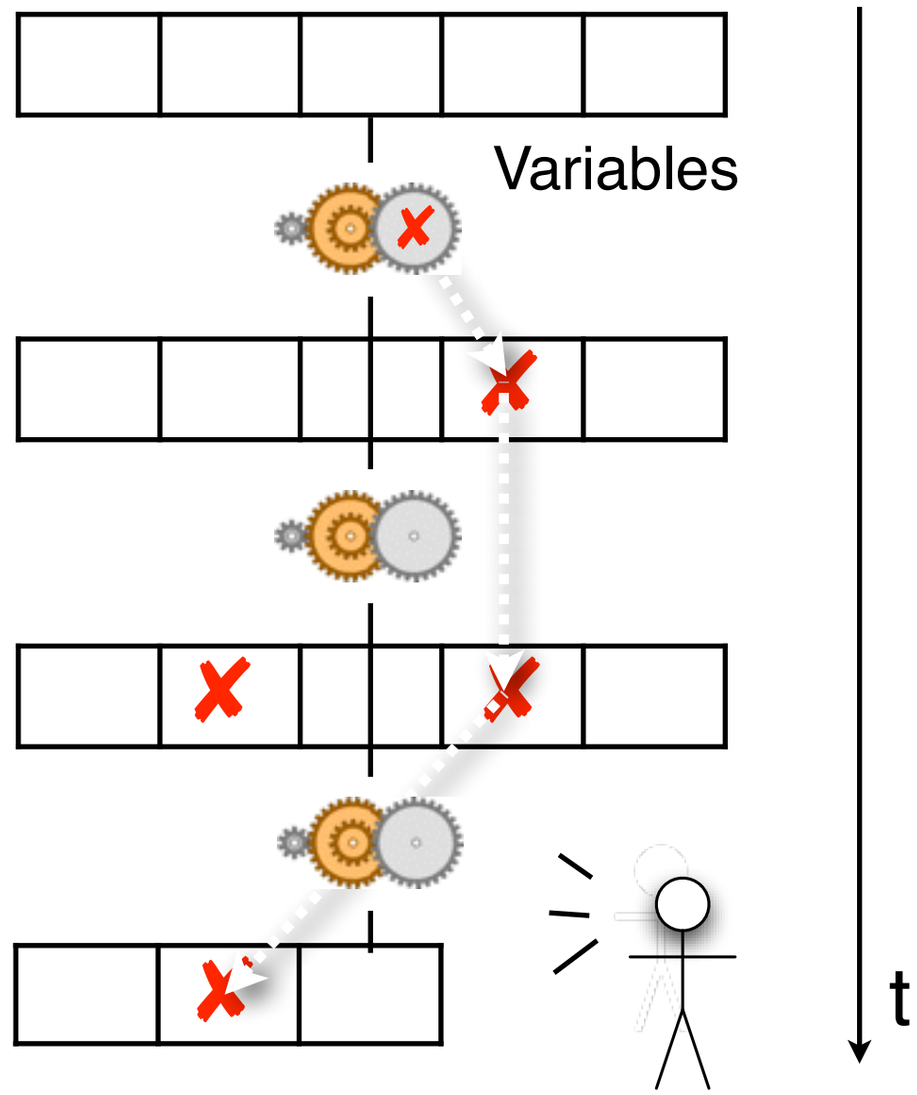
Defecto en código → infecta y propaga ⇒ falla

Todos los Errores

- **Error:** Una desviación no buscada ni intencional de lo que es correcto, esperado o verdadero.
- **Defecto:** Un *error* en el **código** del programa, específicamente uno que puede crear un *infección* (y conducir a una *falla*)
- **Infección:** Un *error* en el **estado** del programa, específicamente uno que puede llevar a una *falla*
- **Falla:** Un error **externamente visible** en el comportamiento del programa.

Los 3 Desafíos del Testing

1. Tenemos que *disparar* el error en cuestión:
   - *ejecutar* el defecto
   - hacer que la infección se *propague*, y
   - resulte en una *falla*.
2. Debemos *reconocer* el error como tal – como una desviación de lo que es correcto, válido, o verdadero.
3. Debemos identificar *funcionalidad faltante*



① Disparar ≈ replicar error → ↑ coverage
② RECONOCER error → oráculo
③ Identificar funcionalidad faltante.
↳ especificación.

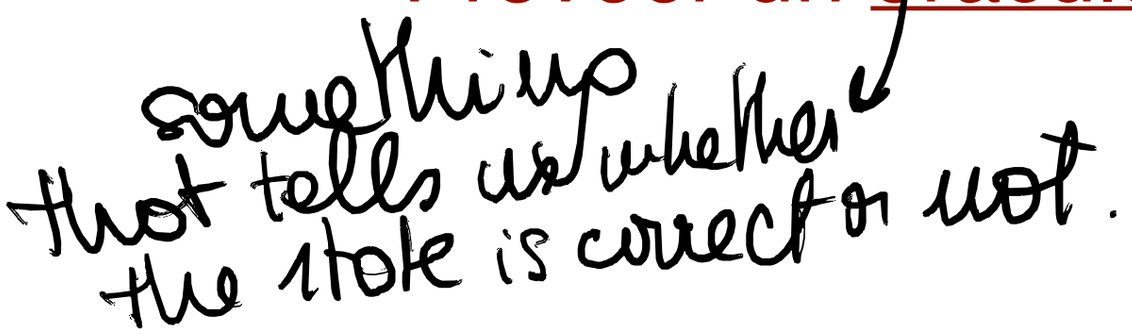
Los 3 Desafíos del Testing

1. Tenemos que *disparar* el error en cuestión:
   - *ejecutar* el defecto
   - hacer que la infección se *propague*, y
   - resulte en una *falla*.

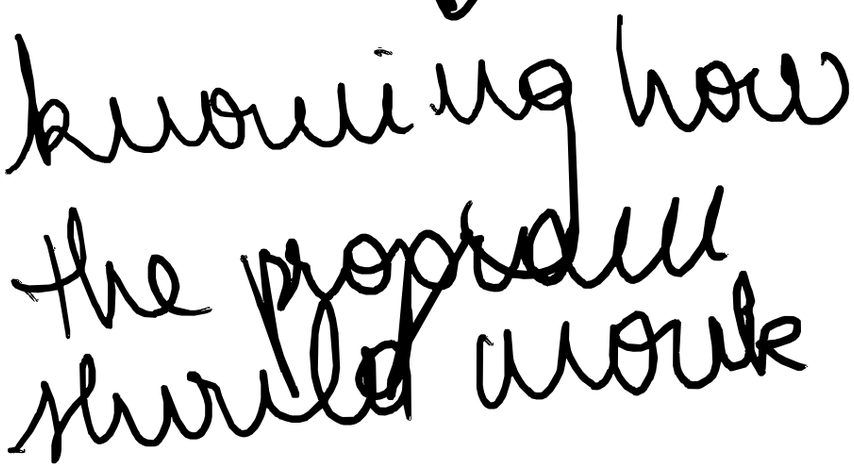
— Cubrir tanto comportamiento como sea posible

2. Debemos *reconocer* el error como tal – como una desviación de lo que es correcto, válido, o verdadero.
3. Debemos identificar *funcionalidad faltante*

— Proveer un oráculo

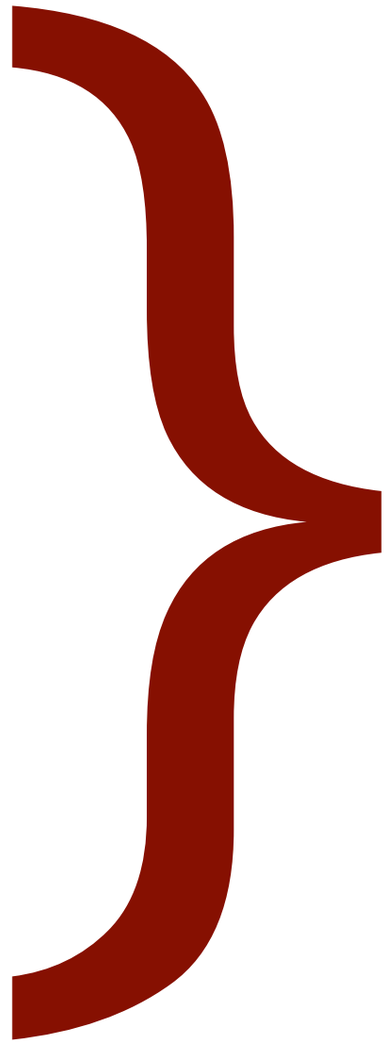


— Tener una especificación



Los 3 Desafíos del Testing

1. Tenemos que *disparar* el error en cuestión:
   - *ejecutar* el defecto
   - hacer que la infección se *propague*, y
   - resulte en una *falla*.
2. Debemos *reconocer* el error como tal – como una desviación de lo que es correcto, válido, o verdadero.
3. Debemos identificar *funcionalidad faltante*



En una forma eficiente (realizable)

Los 3 Desafíos del Testing

1. Tenemos que *disparar* el error en cuestión:
   - *ejecutar* el defecto
   - hacer que la infección se *propague*, y
   - resulte en una *falla*.
2. Debemos *reconocer* el error como tal – como una desviación de lo que es correcto, válido, o verdadero.
3. Debemos identificar *funcionalidad faltante*

— Cubrir tanto comportamiento como sea posible

— Proveer un oráculo

— Tener una especificación

Los 3 Desafíos del Testing

1. Tenemos que *disparar* el error en cuestión:
   - *ejecutar* el defecto
   - hacer que la infección se *propague*, y
   - resulte en una *falla*

— Cubrir tanto comportamiento como sea posible

¿Qué es exactamente el "comportamiento"?

Los 3 Desafíos del Testing

1. Tenemos que *disparar* el error en cuestión:
   - *ejecutar* el defecto
   - hacer que la infección se *propague*, y
   - resulte en una *falla*.

→ debemos *alcanzar* *cada línea del programa*

COVERAGE del comportamiento
↳ reach EVERY LINE of the program.

↗ cuán bueno es mi test suite p/ detectar fallas

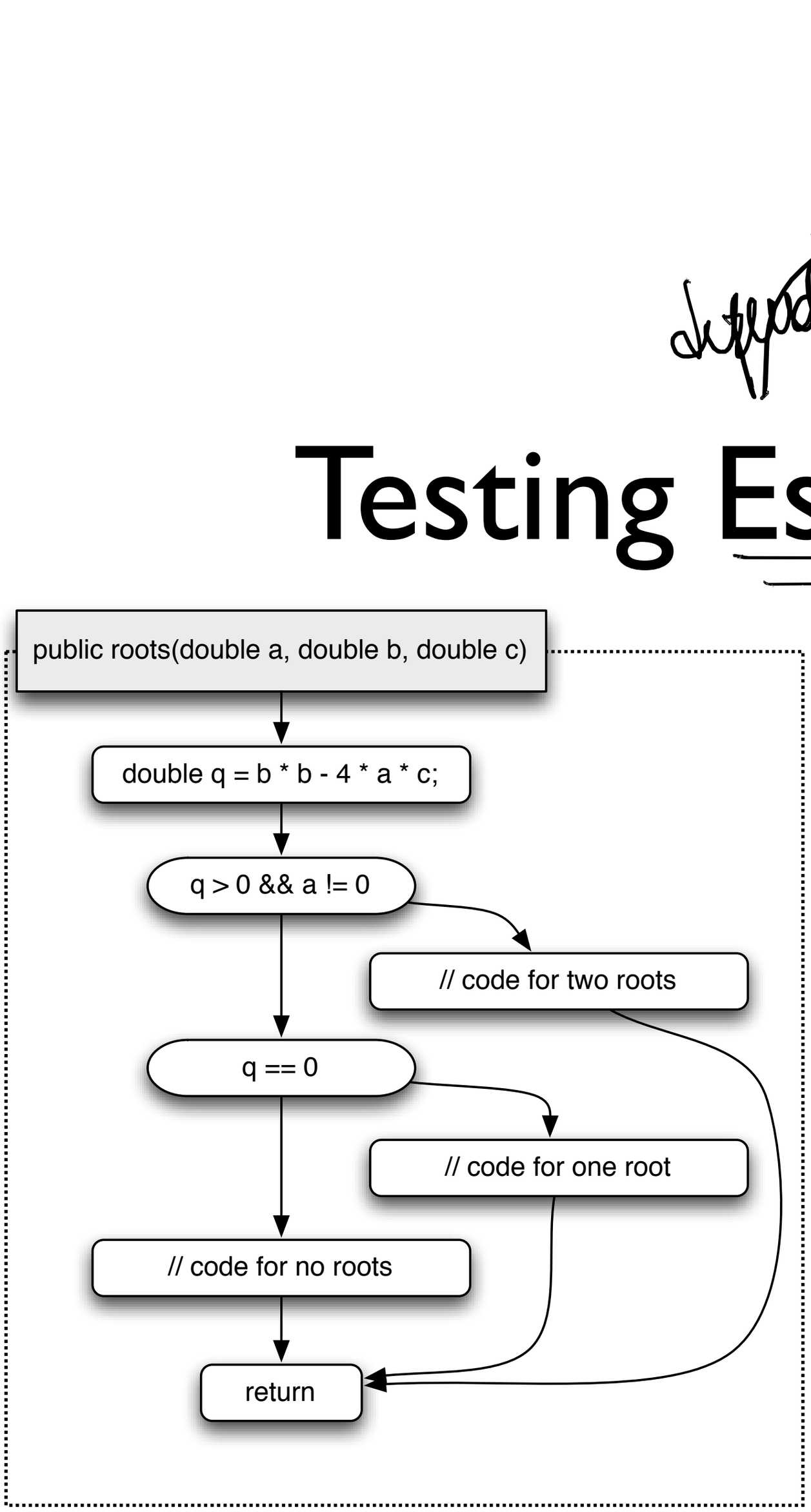
Criterios de Adecuación

= predicado V o F p/ ⟨programa, suite⟩

- ¿Cómo sabemos que una test suite es "suficientemente buena"?
- Un criterio de adecuación de test es un predicade que es verdadero o falso para un par ⟨*programa, test suite*⟩
- Usualmente expresado en forma de una regla – e.g., "todos los statements deben ser ejecutados" } eg.

supone estructura del código que testeo

Testing Estructural



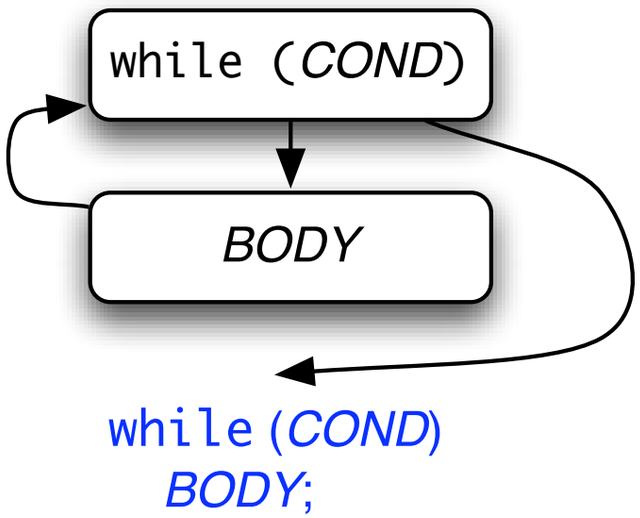
CRITERIO DE ADECUACION.

- El *control flow graph* (CFG) puede servir como un *criterio de adecuación* para test cases
- Cuanto mas "partes" son ejecutadas (cubiertas), mayores las chances de un test de descubrir una falla
- "partes" pueden ser: nodos, ejes, caminos, decisiones…

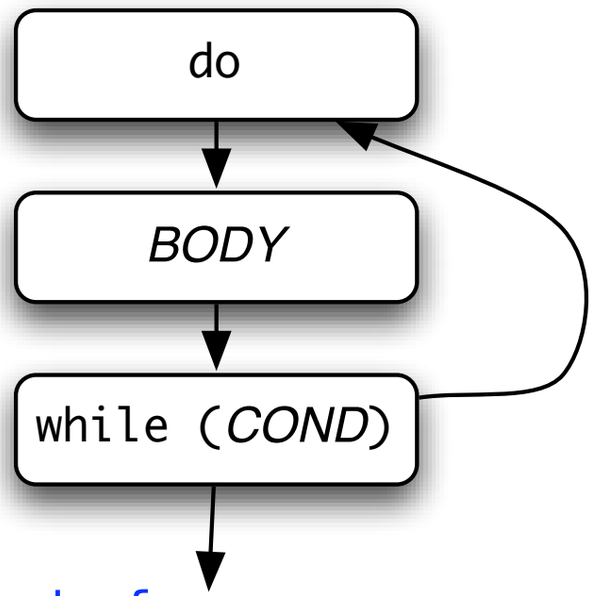
not much to add, good slide :)

WTF son estos control de flujo?

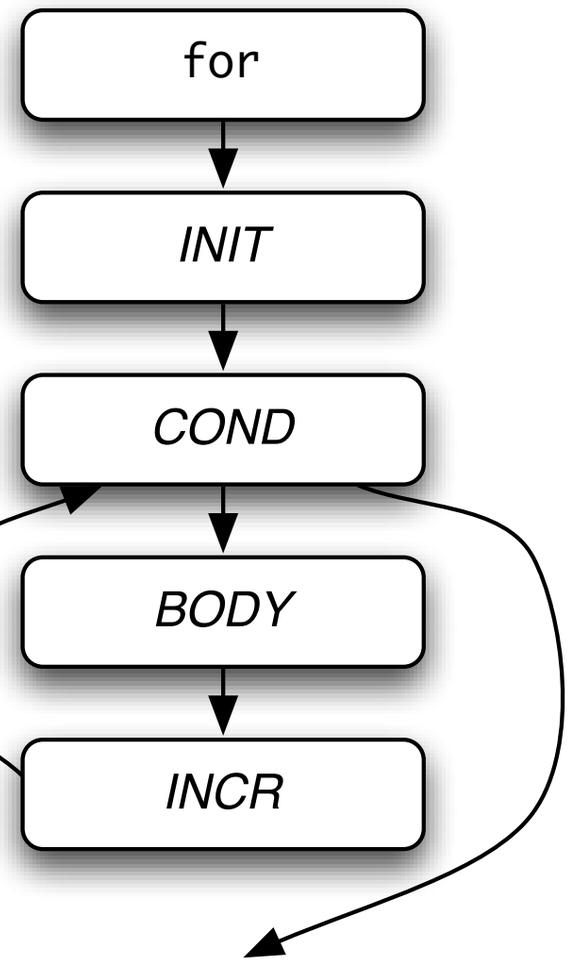
Control Flow Patterns



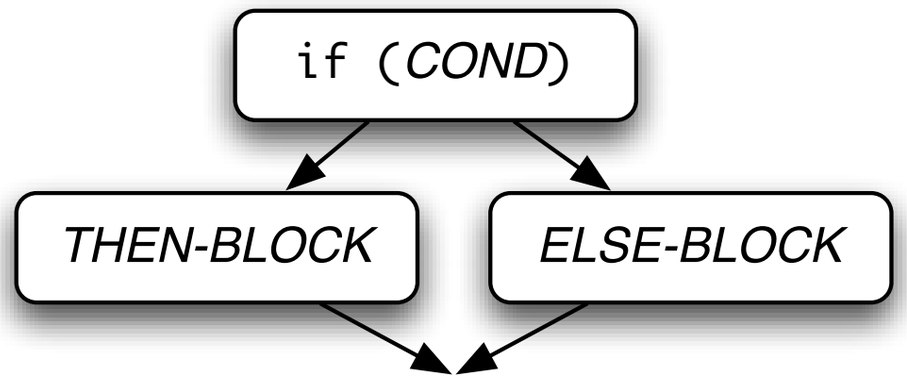

```
while (COND)
   BODY;
```




```
do {
     BODY
}while (COND);
```




```
for (INIT; COND; INCR)
   BODY;
```




```
if (COND)
   THEN-BLOCK;
else
   ELSE-BLOCK;
```

structural

UI

Statement Testing

- Criterio de Adecuación: cada statement (o nodo en el CFG) *debe ser ejecutado al menos una vez*
- Idea: un defecto en un statement sólo puede ser revelado ejecutando el defecto
- Cobertura: $\frac{\text{\# statements ejecutados}}{\text{\# statements}}$

(coverage ratio)

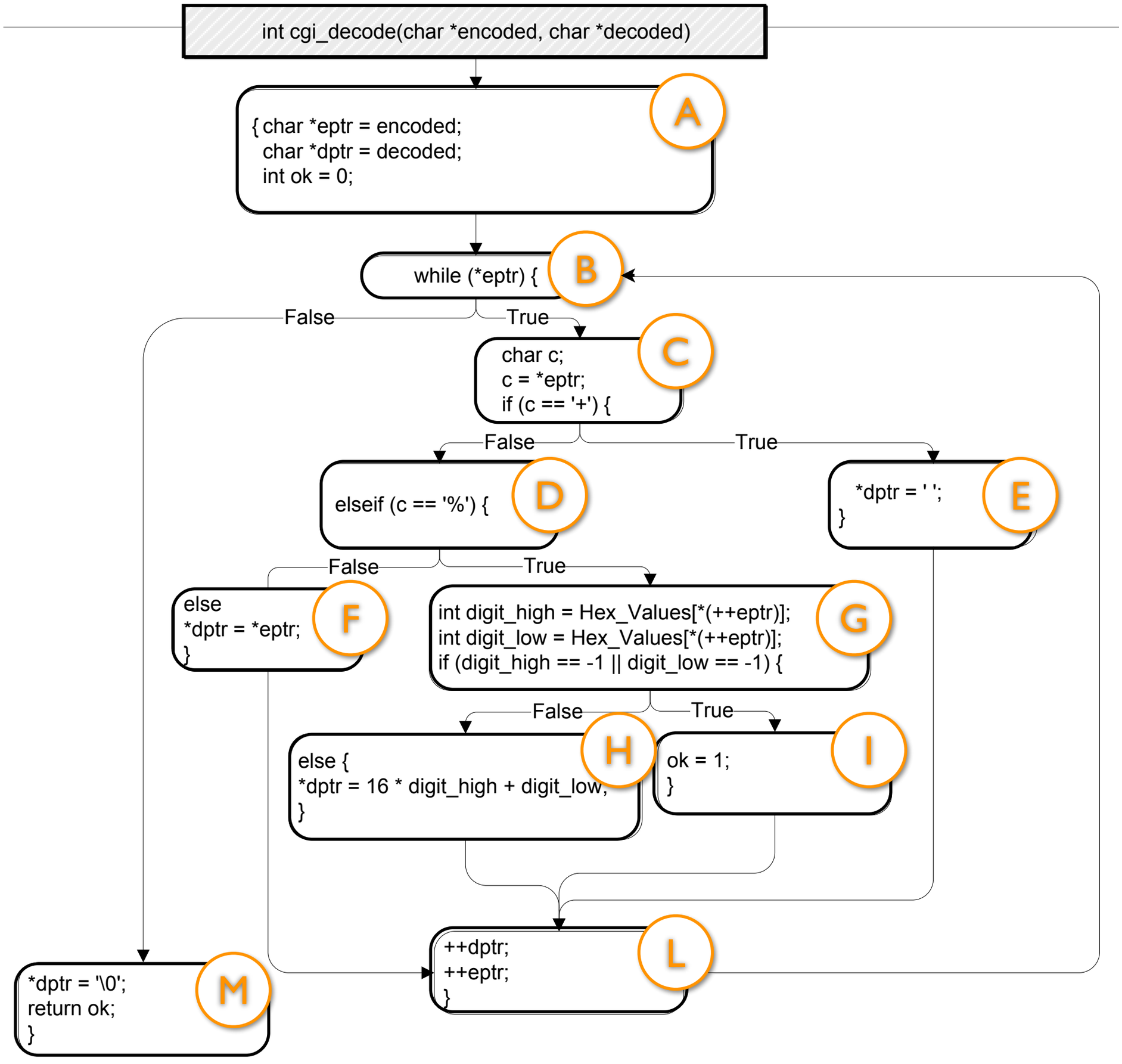
Ejemplo: cgi_decode

```
/**
  * @title cgi_decode
  * @desc
  * Translate a string from the CGI encoding to plain ascii text
  * '+' becomes space, %xx becomes byte with hex value xx,
  * other alphanumeric characters map to themselves
  *
  * returns 0 for success, positive for erroneous input
  * 1 = bad hexadecimal digit
  */

int cgi_decode(char *encoded, char *decoded)
{
    char *eptr = encoded;
    char *dptr = decoded;
    int ok = 0;
```

A

```
while (*eptr)  /* loop to end of string (‘\0’ character) */
{
    char c;
    c = *eptr;
    if (c == ’+’) {  /* ‘+’ maps to blank */
        *dptr = ’ ’;
    } else if (c == ’%’) { /* ’%xx’ is hex for char xx */
        int digit_high = Hex_Values[*(++eptr)];
        int digit_low  = Hex_Values[*(++eptr)];
        if (digit_high == -1 || digit_low == -1)
            ok = 1; /* Bad return code */
        else
            *dptr = 16 * digit_high + digit_low;
    } else { /* All other characters map to themselves */
        *dptr = *eptr;
    }
    ++dptr; ++eptr;
}

*dptr = ‘\0’;   /* Null terminator for string */
return ok;
}
```

B C E D G I H F L M


int cgi_decode(char *encoded, char *decoded)
{ char *eptr = encoded;
char *dptr = decoded;
int ok = 0;
A
while (*eptr) {
B
False
True
char c;
c = *eptr;
if (c == '+') {
C
False
True
elseif (c == '%') {
D
*dptr = ' ';
}
E
False
True
else
*dptr = *eptr;
}
F
int digit_high = Hex_Values[*(++eptr)];
int digit_low = Hex_Values[*(++eptr)];
if (digit_high == -1 || digit_low == -1) {
G
False
True
else {
*dptr = 16 * digit_high + digit_low;
}
H
ok = 1;
}
I
++dptr;
++eptr;
}
L
*dptr = '\0';
return ok;
}
M

int cgi_decode(char *encoded, char *decoded)

"test"

{ char *eptr = encoded;
  char *dptr = decoded;
  int ok = 0;

while (*eptr) {

False

True

char c;
c = *eptr;
if (c == '+') {

False

True

elseif (c == '%') {

*dptr = ' ';
}

E

False

True

else
*dptr = *eptr;
}

int digit_high = Hex_Values[*(++eptr)];
int digit_low = Hex_Values[*(++eptr)];
if (digit_high == -1 || digit_low == -1) {

G

False

True

else {
*dptr = 16 * digit_high + digit_low;
}

H

ok = 1;
}

I

++dptr;
++eptr;
}

*dptr = '\0';
return ok;
}

int cgi_decode(char *encoded, char *decoded)

{ char *eptr = encoded;
  char *dptr = decoded;
  int ok = 0;

“test”

“a+b”

while (*eptr) {

False

True

char c;
c = *eptr;
if (c == '+') {

False

True

elseif (c == '%') {

*dptr = ' ';
}

False

True

else
*dptr = *eptr;
}

int digit_high = Hex_Values[*(++eptr)];
int digit_low = Hex_Values[*(++eptr)];
if (digit_high == -1 || digit_low == -1) {

G

False

True

else {
*dptr = 16 * digit_high + digit_low;
}

H

ok = 1;
}

I

++dptr;
++eptr;
}

*dptr = '\0';
return ok;
}

int cgi_decode(char *encoded, char *decoded)

{ char *eptr = encoded;
  char *dptr = decoded;
  int ok = 0;

“test”

“a+b”

“%3d”

while (*eptr) {

False

True

char c;
c = *eptr;
if (c == '+') {

False

True

elseif (c == '%') {

*dptr = ' ';
}

False

True

else
*dptr = *eptr;
}

int digit_high = Hex_Values[*(++eptr)];
int digit_low = Hex_Values[*(++eptr)];
if (digit_high == -1 || digit_low == -1) {

False

True

else {
*dptr = 16 * digit_high + digit_low;
}

ok = 1;
}

++dptr;
++eptr;
}

*dptr = '\0';
return ok;
}

int cgi_decode(char *encoded, char *decoded)

{ char *eptr = encoded;
  char *dptr = decoded;
  int ok = 0;

while (*eptr) {

False

True

char c;
c = *eptr;
if (c == '+') {

False

True

elseif (c == '%') {

*dptr = ' ';
}

False

True

else
*dptr = *eptr;
}

int digit_high = Hex_Values[*(++eptr)];
int digit_low = Hex_Values[*(++eptr)];
if (digit_high == -1 || digit_low == -1) {

False

True

else {
*dptr = 16 * digit_high + digit_low;
}

ok = 1;
}

++dptr;
++eptr;
}

*dptr = '\0';
return ok;
}

“test”

“a+b”

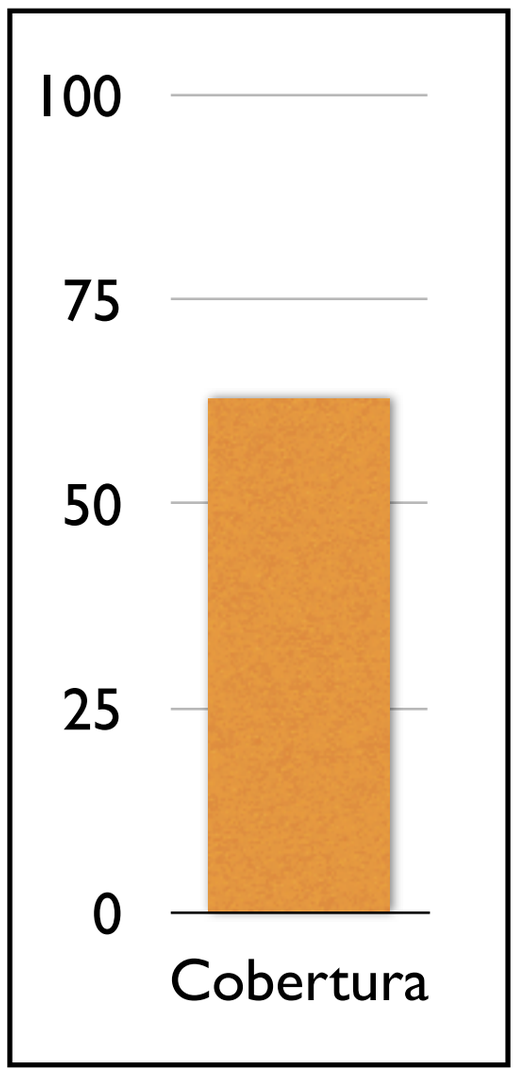
“%3d”

“%g”

full coverage

Statement Testing

- Cobertura: $\frac{\text{\# statements ejecutados}}{\text{\# statements}}$



int cgi_decode(char *encoded, char *decoded)

"test"

{ char *eptr = encoded;
  char *dptr = decoded;
  int ok = 0;

while (*eptr) {

False

True

char c;
c = *eptr;
if (c == '+') {

False

True

elseif (c == '%') {

*dptr = ' ';
}

E

False

True

else
*dptr = *eptr;
}

int digit_high = Hex_Values[*(++eptr)];
int digit_low = Hex_Values[*(++eptr)];
if (digit_high == -1 || digit_low == -1) {

G

False

True

else {
*dptr = 16 * digit_high + digit_low;
}

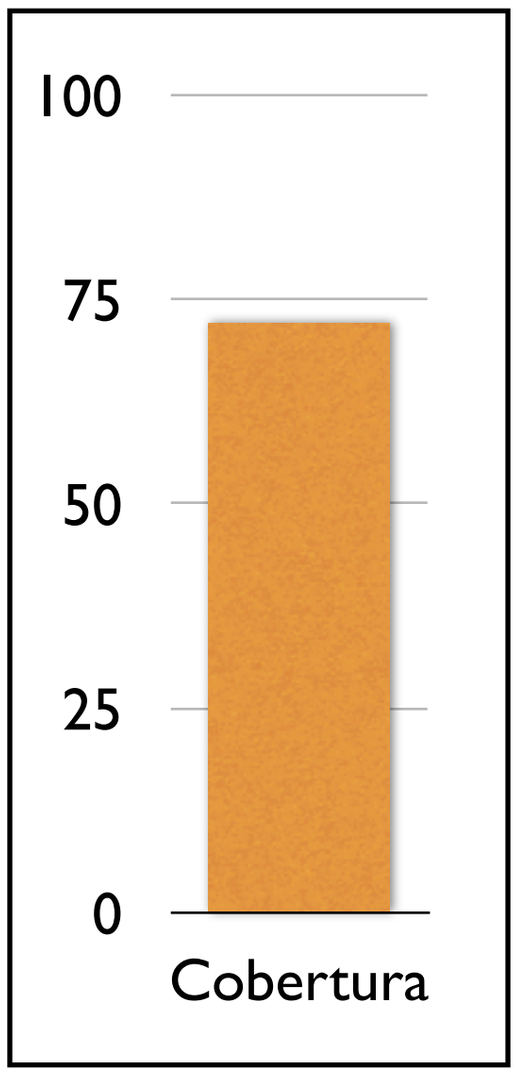
H

ok = 1;
}

I

++dptr;
++eptr;
}

*dptr = '\0';
return ok;
}



“test”

“a+b”

```
int cgi_decode(char *encoded, char *decoded)
{ char *eptr = encoded;
  char *dptr = decoded;
  int ok = 0;
while (*eptr) {
```

False

True

```
char c;
c = *eptr;
if (c == '+') {
```

False

True

```
elseif (c == '%') {
```

```
  *dptr = ' ';
}
```

False

True

```
else
*dptr = *eptr;
}
```

```
int digit_high = Hex_Values[*(++eptr)];
int digit_low = Hex_Values[*(++eptr)];
if (digit_high == -1 || digit_low == -1) {
```

G

False

True

```
else {
*dptr = 16 * digit_high + digit_low;
}
```

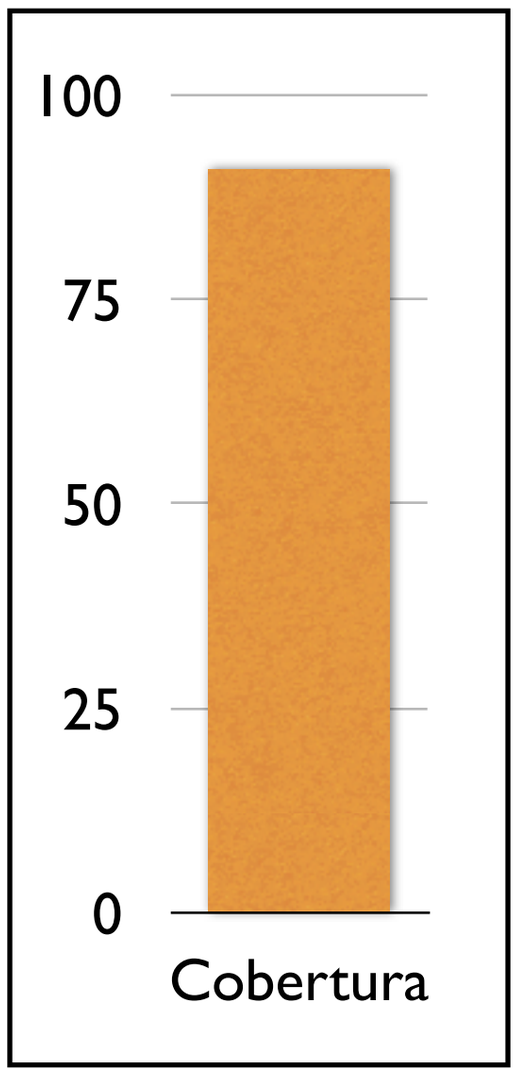
H

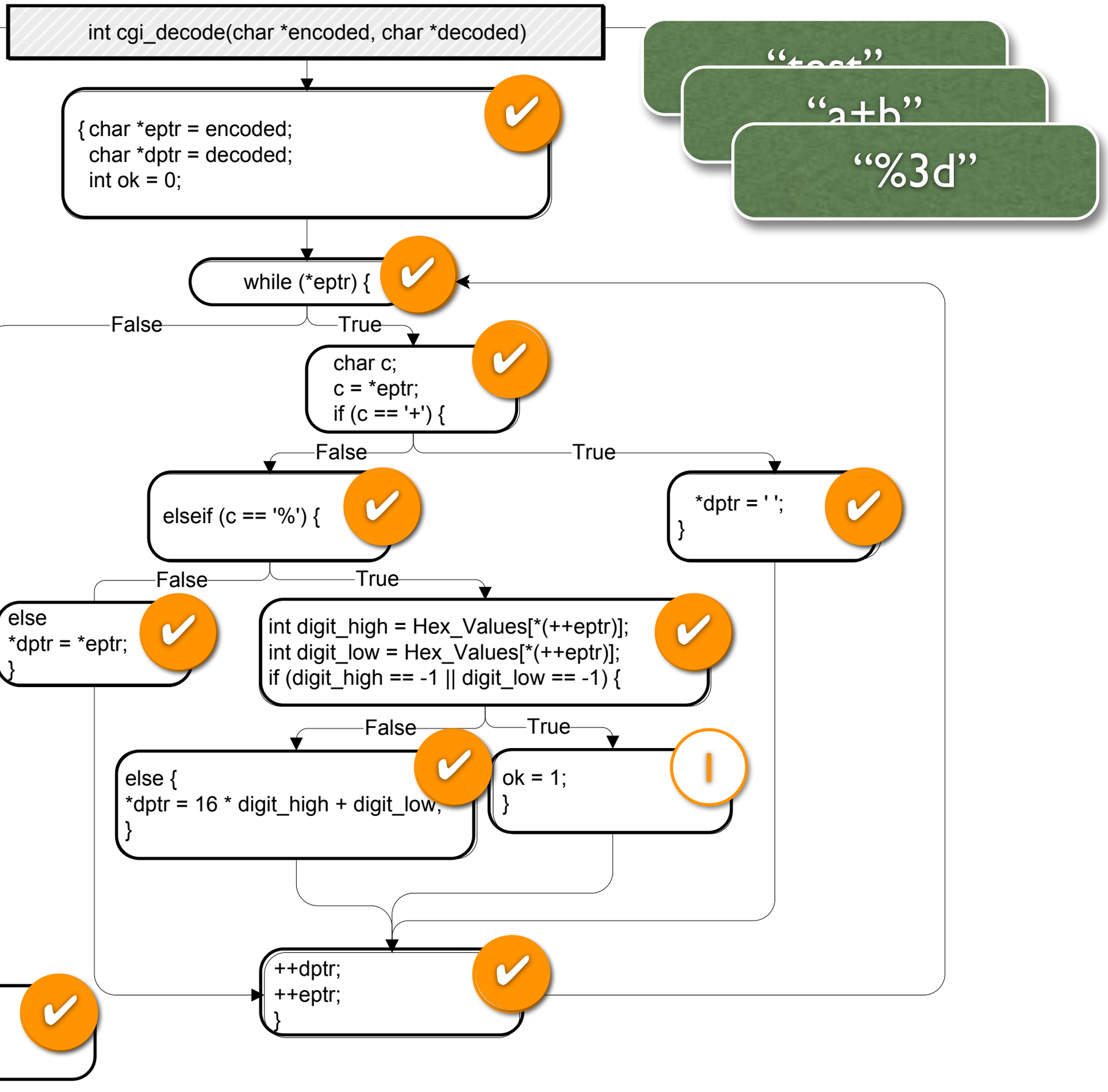
```
ok = 1;
}
```

I

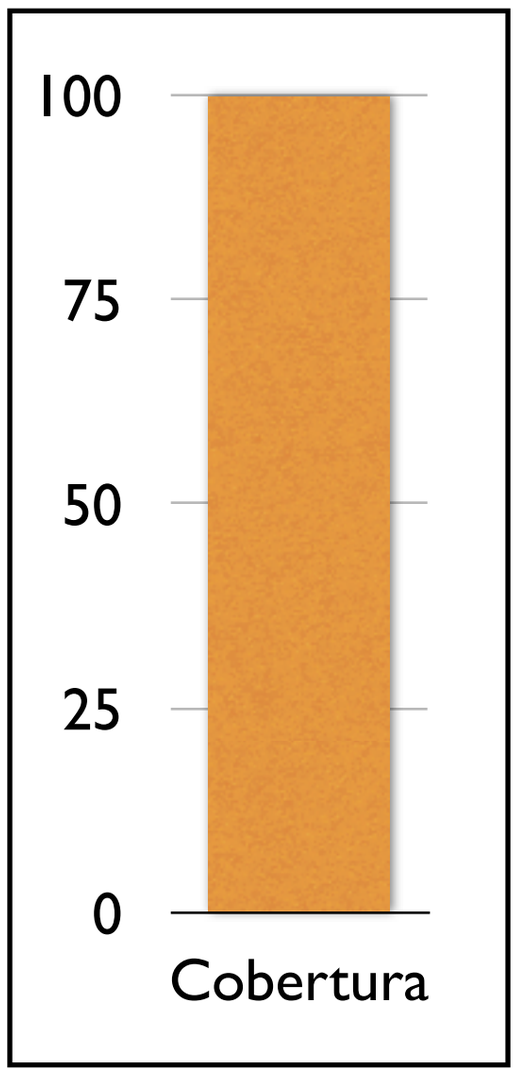
```
++dptr;
++eptr;
}
```

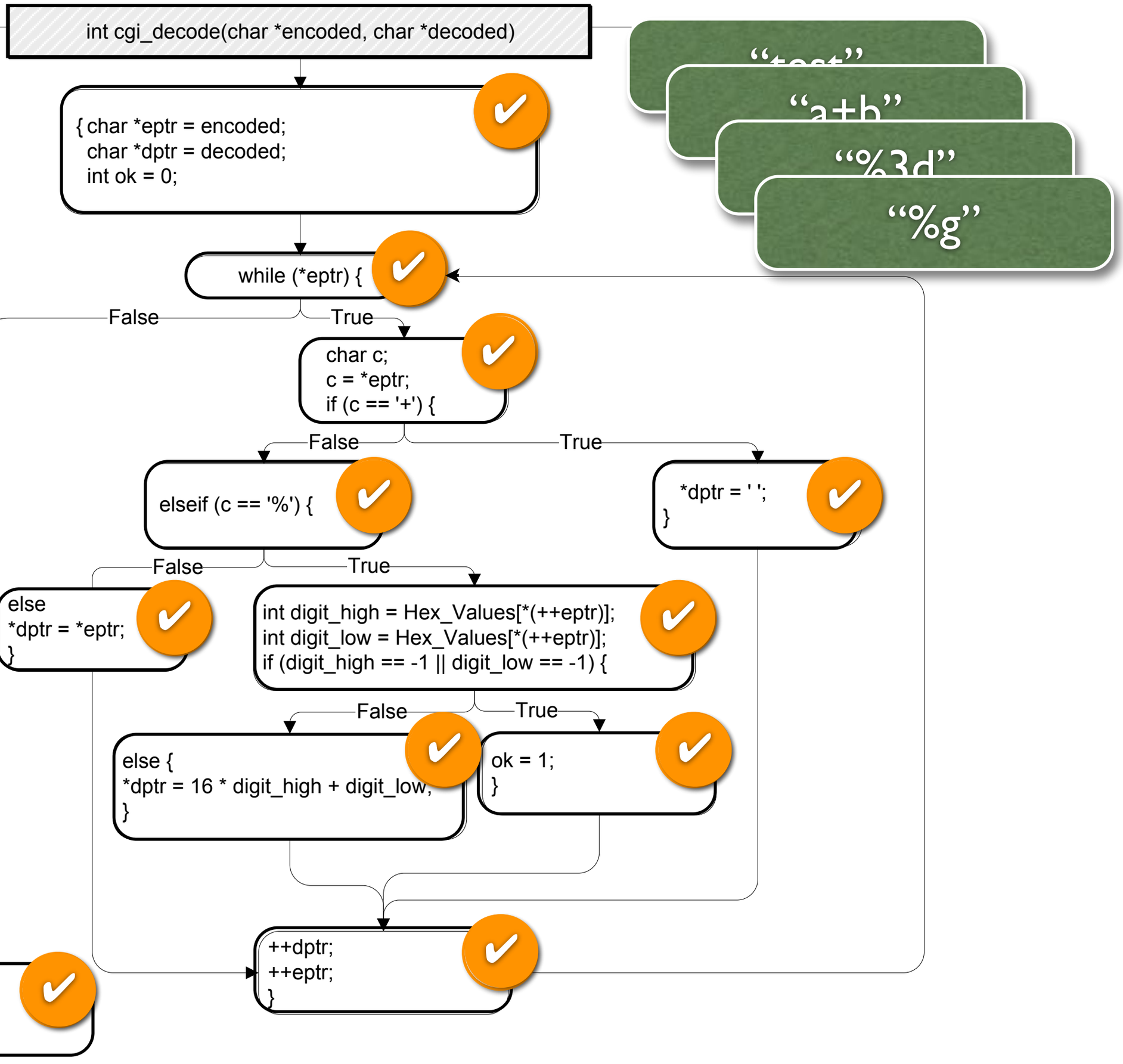
```
*dptr = '\0';
return ok;
}
```


100
75
50
25
0
Cobertura


int cgi_decode(char *encoded, char *decoded)
{ char *eptr = encoded;
char *dptr = decoded;
int ok = 0;
while (*eptr) {
False
True
char c;
c = *eptr;
if (c == '+') {
False
True
elseif (c == '%') {
*dptr = ' ';
}
False
True
else
*dptr = *eptr;
}
int digit_high = Hex_Values[*(++eptr)];
int digit_low = Hex_Values[*(++eptr)];
if (digit_high == -1 || digit_low == -1) {
False
True
else {
*dptr = 16 * digit_high + digit_low;
}
ok = 1;
}
++dptr;
++eptr;
}
*dptr = '\0';
return ok;
}

“test”
“a+b”
“%3d”


100
75
50
25
0
Cobertura


int cgi_decode(char *encoded, char *decoded)
{ char *eptr = encoded;
char *dptr = decoded;
int ok = 0;
while (*eptr) {
False
True
char c;
c = *eptr;
if (c == '+') {
False
True
elseif (c == '%') {
*dptr = ' ';
}
False
True
else
*dptr = *eptr;
}
int digit_high = Hex_Values[*(++eptr)];
int digit_low = Hex_Values[*(++eptr)];
if (digit_high == -1 || digit_low == -1) {
False
True
else {
*dptr = 16 * digit_high + digit_low;
}
ok = 1;
}
++dptr;
++eptr;
}
*dptr = '\0';
return ok;
}

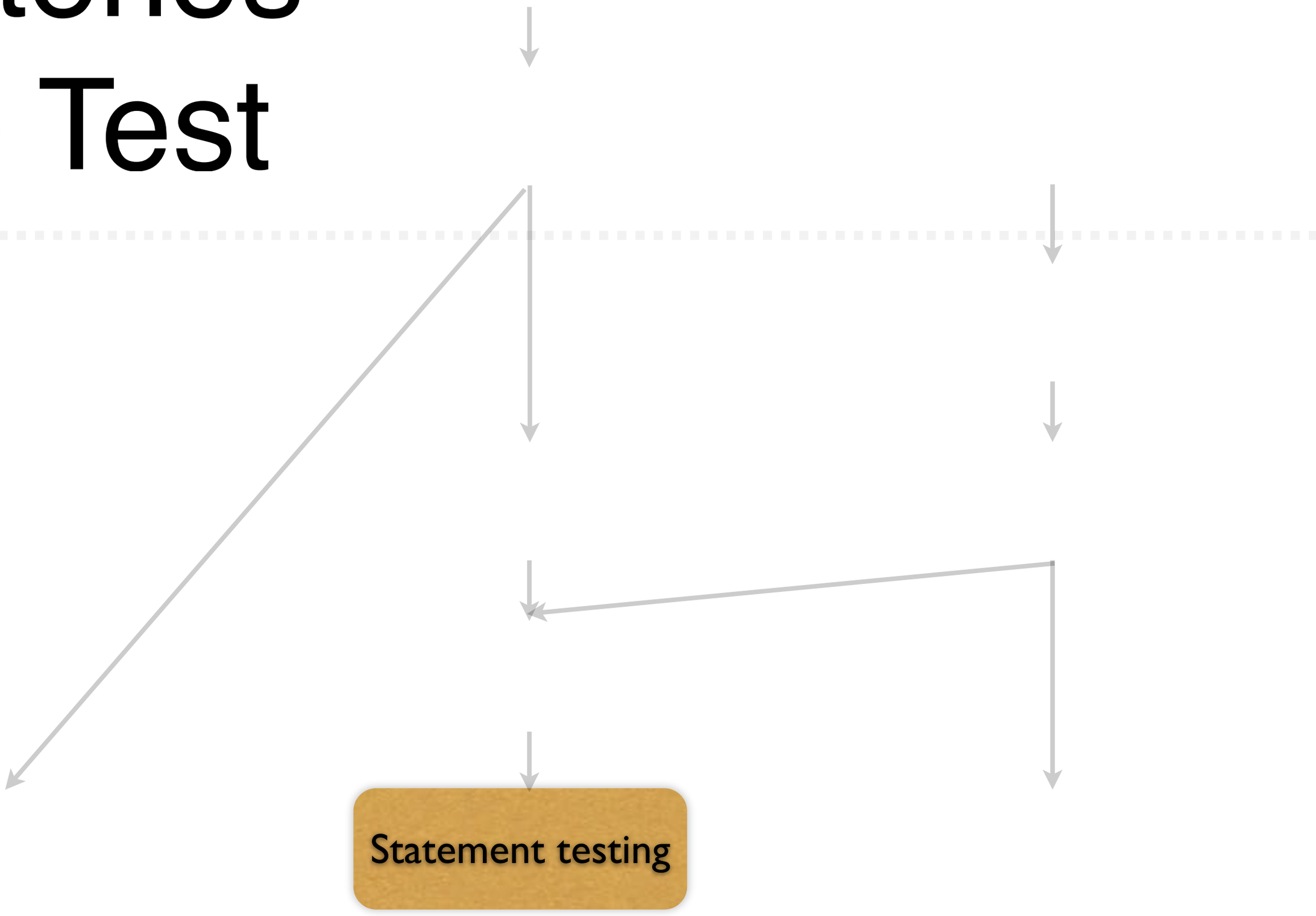
“test”
“a+b”
“%3d”
“%g”

Calculando la Cobertura

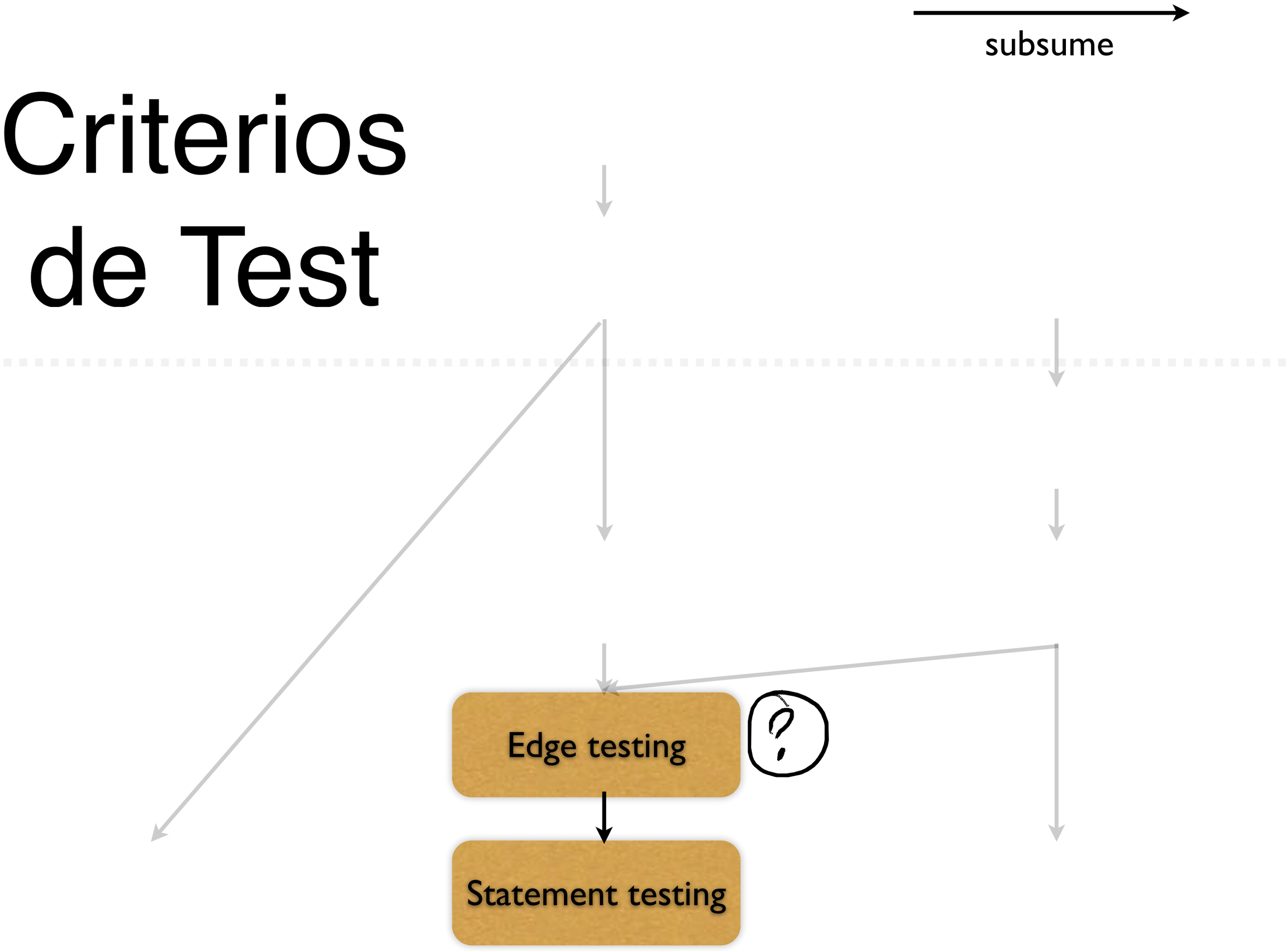
- La Cobertura es computada automáticamente mientras el programa es ejecutado
- Requiere la *instrumentación* en tiempo de compilación
- Luego de la ejecución, una herramienta de *cobertura* analiza y resume los resultados

modificación del programa que permite agregar funcionalidad (profilers)

Criterios de Test



Criterios de Test



Structural vs State vs

Edge Testing

↳ intuitively: try to infect using different state classes

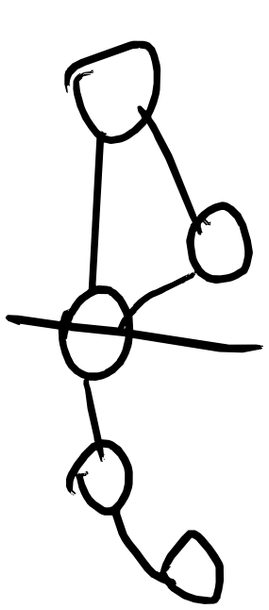
- Criterio de adecuación: cada arco en el CFG debe ser ejecutado al menos una vez
- Cobertura : $\frac{\#\text{ arcos ejecutados}}{\#\text{ arcos}}$ (edge coverage)
- Subsume Statement Testing
  ya que al recorrer todos los arcos recorremos todos los nodos

(Statement testing ⊆ Edge testing)

Structural
vs
Stole
vs
Edge vs

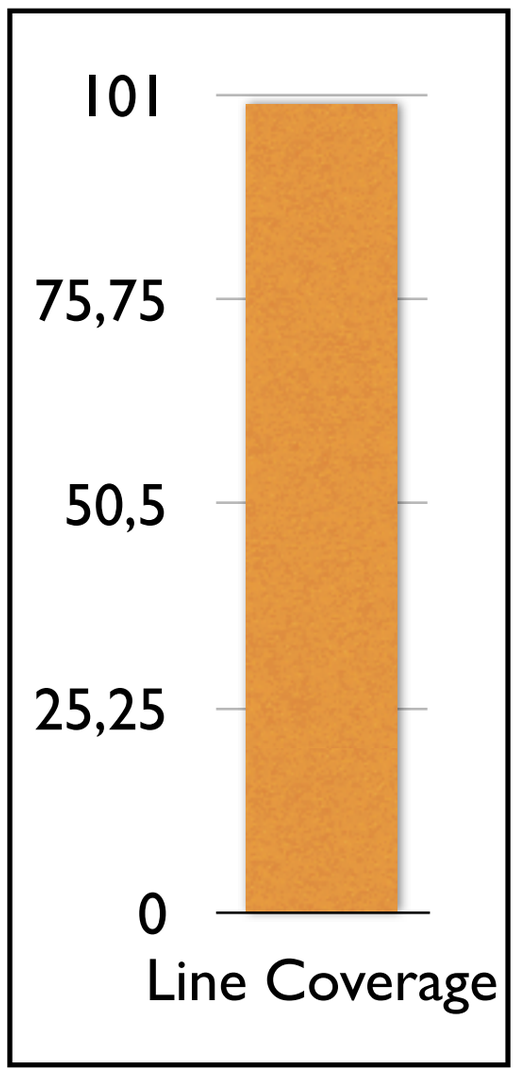
Branch Testing

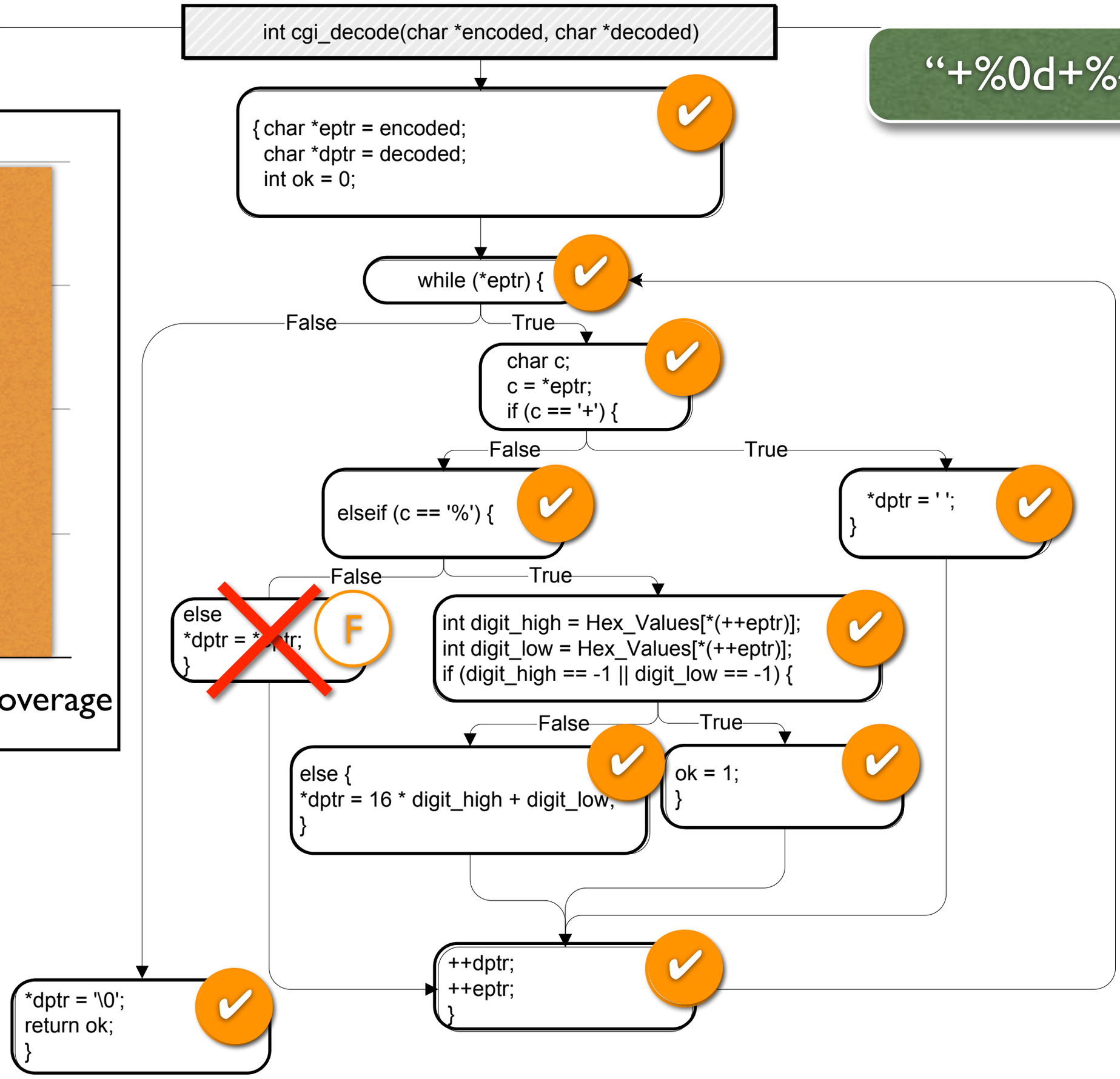
- Criterio de adecuación: cada branch en el CFG debe ser ejecutado al menos una vez
- Cobertura : $\frac{\text{\# branches ejecutados}}{\text{\# branches}}$



- **No** Subsume Statement Testing ni Edge Testing puedo ejercitar todos los branches y no cubrir todos los arcos/nodos
- Mas comúnmente usado en la Industria

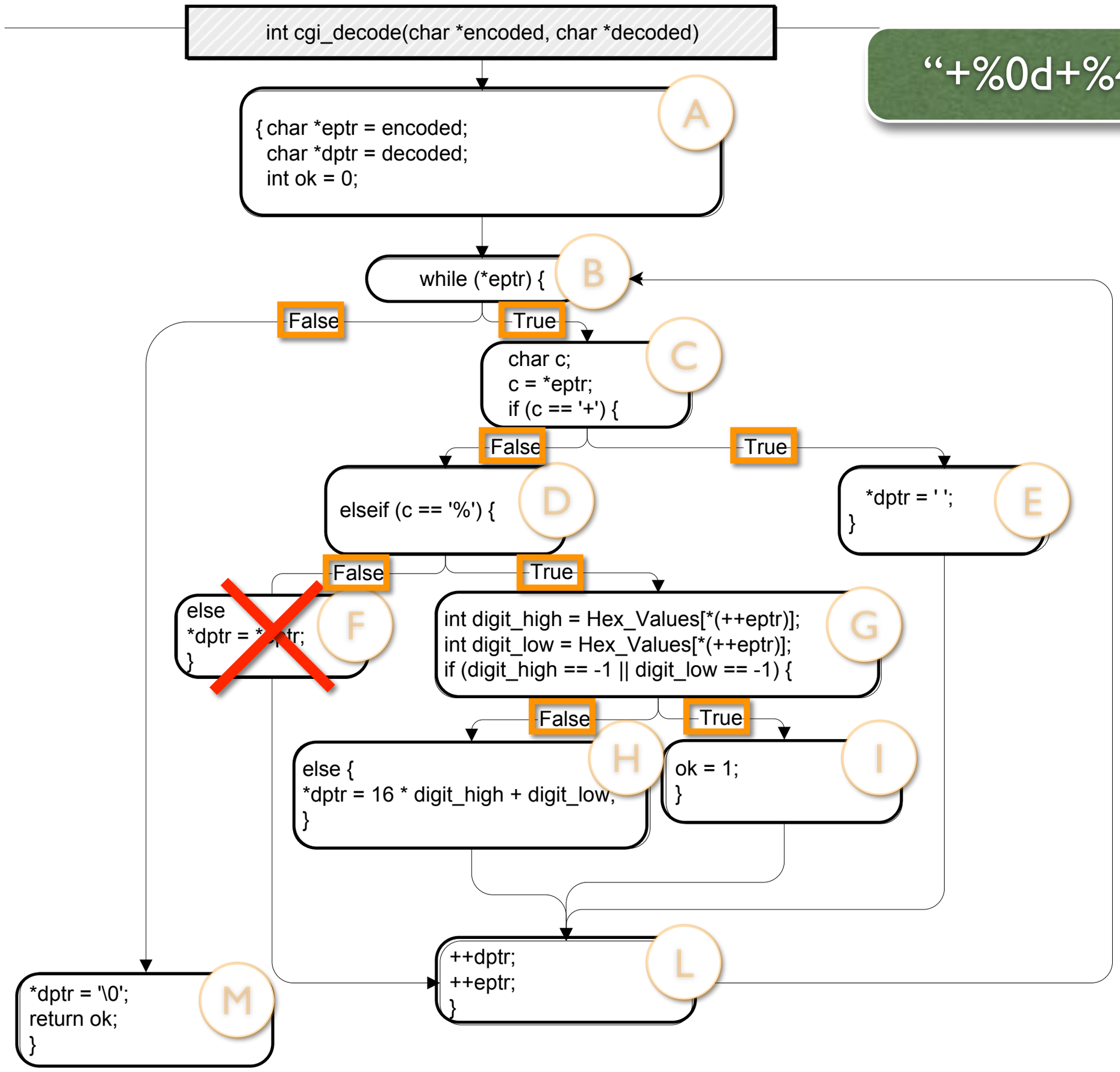
WTF is a BRANCH? → branch of a node?
V o F





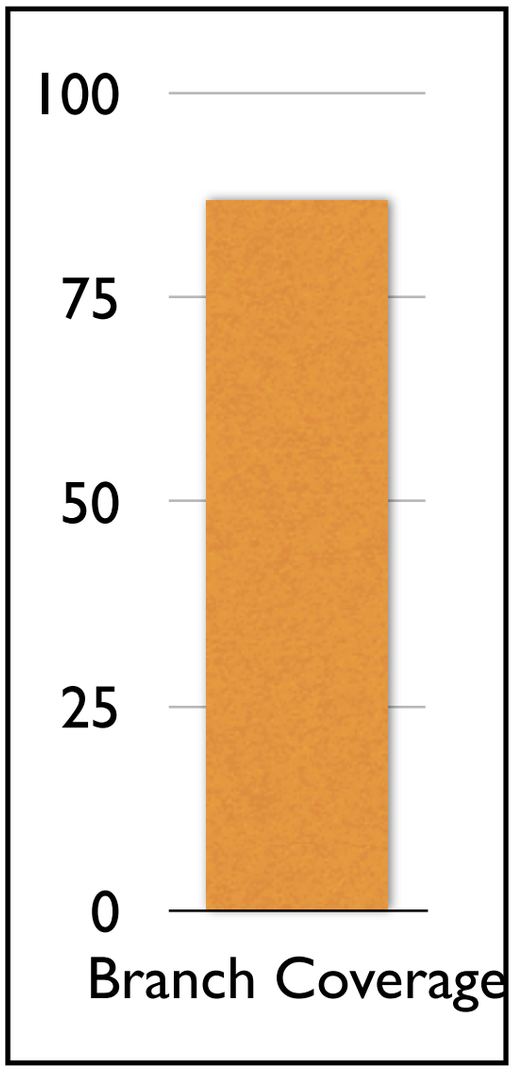
“+%0d+%4j”

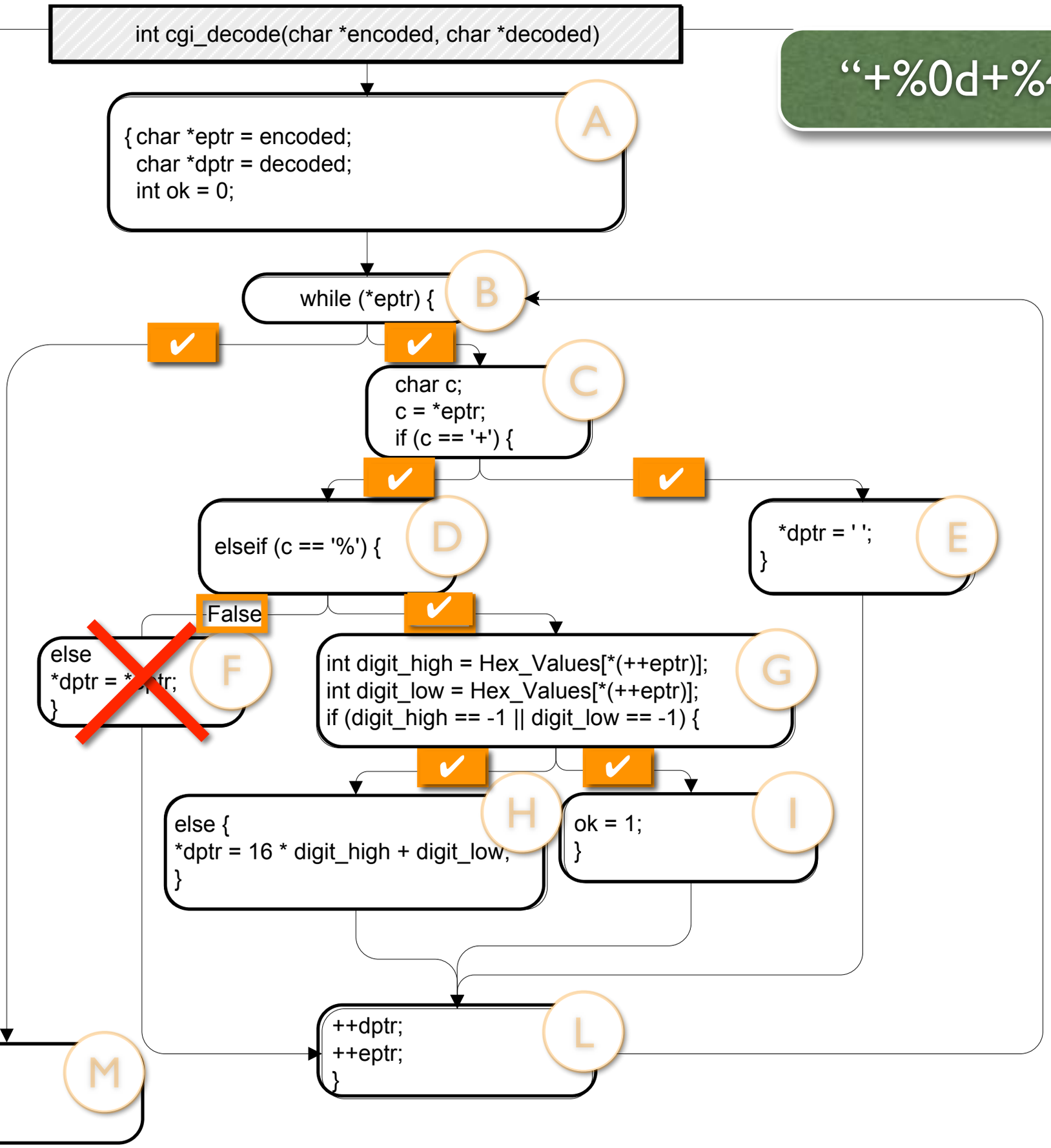
Asumamos que F no existe (defecto)



"+%0d+%4j"

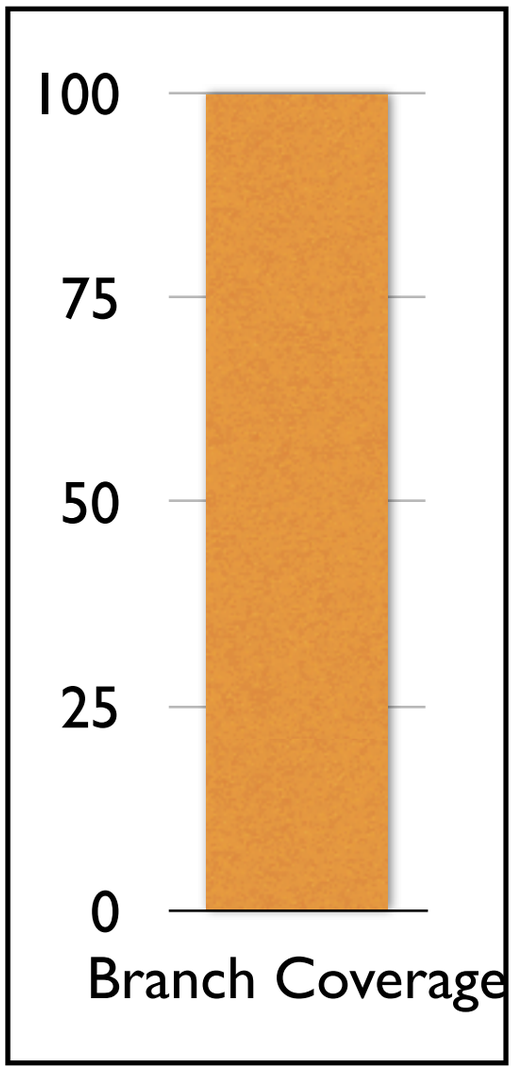
Asumamos que F no existe (defecto)

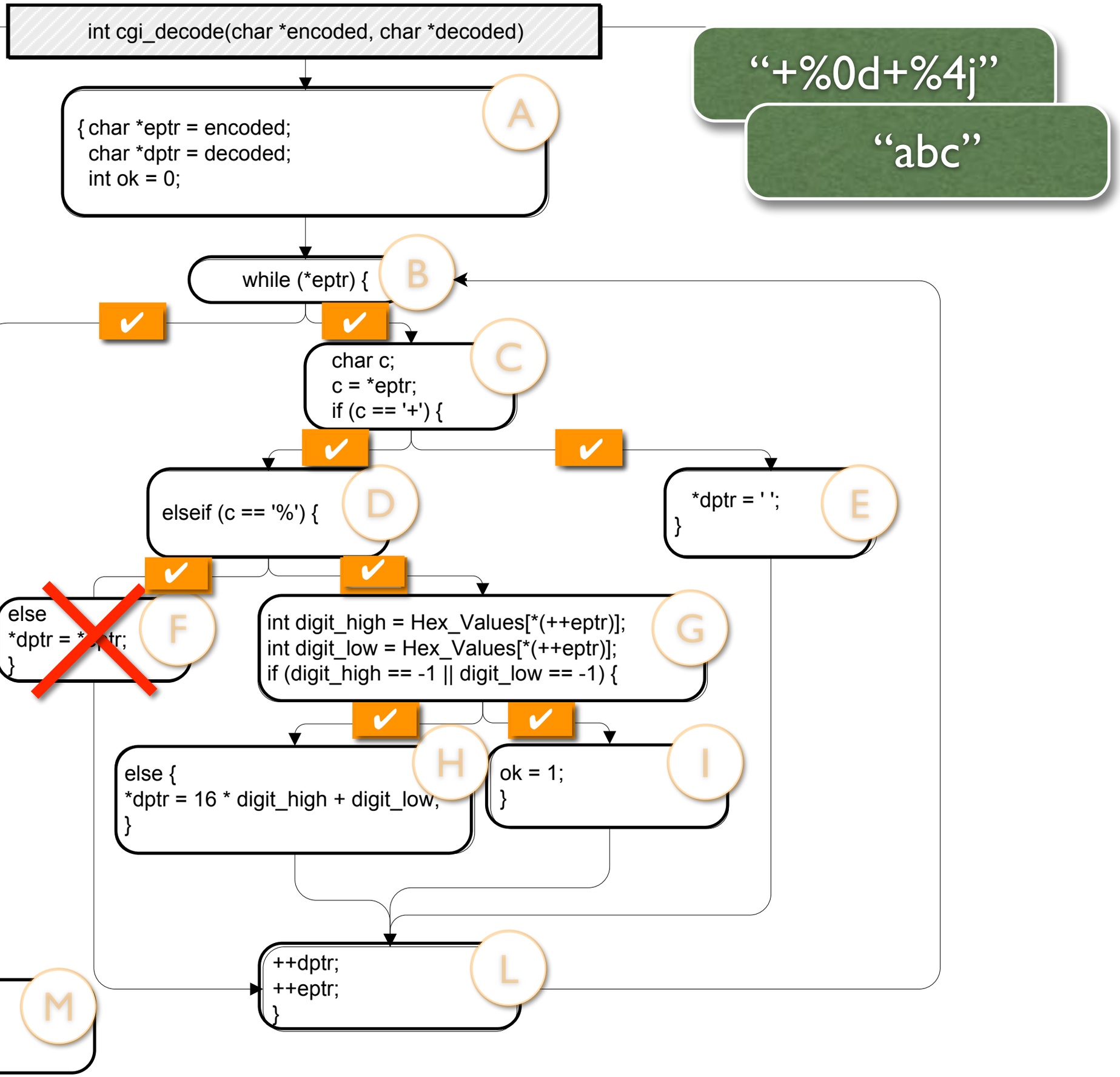




"+%0d+%4j"

Si consideramos los branches, entonces ejercitaríamos el defecto

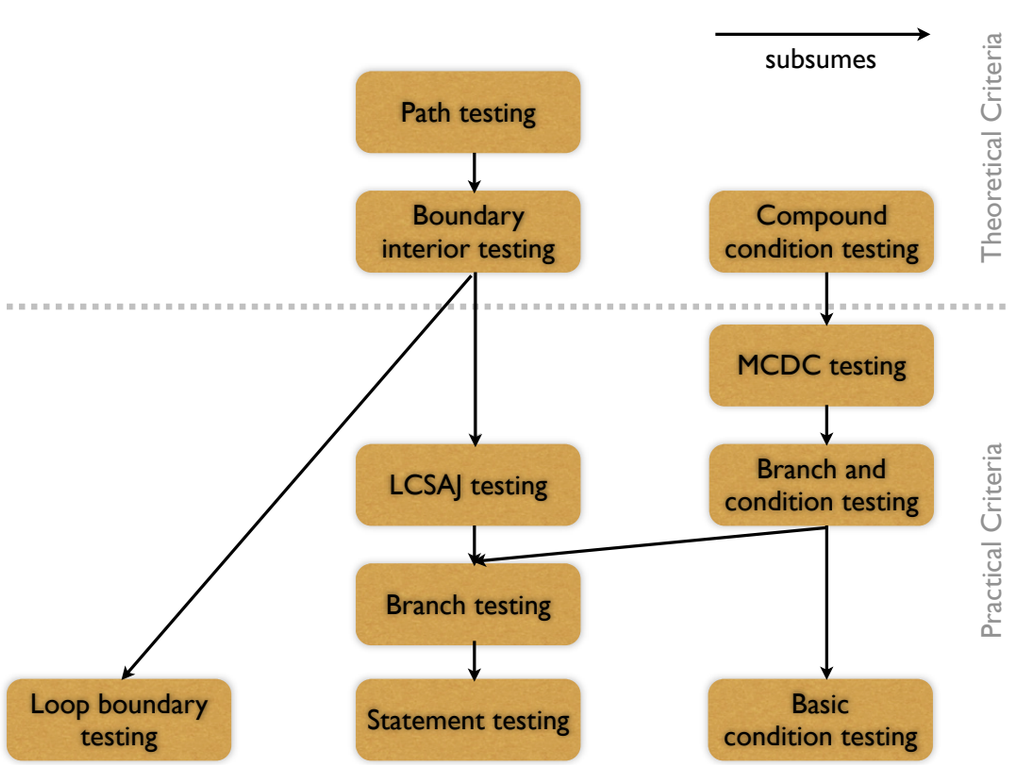

100
75
50
25
0
Branch Coverage

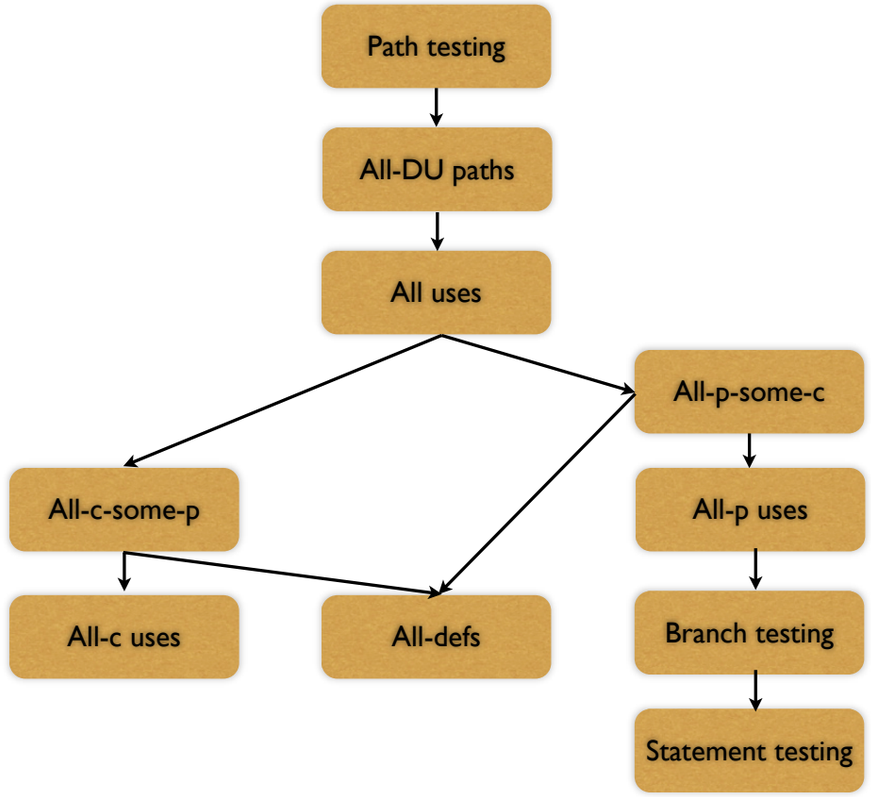

int cgi_decode(char *encoded, char *decoded)
{ char *eptr = encoded;
char *dptr = decoded;
int ok = 0;
A
while (*eptr) {
B
char c;
c = *eptr;
if (c == '+') {
C
elseif (c == '%') {
D
*dptr = ' ';
}
E
else
*dptr = *eptr;
}
F
int digit_high = Hex_Values[*(++eptr)];
int digit_low = Hex_Values[*(++eptr)];
if (digit_high == -1 || digit_low == -1) {
G
else {
*dptr = 16 * digit_high + digit_low;
}
H
ok = 1;
}
I
++dptr;
++eptr;
}
L
*dptr = '\0';
return ok;
}
M

"+%0d+%4j"

"abc"

¿Cuán buena es mi Test Suite?





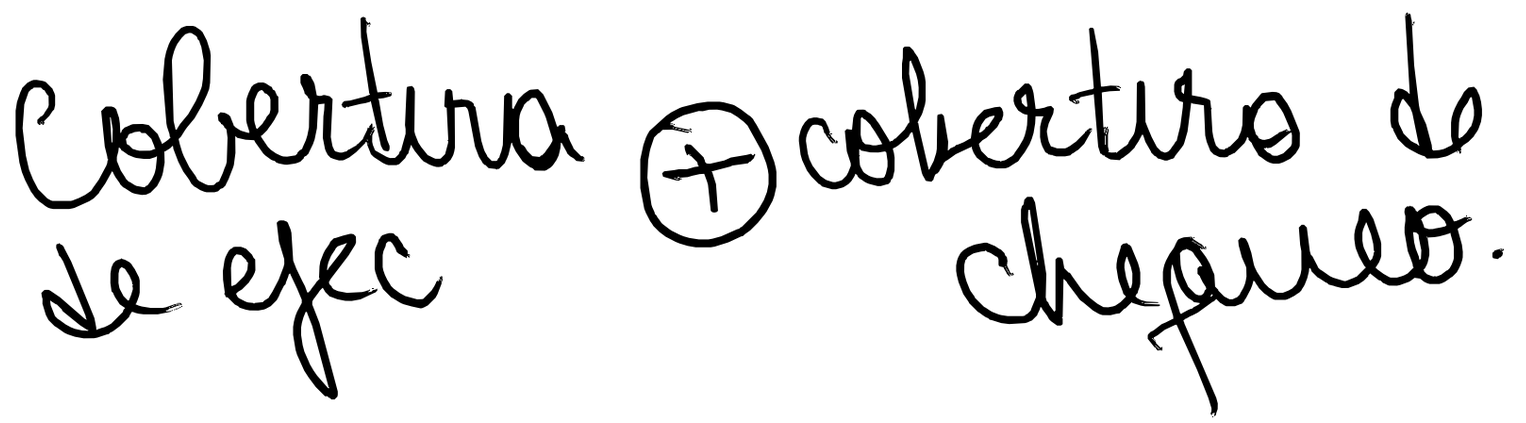
```
/**
 * Make sure Double.NaN is returned iff n = 0
 *
 */
public void testNaN() {
    StandardDeviation std = new StandardDeviation();
    assertTrue(Double.isNaN(std.getResult()));
    std.increment(1d);
    assertEquals(0d, std.getResult(), 0);
}
```

```
/**
 * Make sure Double.NaN is returned iff n = 0
 *
 */
public void testNaN() {
    StandardDeviation std = new StandardDeviation();
    Double.isNaN(std.getResult());
    std.increment(1d);
    std.getResult();
}
```

¡La cobertura del código no cambió!

¿Cuán buenos son mis Tests?

- Cobertura = cuánto de mi código es ejecutado
- Pero… ¿Cuánto de mi código es chequeado contra un comportamiento esperado?
- No sabemos donde se encuentran los bugs (defectos)
- Pero ¡Sabemos los errores que hemos cometido en el pasado!



Aprendiendo de los Errores

- **Idea**: Aprender de errores previos para prevenir que estos ocurran de nuevamente
- **Técnica**: *Simular errores anteriores* y comprobar si los defectos simulados pueden ser detectados

  → puede llegar a fallo (error en el código)

- Conocido como F*ault-Based testing* o *Mutation testing*

  → FAULT BASED TESTING
  or MUTATION TESTING

  → simulate previous errors & check if simulated defects can be detected.

Defect estimation / prediction ANALYSIS

Hipótesis Básicas para Estimar Defectos

(errores de código)

- Juzgamos la efectividad de un test suite para encontrar errores midiendo cuán bien puede encontrar defectos "artificiales".
- Esto es válido únicamente si los bugs plantados son **representativos** de los bugs reales
- No deben ser necesariamente iguales; pero las diferencias no deberían afectar la selección

How to form such bugs?

```
int do_something(int x, int y)
{
  if(x < y)
    return x+y;
  else
    return x*y;
}
```

Programa

```
int a = do_something(5, 10);
assertEquals(a, 15);
```

Test

Test

∴ mutant detected

int do_something(int x, int y)
{ if(x<y)
  ret x-y;
 else
  ret x*y;
}

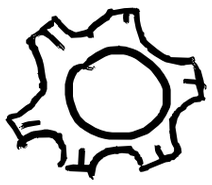
Mutante

Mutation testing includes:

Mutantes

- Modificación
  Versión levemente modificada del programa original
- Cambio Sintáctico Válido (código es compilable)
- Simple ("typo" de programación)

can be thought of as:



Generando Mutantes

(How to generate mutants) X-MEN

- Operadores de Mutación
  Regla para derivar mutantes a partir de un programa
  - Mutaciones basadas en fallas reales
    Operadores de Mutación que representan errores típicos (específicos a un proyecto)
  - Operadores de Mutación Genéricos han sido definidos para la mayoría de los lenguajes de programación
    - > 100 operadores de mutación para C

using recorded history of program

ABS - Absolute Value Insertion

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x % y;
    x = y;
    y = tmp;
  }

  return x;
}
```

ABS - Absolute Value Insertion

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x % y;
    x = abs(y);
    y = tmp;
  }

  return x;
}
```

ABS - Absolute Value Insertion

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x % y;
    x = y;
    y = tmp;
  }

  return -abs(x);
}
```

Mutation ABS

ABS - Absolute Value Insertion

```
int gcd(int x, int y) {
  int tmp;
  while(y != 0) {
    tmp = x % y;
    x = y;
    y = tmp;
  }

  return 0;
}
```

↑ can be zero as defined by ABS mutation

Mutation AOR

AOR - Arithmetic Operator Replacement

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x % y;
    x = y;
    y = tmp;
  }

  return x;
}
```

AOR - Arithmetic Operator Replacement

```
int gcd(int x, int y) {
  int tmp;

  while(y != 0) {
    tmp = x * y;
    x = y;
    y = tmp;
  }

  return x;
}
```

+, -, *, /, %, **, x, y

Mutation ROR (remplazo de predicados)

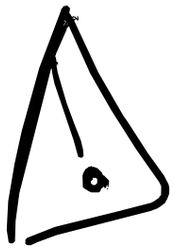
ROR - Relational Operator Replacement

```
int gcd(int x, int y) {
   int tmp;

   while(y != 0) {
      tmp = x % y;
      x = y;
      y = tmp;
   }

   return x;
}
```

ROR - Relational Operator Replacement

```
int gcd(int x, int y) {
  int tmp;

  while(y > 0) {
    tmp = x % y;
    x = y;
    y = tmp;
  }

  return x;
}
```

<, >, <=, >=, =,
!=, false, true



¿Más tipos operadores?

Operadores de Mutación

id	operator	description	constraint
Operand Modifications			
crp	constant for constant replacement	replace constant $C1$ with constant $C2$	$C1 \neq C2$
scr	scalar for constant replacement	replace constant C with scalar variable X	$C \neq X$
acr	array for constant replacement	replace constant C with array reference $A[I]$	$C \neq A[I]$
scr	struct for constant replacement	replace constant C with struct field S	$C \neq S$
svr	scalar variable replacement	replace scalar variable X with a scalar variable Y	$X \neq Y$
csr	constant for scalar variable replacement	replace scalar variable X with a constant C	$X \neq C$
asr	array for scalar variable replacement	replace scalar variable X with an array reference $A[I]$	$X \neq A[I]$
ssr	struct for scalar replacement	replace scalar variable X with struct field S	$X \neq S$
vie	scalar variable initialization elimination	remove initialization of a scalar variable	
car	constant for array replacement	replace array reference $A[I]$ with constant C	$A[I] \neq C$
sar	scalar for array replacement	replace array reference $A[I]$ with scalar variable X	$A[I] \neq X$
cnr	comparable array replacement	replace array reference with a comparable array reference	
sar	struct for array reference replacement	replace array reference $A[I]$ with a struct field S	$A[I] \neq S$
Expression Modifications			
abs	absolute value insertion	replace e by `abs(`e`)`	$e < 0$
aor	arithmetic operator replacement	replace arithmetic operator ψ with arithmetic operator ϕ	$e_1 \psi e_2 \neq e_1 \phi e_2$
lcr	logical connector replacement	replace logical connector ψ with logical connector ϕ	$e_1 \psi e_2 \neq e_1 \phi e_2$
ror	relational operator replacement	replace relational operator ψ with relational operator ϕ	$e_1 \psi e_2 \neq e_1 \phi e_2$
uoi	unary operator insertion	insert unary operator	
cpr	constant for predicate replacement	replace predicate with a constant value	
Statement Modifications			
sdl	statement deletion	delete a statement	
sca	switch case replacement	replace the label of one case with another	
ses	end block shift	move `}` one statement earlier and later	

OOM - object oriented mutation

Mutación Orientada a Objetos

- Hasta ahora los operadores sólo han considerado mutaciones de sentencias
  - Podemos mutar tambien elementos de lenguajes OO:

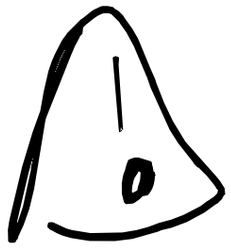
```
public class test {
   // ..
   private void do(){
   // ...
   }
}
```

mutation

```
protected void do(){
...
}
```

```
private void do(){
...
}
```

mutation can be applied if it encounters compilation rules



OOM

Mutación Orientada a Objetos

- AMC - Access Modifier Change
- HVD - Hiding Variable Deletion
- HVI - Hiding Variable Insertion
- OMD - Overriding Method Deletion
- OMM - Overridden Method Moving
- OMR - Overridden Method Rename
- SKR - Super Keyword Deletion
- PCD - Parent Constructor Deletion
- ATC - Actual Type Change
- DTC - Declared Type Change
- PTC - Parameter Type Change
- RTC - Reference Type Change
- OMC - Overloading Method Change
- OMD - Overloading Method Deletion
- AOC - Argument Order Change
- ANC - Argument Number Change
- TKD - this Keyword Deletion
- SMV - Static Modifier Change
- VID - Variable Initialization Deletion
- DCD - Default Constructor 2

Mutation Order (of magnitude?)

Orden de Mutaciones

!
- First Order Mutant (FOM)
  Exactamente una mutación (programa modificado)
- Cada operados de mutación define un conjunto de FOMs
- Número de FOMs
  ~ número de referencias a datos * números de referencias a objetos
- Higher Order Mutant (HOM)
  Mutación de otra mutación
- #HOM = $2^{\#FOM} - 1$

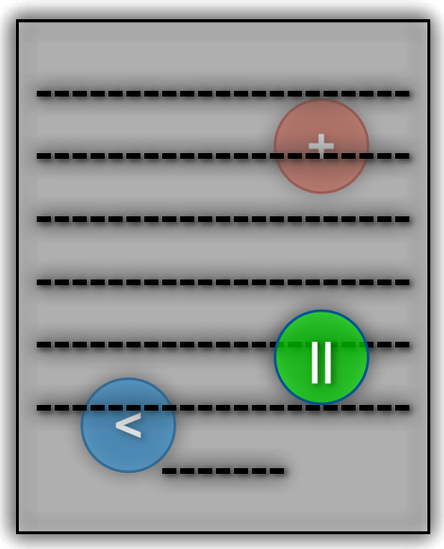
O → M1 → M2
FOM HOM

→ exponential explosion

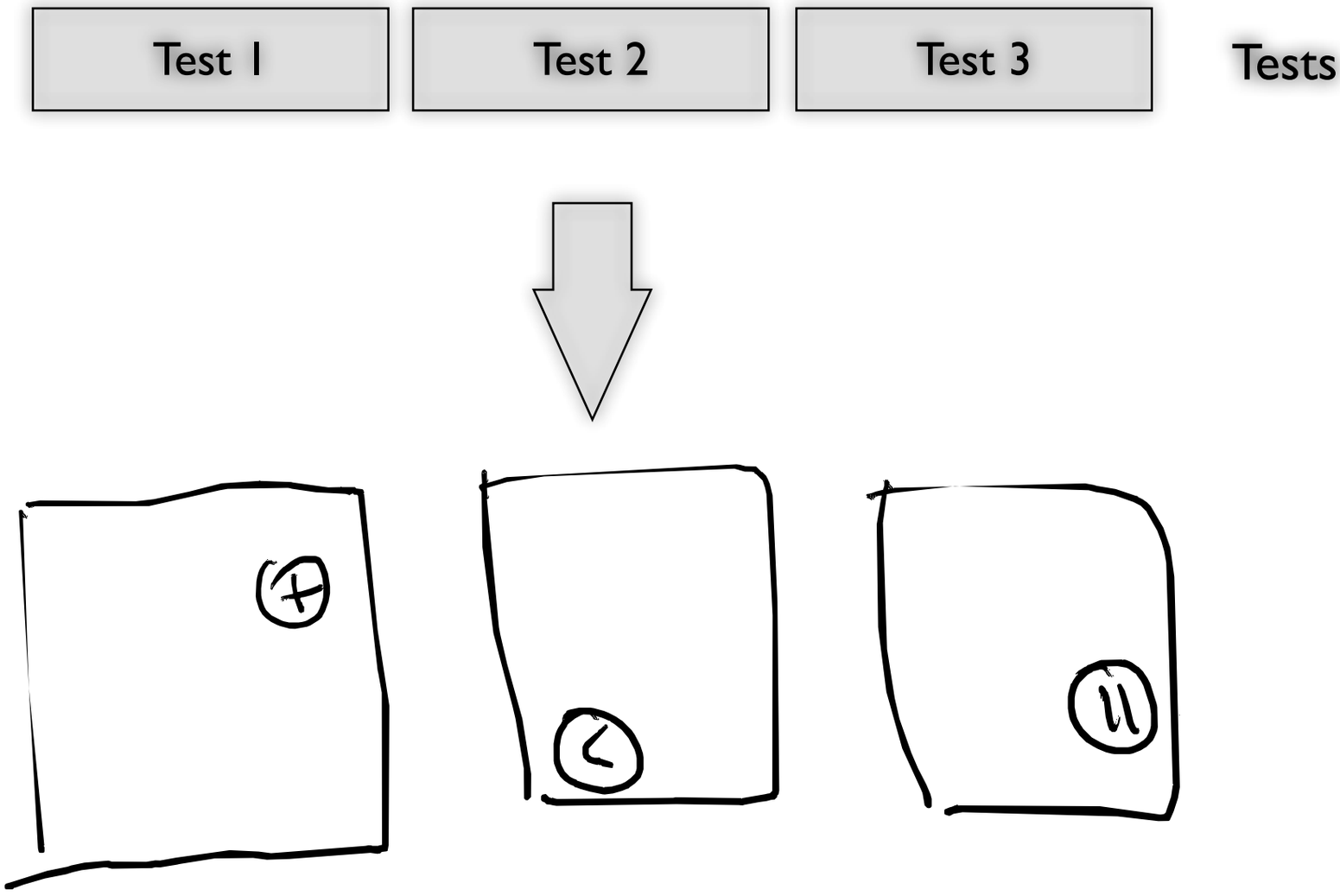
example?

Operadores

Programa Original



Tests

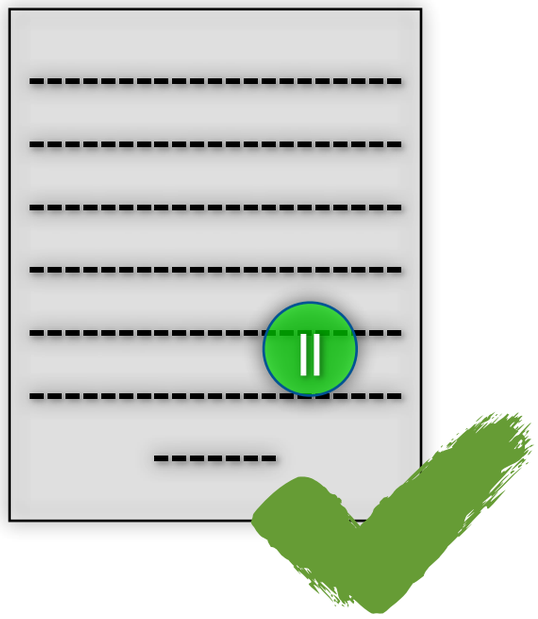


given mutation operators ⊕ original program

↳ always same amount of mutants are created

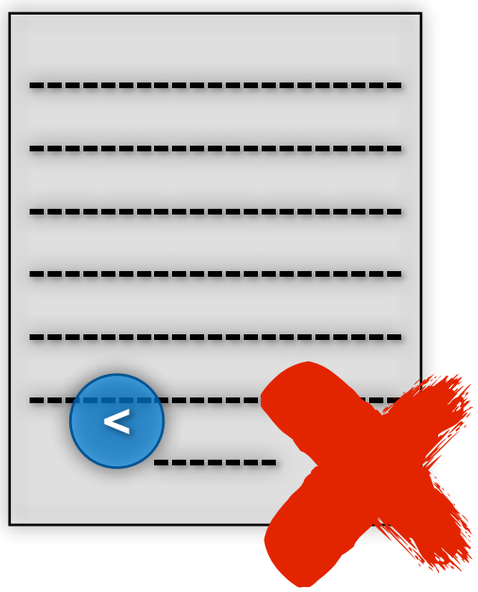
⇓

MUTANTS ALLOW FOR CHECKING HOW POWERFUL A TEST SUITE IS AT DETECTING DEFECTS



Mutante Vivo - necesitamos más Tests

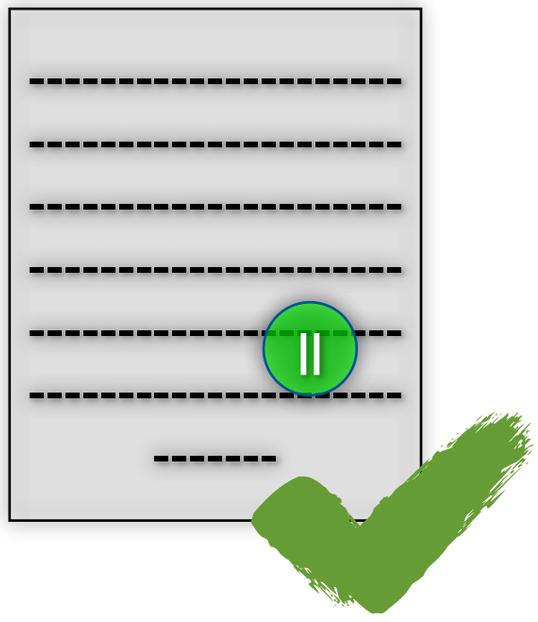
(not detected)
∴ current test suite is incapable of detection

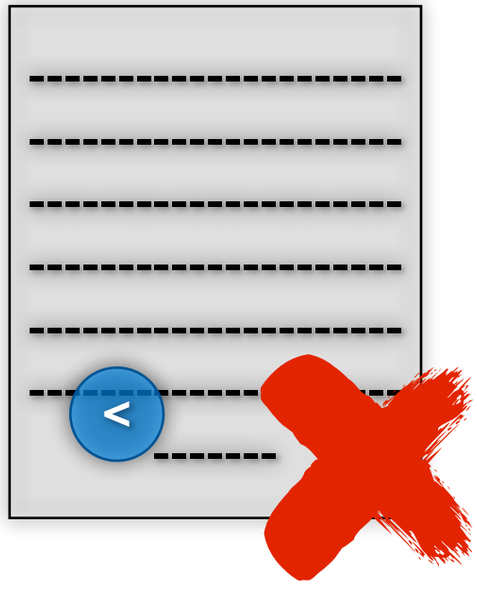


Mutante Muerto - no es más útil

(V) is good → killing off mutants means test suite can detect them

MUTATION SCORE



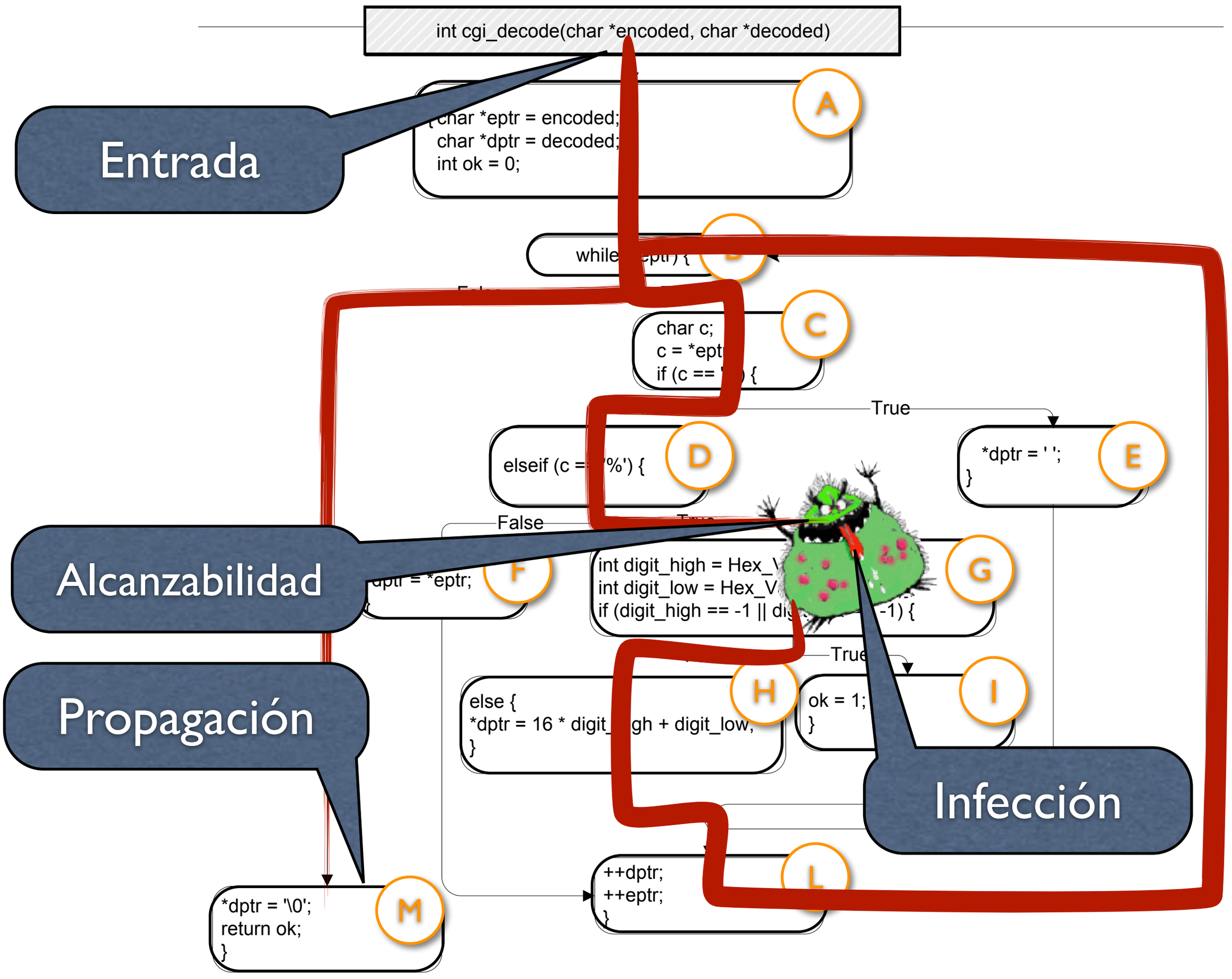


Mutation Score:

$$\frac{\text{Mutantes Muertos}}{\text{Total Mutantes}}$$

↳ should remain constant.

ejecución del defecto ⊕
infección ⊕ propagación ⊕ fallo



Equivalence in mutants ≈

Mutantes Equivalentes

≈ código inalcanzable en cobertura

- Mutación = cambio sintáctico
- El cambio puede dejar la semántica inalterada
- Los Mutantes Equivalentes son difíciles de detectar (problema indecidible)
  - Pueden ser alcanzados, pero quizás no se infecte el estado del programa
  - Pueden producir una infección, pero sin propagación

↓

EM basically means that the mutation keeps the defect without showing it/exposing it.

↳ doesn't alter the semantic in a useful way.

if mutation affects unreachable code
⟹ equivalent mutation → never reaches failure.

Ejemplo I

```
int max(int[] values) {
   int r, i;

   r = 0;
   for(i = 1; i<values.length; i++) {
     if (values[i] > values[r])
        r = i;
   }

   return values[r];
}
```

mutante equivalente (=)
Añadir código que modifique valido entre programa original y el mutado

Ejemplo I

```
int max(int[] values) {
   int r, i;

   r = 0;
   for(i = 0; i<values.length; i++) {
      if (values[i] > values[r])
         r = i;
   }

   return values[r];
}
```

¿Es equivalente? SI

Ejemplo I

```
int max(int[] values) {
  int r, i;

  r = 0;
  for(i = 1; i<values.length; i++) {
    if (values[i] >= values[r])
      r = i;
  }

  return values[r];
}
```

¿Es equivalente? SI

Ejemplo I

```
int max(int[] values) {
   int r, i;

   r = 0;
   for(i = 1; i<values.length; i++) {
      if (values[r] > values[r])
         r = i;
   }

   return values[r];
}
```

¿Es equivalente? NO

Ejemplo 2

```
if(x > 0) {
   if(y > x) {
      // ...
   }
}
```

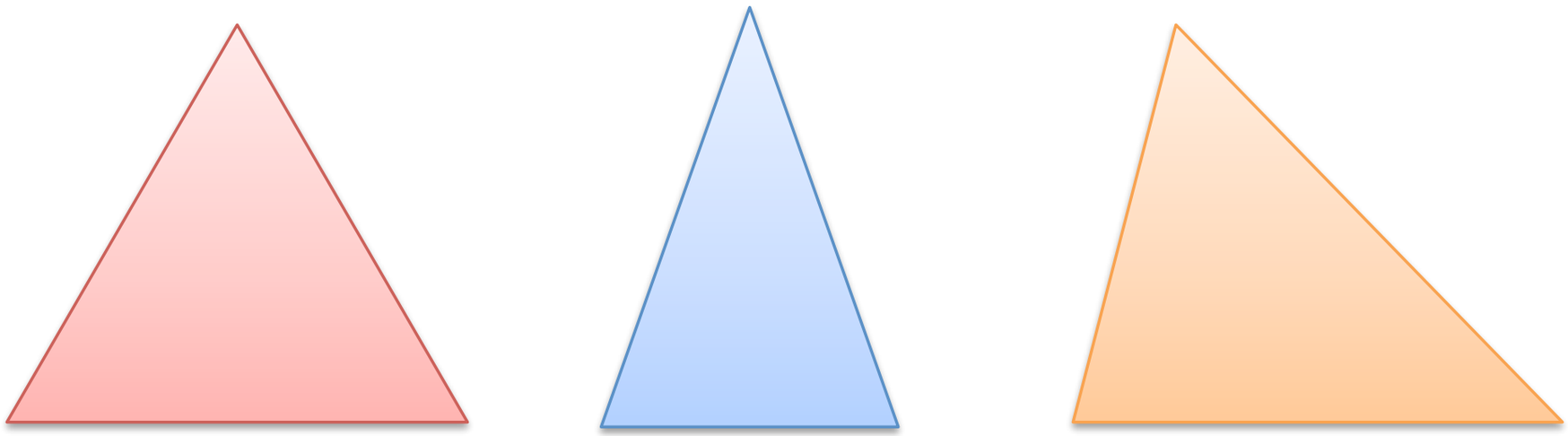
Ejemplo 2

```
if(x > 0) {
   if(y > abs(x)) {
      // ...
   }
}
```

¿Es equivalente? SI

Ejemplo

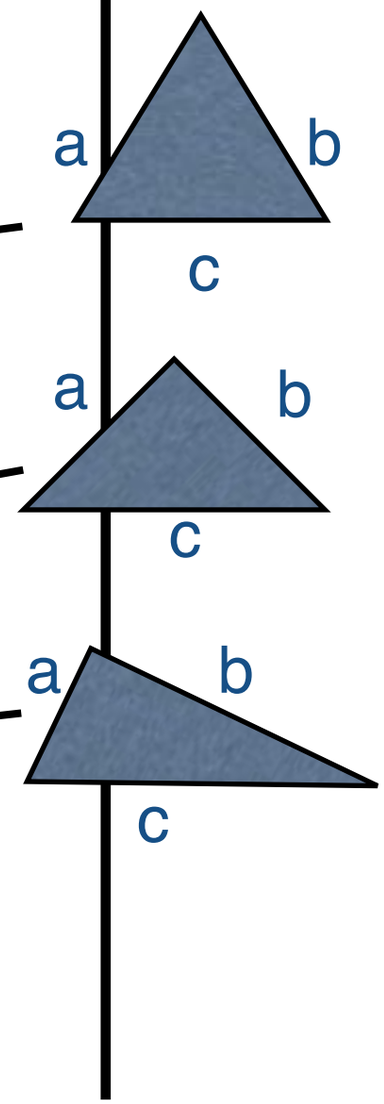
Clasificar un triángulo por el tamaño de sus lados



Equilátero Isósceles Escaleno

Ejemplo: Clasificador de Triángulos

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```



```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0)

(1, 1, 3)

(2, 2, 2)

(2, 2, 3)

(2, 3, 4)

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || !(a == c)) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0) ✓

(1, 1, 3) ✓

(2, 2, 2) ✓

(2, 2, 3) ✓

(2, 3, 4) ✗

GOOD. capable of detecting it.

```
int triangle(int a, int b, int c) {

    if (b <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0)

(1, 1, 3)

(2, 2, 2)

(2, 2, 3)

(2, 3, 4)

bad, undetected

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a++ == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(0, 0, 0) ✓

(1, 1, 3) ✓

(2, 2, 2) ✓

(2, 2, 3) ✓

(2, 3, 4) ✓

equivalente!

doesn't affect semantics.

Performance

```
int triangle(int a, int b, int c) {
  if (a <= 0 || b <= 0 || c <= 0) {
      return 4; // invalid
  }
  if (! ( a + b > c && a + c > b && b + c > a)) {
      return 4; // invalid
  }
  if (a == b && b == c) {
      return 1; // equilateral
  }
  if (a == b || b == c || a == c) {
      return 2; // isosceles
  }
  return 3;  // scalene
}
```

Problemas de Performance

- Muchos operadores de mutación posibles
  Proteum - 103 para C
  MuJava - Agrega 24 mutaciones OO
- Cada operador de mutación resulta en muchos mutantes
  - Dependen del programa bajo test
- Cada mutante debe ser compilado
- Cada test necesita ser ejecutado para cada mutante

mutants → low PERFORMANCE
↳ more code to compile & execute

cover the mutated lines & not others

only uses tests within that reach mutated lines.

Uso de Cobertura

```
int triangle(int a, int b, int c) {

    if (a <= 0 || b <= 0 || c <= 0) {
        return 4; // invalid
    }
    if (! (a + b > c && a + c > b && b + c > a)) {
        return 4; // invalid
    }
    if (a == b && b == c) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scal
}
```

(0, 0, 0)
(1, 1, 3)
(2, 2, 2)
(2, 2, 3)
(2, 3, 4)
(0, 1, 1)
(4, 3, 2)
(1, 1, 1)
(4, 3, 2)

Únicamente estos Tests ejecutan mutantes en esta línea de código!

Meta-mutantes

```
a + b > c
```

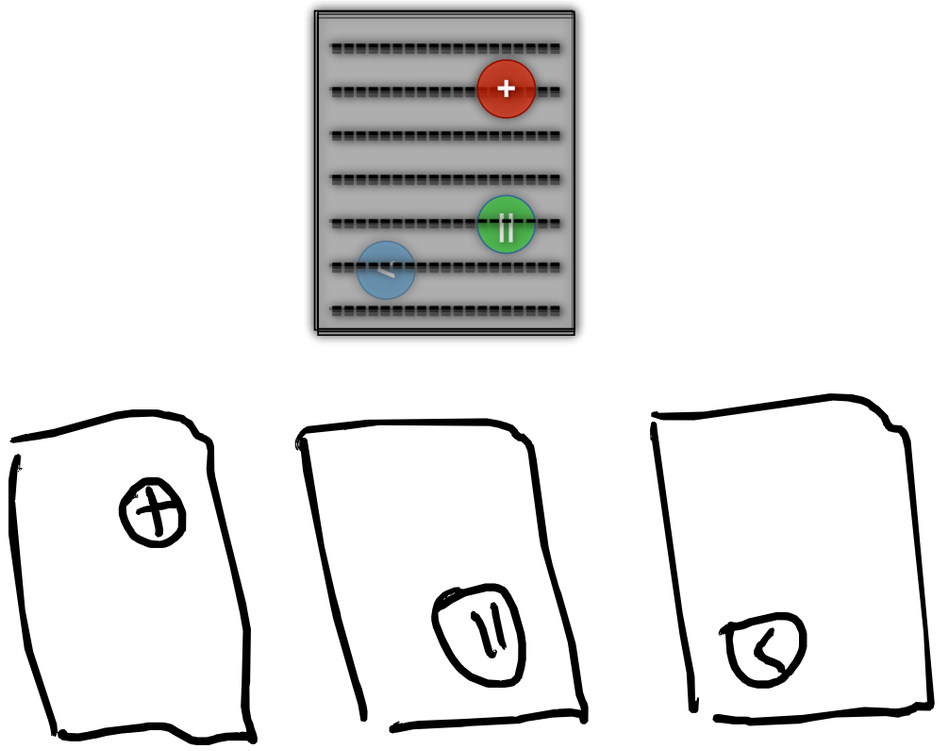
```
arithOp(a, b, 42) > c
```

```
int arithOp(int op1, int op2, int location) {
  switch(variant(location)) {
    case aoADD:   return op1 + op2;
    case aoSUB:   return op1 - op2;
    case aoMULT:  return op1 * op2;
    case aoDIV:   return op1 / op2;
    case aoMOD:   return op1 % op2;
    case aoLEFT:  return op1;
    case aoRIGHT: return op2;
  }
}
```

Seleccionar variante de mutante

Meta-mutants → generates only 1 mutated code that compiles once and tests all mutations.

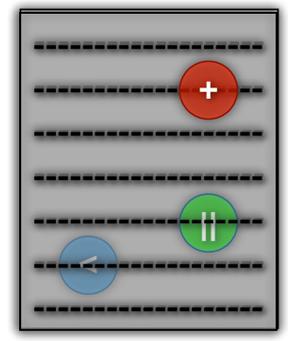
Meta-mutantes



Mutante 1:
Compilar
Ejecutar

Mutante 2:
Compilar
Ejecutar

Mutante 3:
Compilar
Ejecutar



Meta-Mutante:
Compilar
Switch para Mutante 1
Ejecutar
Switch para Mutante 2
Ejecutar
Switch para Mutante 3
Ejecutar

all compiled at once

Real world mutation testing

PIT is a state of the art **mutation testing** system, providing **gold standard test coverage** for Java and the jvm. It's fast, scalable and integrates with modern test and build tooling.

Get Started

User Group Issues Source Maven Central

Strong Mutation → detect if program output changes

Weak Mutation → detect infection (stops execution running time)

PiTest

- Utiliza meta-mutantes (no necesita recompilar N mutantes)
- Strong Mutation (test pasa o no pasa)
- Open-Source (https://github.com/hcoles/pitest)
- Aprovecha Cobertura: Ejecuta el test suite solo sobre mutantes cubiertos

ejemplo de herramiento p/ mutation analysis

PiTest

mutation Analysis is most useful for unit level testing

- Integrado con Gradle, ANT, Maven
- Puede seleccionar los operadores a aplicar
- Produce reportes HTML
  - Mutantes Vivos (ya sea que estén cubiertos o no por algún Test)
  - Mutantes Muertos (ie detectados por al menos un Test)

Taller 1: Implementación de 1 stack, uso herramiento p/ generar mutantes s/ un testsuite dado

Creación **Automática** de Casos de Tests

Programa

Casos de Test (Test Suite)

```
@Test
public void test0()  throws Throwabl
    Foo foo0 = new Foo();
    Bar bar0 = new Bar("baz3");
    bar0.coverMe(foo0);
    assertEquals(0, foo0.getX());
}
```

Criterios de Adecuación

- ¿Cómo sabemos que una test suite es "suficientemente buena"?
- Un criterio de adecuación de test es un predicade que es verdadero o falso para un par ⟨*programa, test suite*⟩
- Usualmente expresado en forma de una regla – e.g., "todos los statements deben ser ejecutados"

El Problema del Oráculo

- Ejecutar todo el programa suele no ser suficiente
- Necesitamos chequear el comportamiento funcional del programa
- Hace el programa realmente lo que queremos?

Oráculo de Delfos

Mejoras

Do fewer

- Sampling de Mutantes
- Mutación Selectiva

Do smarter

- Paralelización
- Mutación Débil
- Uso de Cobertura
- Impacto

Do faster

- Mutar bytecode
- Meta-mutantes