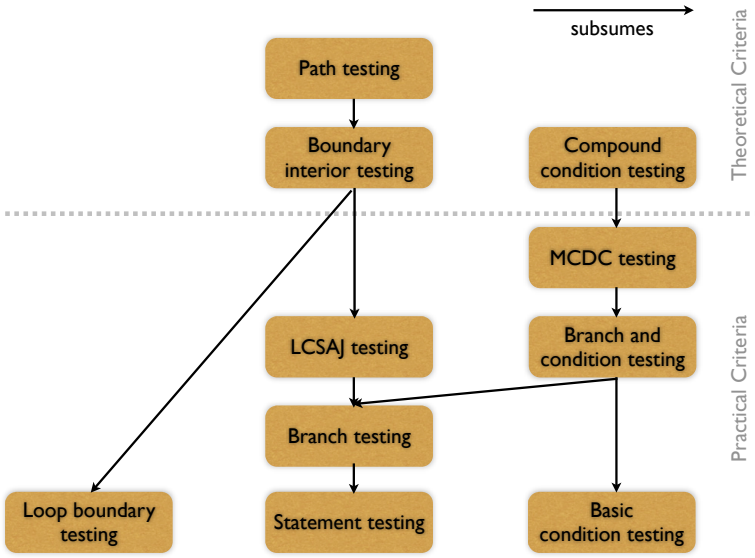
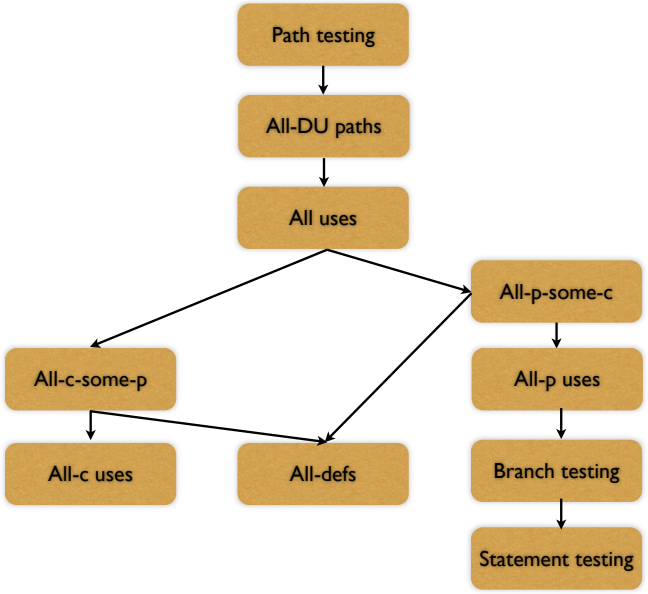
Fault-Based Testing

Generación Automática de Casos de Tests - 2018

¿Cuán buena es mi Test Suite?





```java
/**
 * 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);
}
```

```java
/**
 * 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!

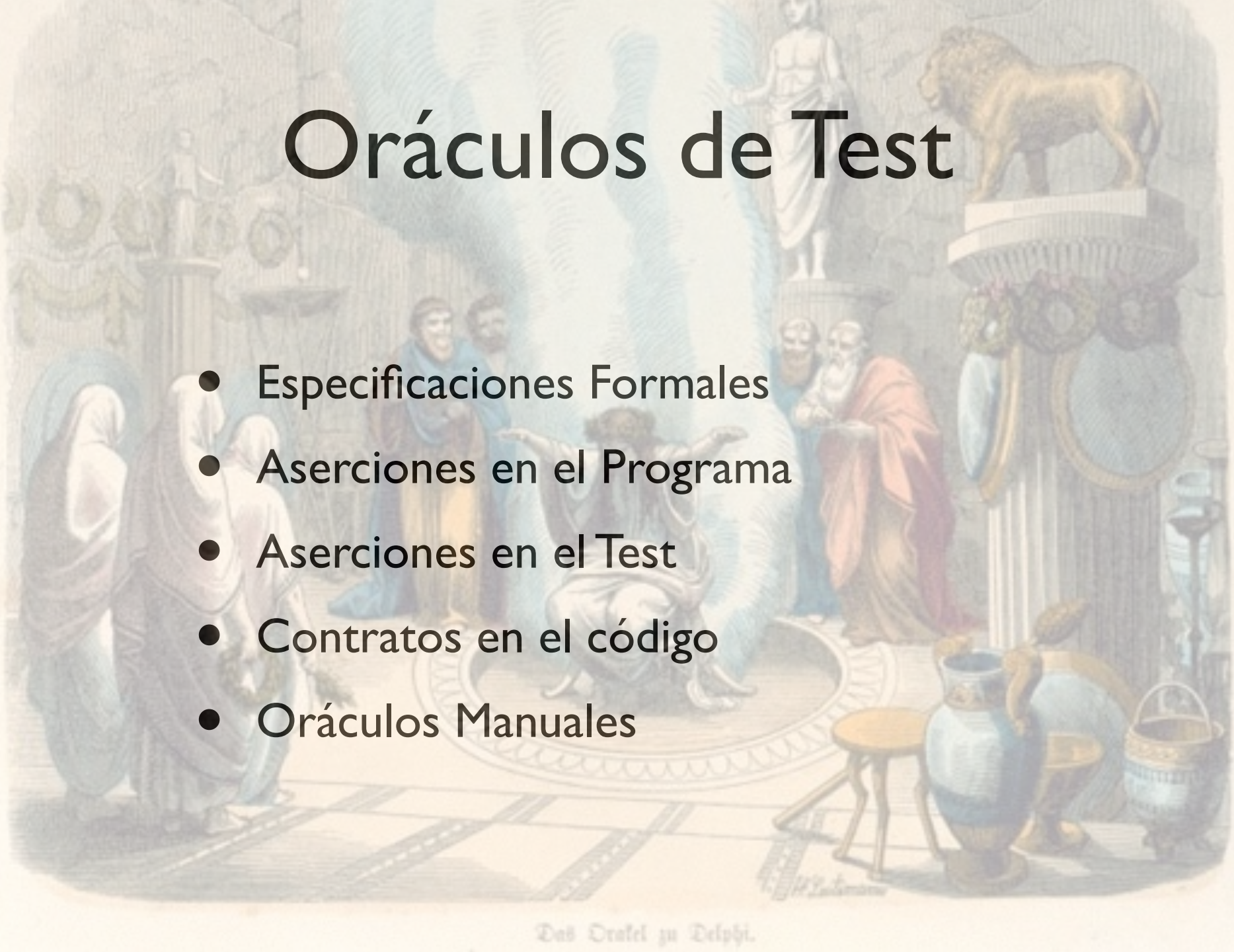
El Problema del Oráculo

- 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

Oráculos de Test

- Especificaciones Formales
- Aserciones en el Programa
- Aserciones en el Test
- Contratos en el código
- Oráculos Manuales



¿Cuán buenos son mis Tests?

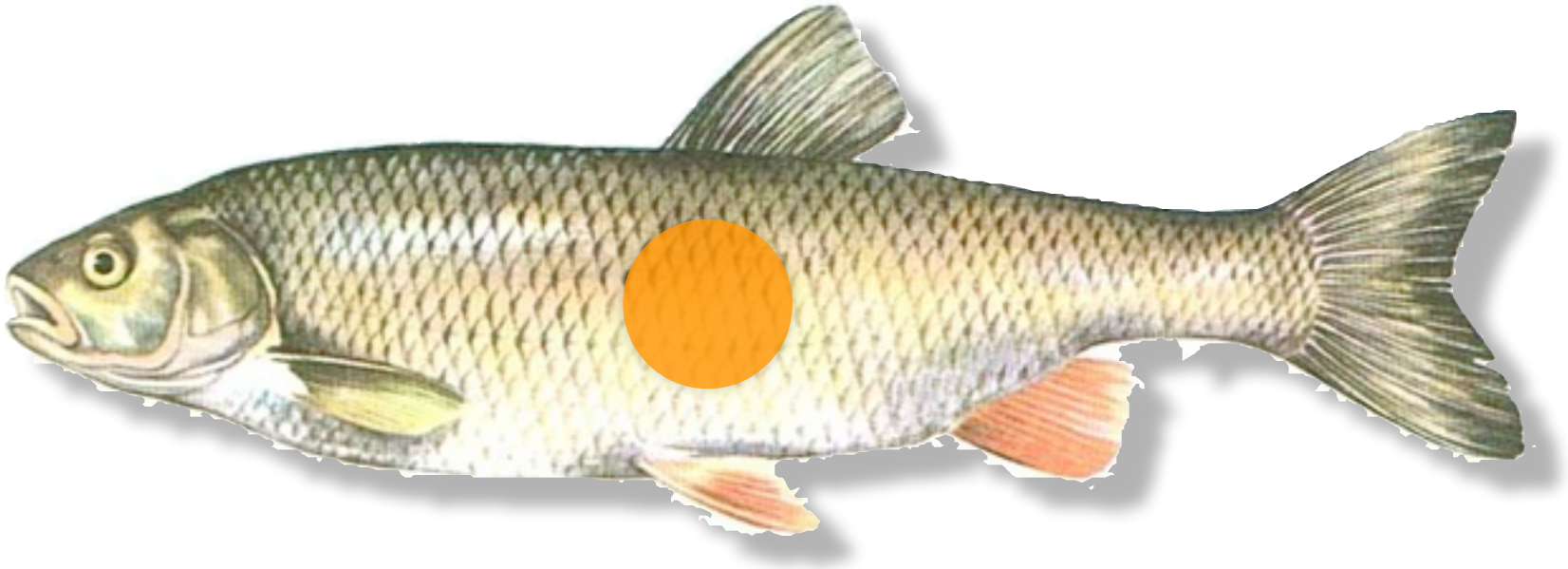
- Cobertura = cuánto de mi código es ejecutado
- Pero… ¿Cuánto de mi código es chequeado contra un comportamiento esperado?
- No sabemos donde se encuentran los bugs (defectos)
  - Pero ¡Sabemos los errores que hemos cometido en el pasado!

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

¿Cuántos peces hay en este lago?

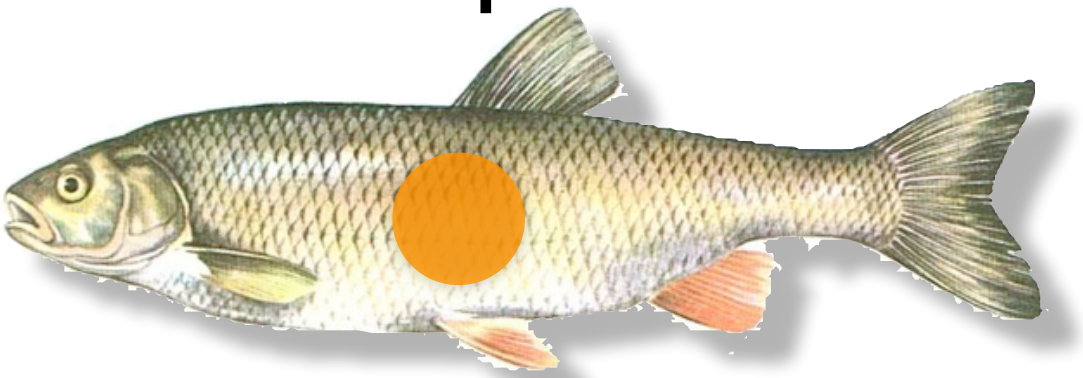
Etiquetado de Peces



- Atrapamos1000 peces y los etiquetamos
- Luego de etiquetarlos, los devolvemos al lago

A la semana siguiente, los pescadores pescan…

50 pescados etiquetados



300 pescados sin etiquetar

- ¿Puedo estimar cuántos peces hay en el Lago sin etiquetar?

Estimando la población

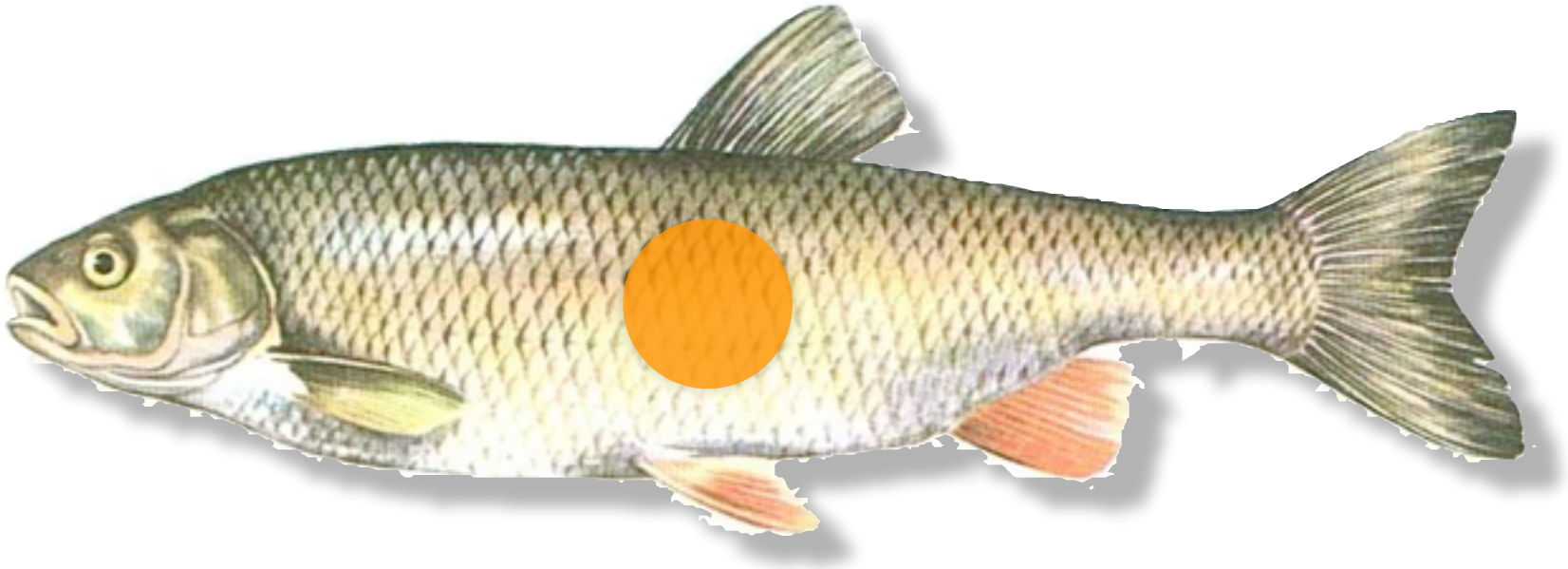
$$\frac{1000}{\text{¿Población de peces sin etiquetar?}} = \frac{50}{300}$$

- Podemos **aproximar/estimar** la población sin etiquetar
- Población sin etiquetar: aprox. 6000 peces

Estimando una Población

Estimando una Población

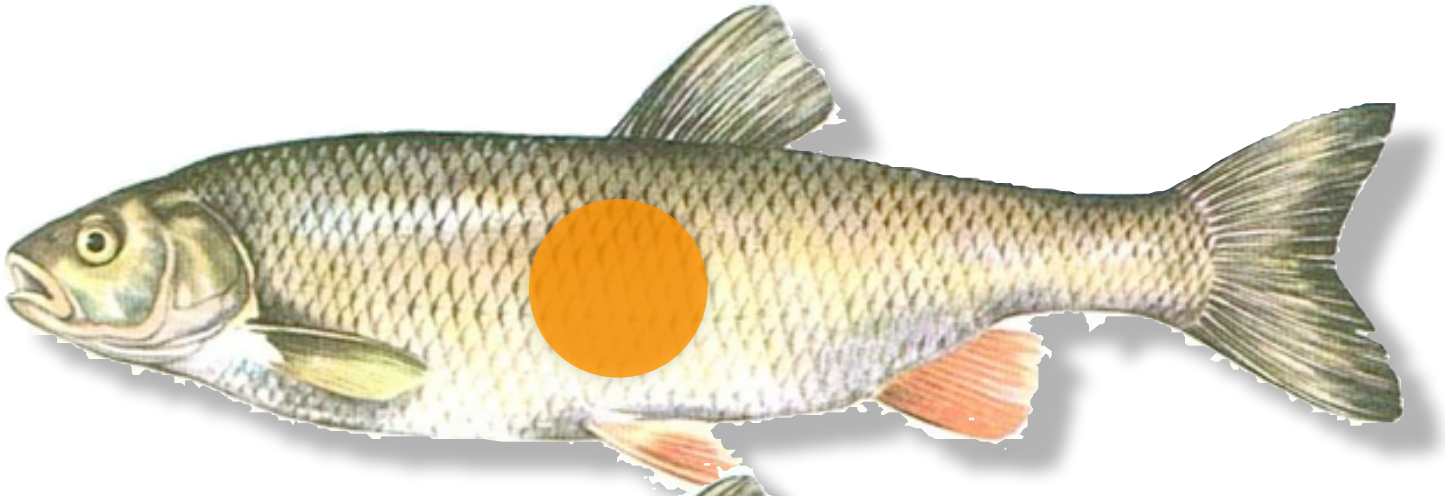
Un Mutante



- Plantamos 1000 mutaciones en nuestro programa

Contando Mutantes

50
defectos
“artificiales”
detectados



300
defectos
“naturales”
detectados

Estimamos los Defectos sin Detectar

$$\frac{1000}{\text{defectos sin detectar?}} = \frac{50}{300}$$

- Quedan aprox. 6000 defectos sin detectar en nuestro programa

Hipótesis Básicas para Estimar Defectos

- 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

```
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

Mutante

Mutantes

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

Generando Mutantes

- 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

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);
}
```

ABS - Absolute Value Insertion

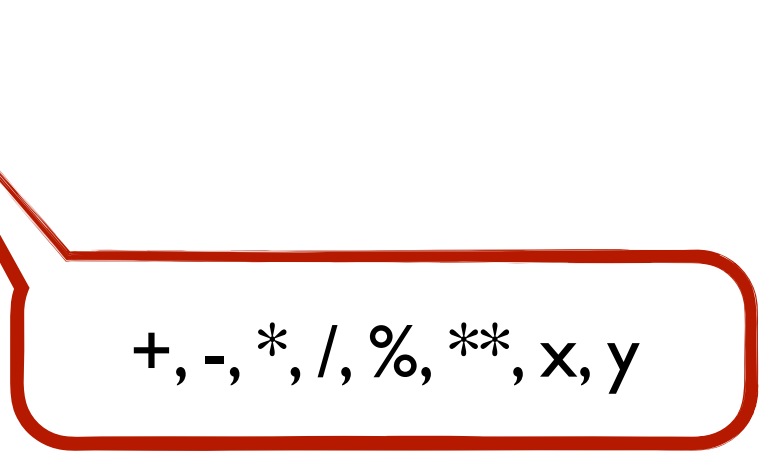
```
int gcd(int x, int y) {
  int tmp;

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

  return 0;
}
```

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;
}
```

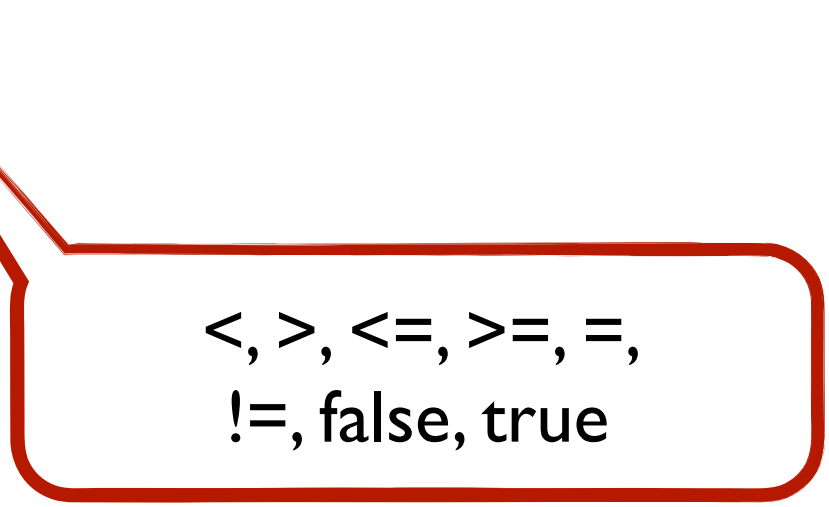


ROR - Relational Operator Replacement

```
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;
}
```



COR - Conditional Operator Replacement

```
if(a && b)
```

```
if(a || b)
if(a & b)
if(a | b)
if(a ^ b)
if(false)
if(true)
if(a)
if(b)
```

SOR - Shift Operator Replacement

```
x = m << a
```

```
x = m >> a
x = m >>> a
x = m
```

LOR - Logical Operator Replacement

```
x = m & n
```

```
x = m | n
x = m ^ n
x = m
x = n
```

ASR - Assignment Operator Replacement

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

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

  return x;
}
```

ASR - Assignment Operator Replacement

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

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

  return x;
}
```

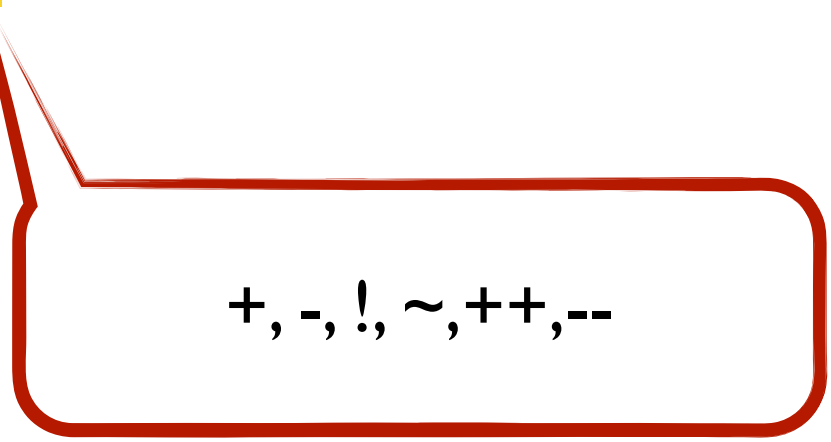
+=, -=, *=, /=, %=,
&=,

UOI - Unary Operator Insertion

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

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

  return x;
}
```

UOI - Unary Operator Insertion

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

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

  return x;
}
```



UOD - Unary Operator Deletion

```
if !(a > -b)
```

```
if (a > -b)
if !(a > b)
```

SVR - Scalar Variable Replacement

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

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

  return x;
}
```

SVR - Scalar Variable Replacement

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

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

  return x;
}
```

tmp = x % y
tmp = x % x
tmp = y % y
x = x % y
y = y % x
tmp = tmp % y
tmp = x % tmp

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	

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 dividido() {
    // ...
  }
}
```

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

Orden de Mutaciones

- First Order Mutant (FOM)
  Exactamente una mutación
- 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
- $\#HOM = 2^{\#FOM} - 1$

Hipótesis del Programador Competente

El programador escribe un programa que se encuentra en la "vecindad" del conjunto de programas correctos

Efecto Acoplamiento

Tests que distinguen todos los programas que diferen del programa correcto usando únicamente *errores sencillos* son lo suficientemente sensibles para detectar *errores más complejos*.

Mutation Testing se centra en First Order Mutants (FOMs)

Hipótesis del Programador Competente

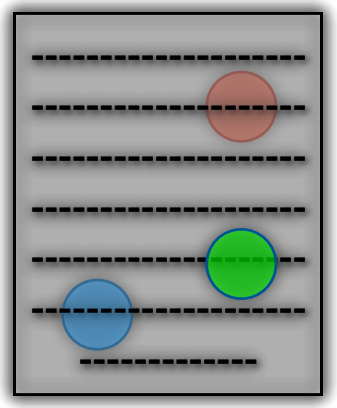
Efecto Acoplamiento

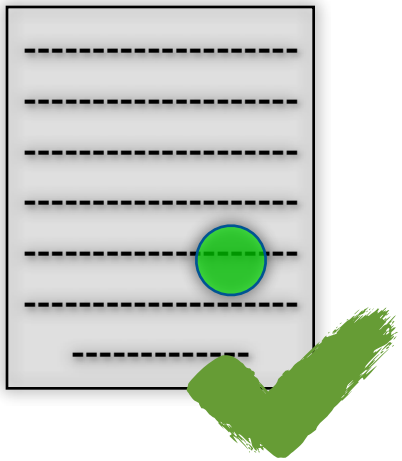
Operadores

Test 1	Test 2	Test 3

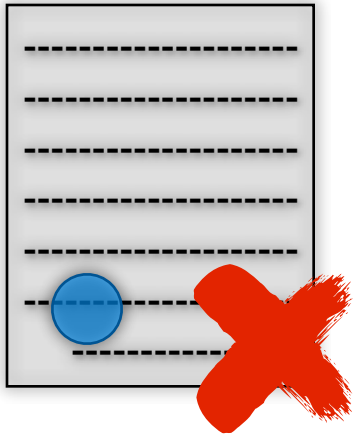
Tests

Programa Original

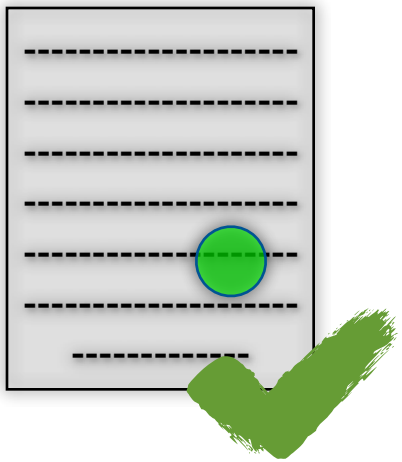


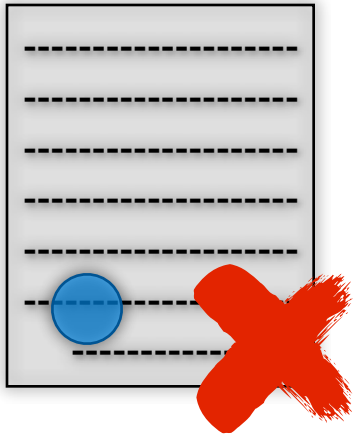


Mutante Vivo - necesitamos más Tests



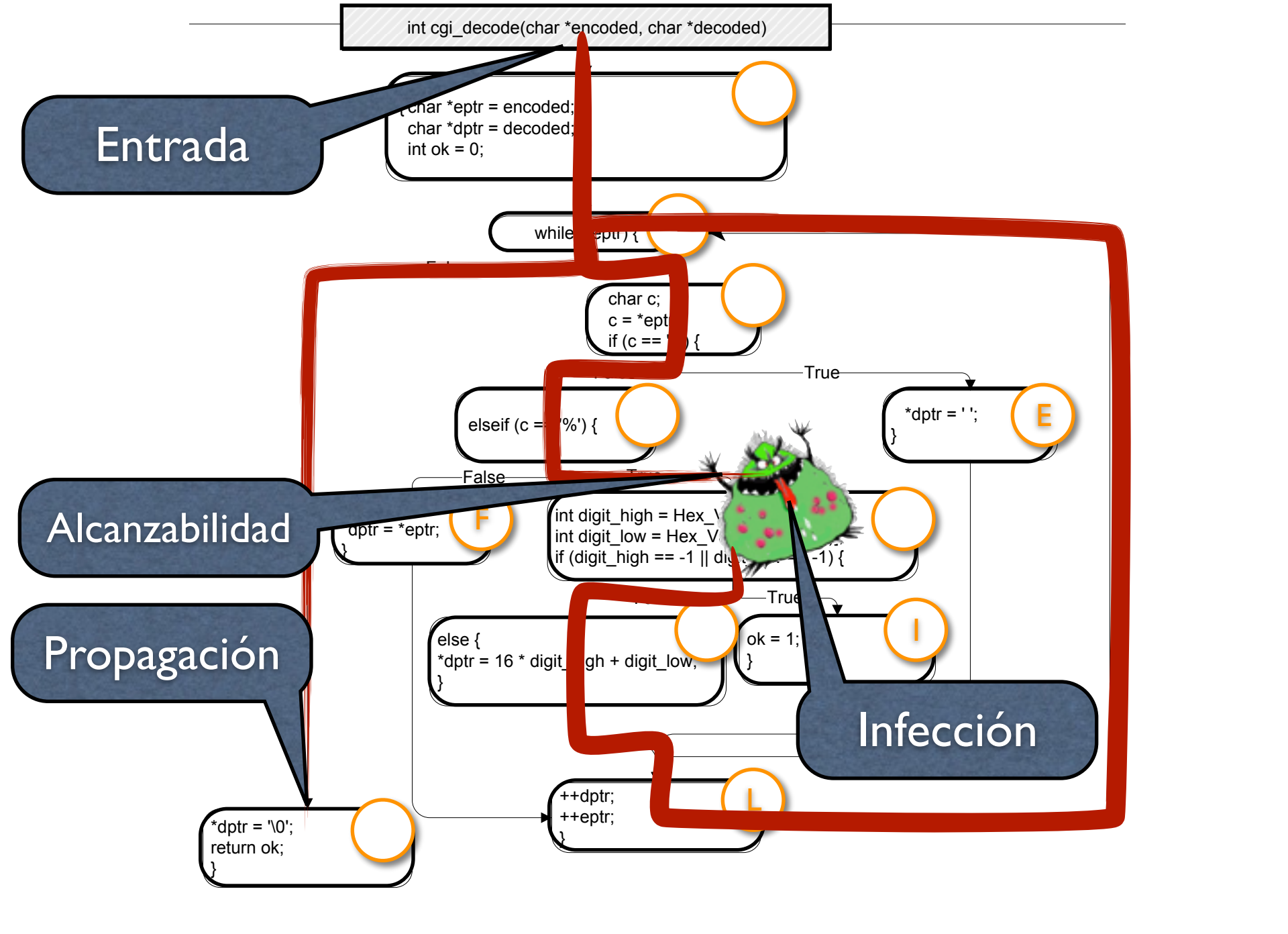
Mutante Muerto - no es más útil





Mutation Score:

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


int cgi_decode(char *encoded, char *decoded)
char *eptr = encoded;
char *dptr = decoded;
int ok = 0;
while
char c;
c = *ept
if (c ==
True
elseif (c =
'%') {
*dptr = ' ';
}
E
False
dptr = *eptr;
}
F
int digit_high = Hex_V
int digit_low = Hex_V
if (digit_high == -1 || di
-1) {
True
else {
*dptr = 16 * digit_
gh + digit_low;
}
ok = 1;
}
I
++dptr;
++eptr;
}
L
*dptr = '\0';
return ok;
}
Entrada
Alcanzabilidad
Propagación
Infección

Mutantes Equivalentes

- 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

Ejemplo 1

```
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];
}
```

Ejemplo 1

```
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 1

```
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 1

```
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) {
    // ...
  }
}
```

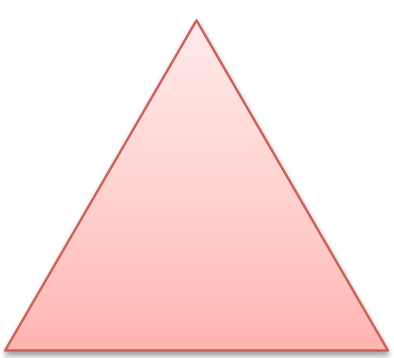
Ejemplo 2

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

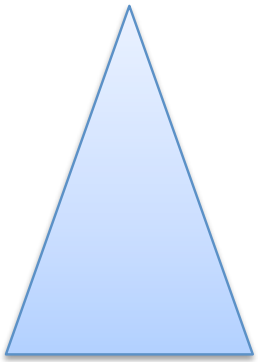
¿Es equivalente? SI

Ejemplo

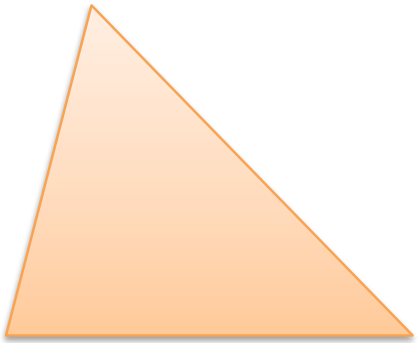
Clasificar un triángulo por el tamaño de sus lados



Equilátero



Isósceles



Escaleno

```
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) ✗

```
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) ✗

```
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) ✓

```
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) ✓

```
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) ✓

```
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 (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, 1, 1)

```
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
}
```

(4, 3, 2)

```
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
}
```

(1, 1, 1)

```
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
}
```

(4, 3, 2)

(0, 0, 0) ✓

(1, 1, 3) ✓

(2, 2, 2) ✓

(2, 2, 3) ✓

(2, 3, 4) ✓

```
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 ==
        return 2; // is
    }

    return 3;  // scale
}
```

(1, 1, 1)

```
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
}
```

(4, 3, 2)

```
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
}
```

equivalente!

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

    return 3;  // scalene
}
```

(1, 1, 1)

```
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++) {
        return 1; // equilateral
    }
    if (a == b || b == c || a == c) {
        return 2; // isosceles
    }

    return 3;  // scalene
}
```

(4, 3, 2)

(0, 0, 0) ✓

(1, 1, 3) ✓

(2, 2, 2) ✓

(2, 2, 3) ✓

(2, 3, 4) ✓

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
}
```

a + b > c

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

Mejoras

Do fewer

- Sampling de Mutantes
- Mutación Selectiva

Do smarter

- Paralelización
- Mutación Débil
- Uso de Cobertura
- Impacto

Do faster

- Mutar bytecode
- Meta-mutantes

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;  // sc
}
```

(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!

Mutación Fuerte vs. Mutación Débil

- Mutación Fuerte (Strong Mutation)
  La Mutación se ha propagado hasta algún comportamiento observable
- Mutación Débil (Weak Mutation)
  La Mutación únicamente ha afectado el estado del programa(infección)
- Compara estado interno luego de la mutación
- No garantiza propagación
- Ahorra hasta 50% del tiempo de ejecución

Mutación Fuerte vs. Mutación Débil

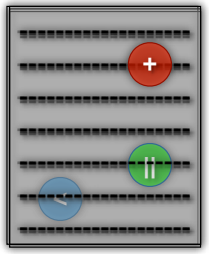
```
int gcd(int x, int y) {
  int tmp;

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

  return x;
}
```

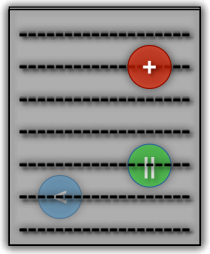
Weak mutation

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

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

  return x;
}
```

Strong mutation

Meta-mutantes





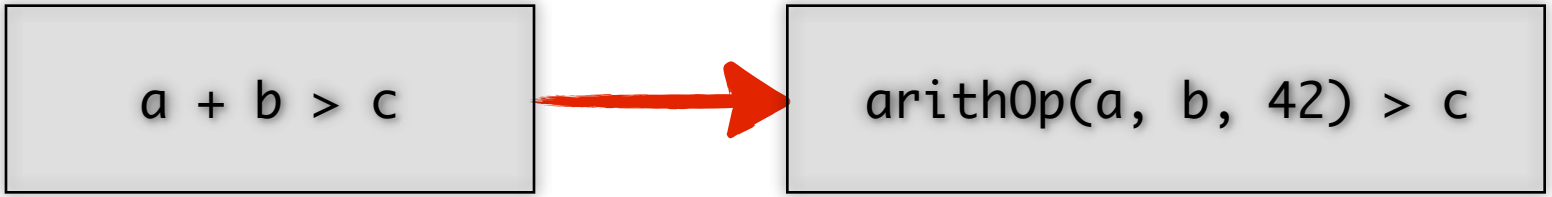
Mutante 1:
Compilar
Ejecutar

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

Meta-mutantes



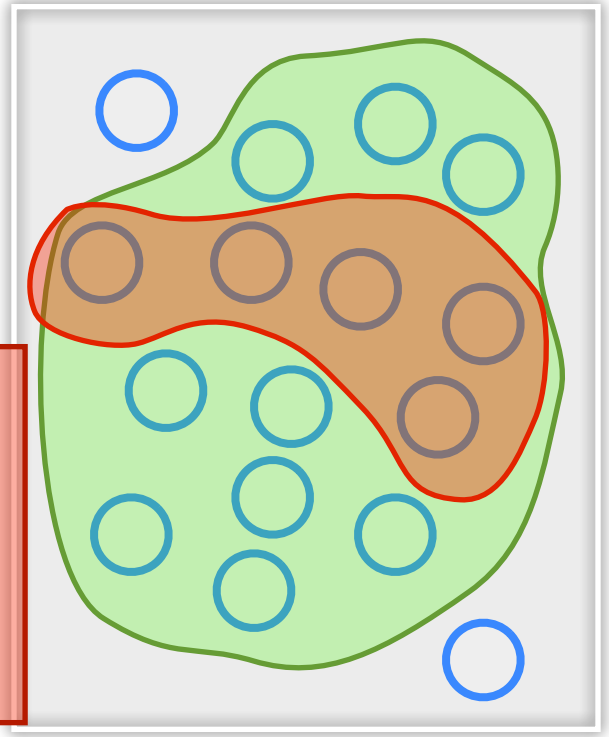

```
int arithOp(int op1, int op2, int location) {
  switch(variant(location)) {
    case aoADD:   return op1 + op2;
    case aoSUB:   return op1 - op2;
    case aoMULT:  return op1 * op2;
    case aoDIV:   return op1 / op2;
    case aoMOD:   return op1 % op2;
    case aoLEFT:  return op1;
    case aoRIGHT: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

Seleccionar variante de mutante

Mutación Selectiva

Conjunto completo:
Test cases que pueden matar todos los mutantes

Subconjunto suficiente:
Test cases que maten a **estos** mutantes serán capaces de matar a **todos** mutants

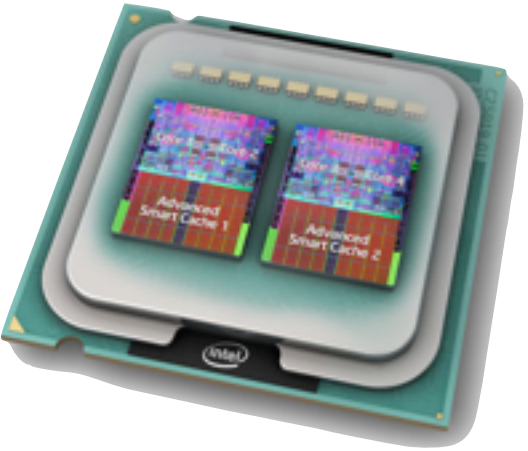


Mutación Selectiva

- Usar únicamente un **subconjunto** de operadores de mutación en lugar de todos los operadores
- Un subconjunto lo suficientemente rico puede detectar >99% de todos los mutantes
- Experimentalmente:
  - ABS, AOR, COR, ROR, UOI

Do Smarter/Faster

Mutation testing es inherentemente paralelizable




```
6:  iload_1 // a
7:  ifeq 14
10: iload_2 // b
11: ifne 23
14: iload_3 // c
15: ifeq 34
// ...
23: invokevirtual  #4;
// ...
34: invokevirtual  #5;
```

La mutación de bytecode/assembly evita la recompilación

Samplear subconjunto de mutantes

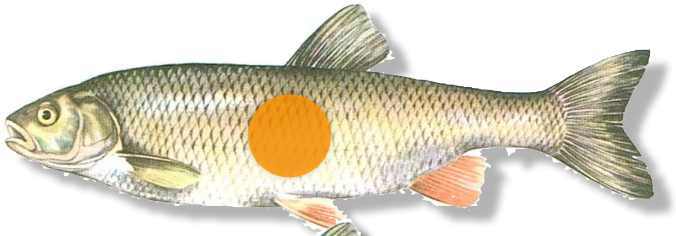
El Problema del Oráculo

- 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

Contando Mutantes

50 defectos "artificiales" detectados



300 defectos "naturales" detectados

Mutation Testing se centra en First Order Mutants (FOMs)

Hipótesis del Programador Competente

Efecto Acoplamiento

Mejoras

Do fewer

- Sampling de Mutantes
- Mutación Selectiva

Do smarter

- Paralelización
- Mutación Débil
- Uso de Cobertura
- Impacto

Do faster

- Mutar bytecode
- Meta-mutantes


Break

Taller: PiTest

Generación Automática de Casos de Test - 2017

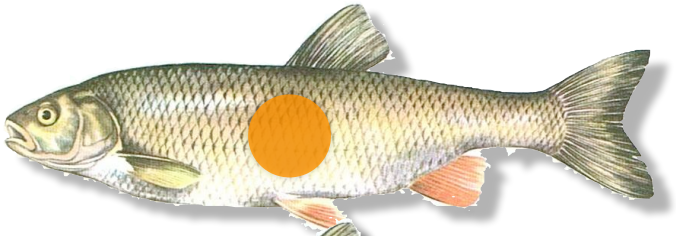
El Problema del Oráculo

- 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

Contando Mutantes

50 defectos "artificiales" detectados



300 defectos "naturales" detectados

Mutation Testing se centra en First Order Mutants (FOMs)

Hipótesis del Programador Competente

Efecto Acoplamiento

Mejoras

Do fewer

- Sampling de Mutantes
- Mutación Selectiva

Do smarter

- Paralelización
- Mutación Débil
- Uso de Cobertura
- Impacto

Do faster

- Mutar bytecode
- Meta-mutantes

Presentado en ISSTA 2016

PIT a Practical Mutation Testing Tool for Java (Demo)

Henry Coles
NCR, Edinburgh
henry@pitest.org

Thomas Laurent
Lero@UCD, Ireland & École Centrale de Nantes, France
thomas.laurent@eleves.ec-nantes.fr

Christopher Henard
University of Luxembourg
christopher.henard@uni.lu

Mike Papadakis
University of Luxembourg
michail.papadakis@uni.lu

Anthony Ventresque
Lero@UCD, UCD, Ireland
anthony.ventresque@ucd.ie

ABSTRACT

Mutation testing introduces artificial defects to measure the adequacy of testing. In case candidate tests can distinguish the behaviour of mutants from that of the original program, they are considered of good quality – otherwise developers need to design new tests. While, this method has been shown to be effective, industry-scale code challenges its applicability due to the sheer number of mutants and test executions it requires. In this paper we present PIT, a practical mutation testing tool for Java, applicable on real-world codebases. PIT is fast since it operates on bytecode and optimises mutant executions. It is also robust and well integrated with development tools, as it can be invoked through a command line interface, Ant or Maven. PIT is also open source and hence, publicly available at http://pitest.org/

CCS Concepts

•**Software and its engineering** → **Software testing**

much of the source code is covered (i.e., merely executed) by the tests. Coverage metrics imply that the more coverage the merrier. This notion, while widely applied, has a major drawback; it only checks if a line/instruction is tested, not how well it is tested.

Mutation testing [4] is a technique that gives a better understanding of what the tests exercise on the program under analysis. Mutation introduces defects, in the form of small code modifications, which should result in an abnormal behaviour when exercised by tests. If the tests fail to expose the defects then the testers/developers can reasonably infer that the tests are not checking every possible behaviour and that the tests need to be improved.

This paper presents PIT, a mutation testing system for Java. PIT is considerably fast as it manipulates bytecode and runs only the tests that have a chances to kill the used mutants (i.e., the tests that execute the instruction where the mutant is located). PIT's major advantage is that it is robust, easy to use and well integrated with development

pitest.org

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

PiTest

- Utiliza meta-mutantes (no necesita recompilar N mutantes)
- Strong Mutation (test pasa o no pasa)
- Open-Source (https://github.com/hcoles/pitest)

PiTest

- 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)

Operadores Activos por Default

- Conditionals Boundary Mutator
- Increments Mutator
- Invert Negatives Mutator
- Math Mutator
- Negate Conditionals Mutator
- Return Values Mutator
- Void Method Calls Mutator

Operadores no Activos por Default (activables)

- Constructor Calls Mutator
- Inline Constant Mutator
- Non Void Method Calls Mutator
- Remove Conditionals Mutator
- Experimental Member Variable Mutator
- Experimental Switch Mutator

CONDITIONALS_BOUNDARY

- < es reemplazado por <=
- <= es reemplazado por <
- > es reemplazado por >=
- >= es reemplazado por >

Taller #2: PiTest

- Vamos a usar el Test Suite que crearon en el Taller #1 para StackAr
- Vamos a usar PiTest como un plugin de Maven
- Vamos a mejorar el Test Suite para hacerlo más robusto wrt Mutation Testing

- Apache Maven es un *"software project management and comprehension tool"*
- Nos permite hacer deployment de proyectos (símil Makefiles)
- A fines del Taller, sólo es una línea de comando

- Descargar el .zip file en
  - www.dc.uba.ar/autotest/descargas/Taller02-pitest/stackar.zip
- Descomprimirlo y moverse a la carpeta "stackar"
- Ejecutar desde una terminal:
  - $ mvn clean install org.pitest:pitest-maven:mutationCoverage

- ¿Qué hizo Maven?
  - Descargo las dependencies (librerías) necesarias para el proyecto (PiTest,JUnit)
  - Compilo el código fuente
  - Ejecutó los JUNITs
  - Ejecutó PiTest y generó los reportes HTML

Reporte de Mutantes Muertos/Vivos de PiTest

- En la carpeta stackar/target encontrará:
  - Una carpeta “pit-reports” con el reporte de la actividad de PiTest
    - $ open target/pit-reports/*/index.html

Pit Test Coverage Report

Package Summary

org.autotest

Number of Classes	Line Coverage		Mutation Coverage	
1	0%	0/52	0%	0/37

Breakdown by Class

Name	Line Coverage		Mutation Coverage	
StackAr.java	0%	0/52	0%	0/37

Report generated by PIT 1.1.10

StackAr.java

```
1   package org.autotest;
2
3   import java.util.Arrays;
4
5   public class StackAr {
6
7           private final static int DEFAULT_CAPACITY = 10;
8
9           private final Object[] elems;
10
11          private int readIndex = -1;
12
13          public StackAr() {
14                  this(DEFAULT_CAPACITY);
15          }
16
17          public StackAr(int capacity) throws IllegalArgumentException {
18  2               if (capacity < 0) {
19                          throw new IllegalArgumentException();
20                  }
21                  this.elems = new Object[capacity];
22          }
23
24          public int size() {
25  2               return readIndex+1;
26          }
27
28          public boolean isEmpty() {
29  2               return size() == 0;
30          }
```

Enunciado (1)

1. Reemplazar TestStackAr.java con el test suite creado para el Taller #1
   - Copiar su archivo TestStackAr.java a la carpeta stackar/src/test/java/org/autotest/

Enunciado (2)

2. Ejecutar PiTest usando el nuevo test suite:
   - Ejecutar el comando:
     - $ mvn clean install org.pitest:pitest-maven:mutationCoverage
   - ¿Cuántas líneas cubiertas reporta PiTest?
   - ¿Cuántos mutantes vivos reporta PiTest?
   - ¿Cuál es el mutation score (mutantes vivos / mutantes totales) que reporta PiTest?

Enunciado (3)

3. Extender el test suite TestStackAr para obtener el mejor mutation score posible con PiTest
   - ¿Cuál es el mejor mutation score que pudo obtener?
   - ¿Cuántos mutantes equivalentes encontró?