Testing basado en Mutación

Testing basado en Mutación: Testing de mutación | Mutantes vivos/muertos | Mutation Score | Operadores de mutación |Herramientas automáticas

Criterios de Testing

La forma más efectiva de hacer testing es de manera exhaustiva, pero es en general impracticable.

Luego, se necesita algún criterio para seleccionar tests.

Un criterio de testing es una mecanismo para decidir si una suite es adecuada o no.

Una forma de saber cuando construí suficientes tests

Lo visto hasta ahora…

Criterios de Caja Negra: Particionado en Clases de equivalencias, valores bordes.

Criterios de Caja Blanca: Cobertura de Sentencia, Cobertura de Decisión, etc.

Testing basado en Mutación

Es un enfoque algo diferente a los vistos anteriormente.

Una *suite de tests es adecuada*, si es efectiva para descubrir defectos en el programa.

Medir número de defectos detectados

El programa efectivamente tiene defectos?

Cuantos?

En que consiste este criterio?

Los defectos se “inyectan” automáticamente en el código “mutando” operaciones del mismo

Cada variante del código original es un **Mutante**.

Según este criterio, **una suite de test es adecuada si es efectiva para descubrir defectos** inyectados.

En que consiste este criterio?

- Se inyectan defectos en el código para crear mutantes
  - Los mutantes son programas que deben compilar con alguna variación sintáctica con respecto al programa original.
  - Una suite de test es “adecuada” si puede “distinguir” el programa de una conjunto **mutantes**

¿Qué es un mutante?

Consideremos el siguiente programa:

```
public static boolean bisiesto(int a) {
    boolean b = false;
    if ((a%4==0) && (a%100!= 0))
            b = true;
    if(a%400==0)
        b = true;
    return b;
}
```

```
public static boolean bisiesto(int a) {
    boolean b = false;
    if ((a%4==0) || (a%100!= 0))
            b = true;
    if(a%400==0)
        b = true;
    return b;
}
```

Mutante

Testing de Mutación

Consideremos el siguiente pr

```
public static boolean bisiesto(int a) {
    boolean b = false;
    if ((a%4==0) && (a%100!= 0))
            b = true;
    if(a%400==0)
        b = true;
    return b;
}
```

```
@Test
public void testBisiesto1() {
    int a =2008;
    boolean b  = SimpleExamples.bisiesto(a);
    assertEquals(true, b);

}
```

```
@Test
public void testBisiesto2() {
    int a =2000;
    boolean b  = SimpleExamples.bisiesto(a);
    assertEquals(true, b);
}
```

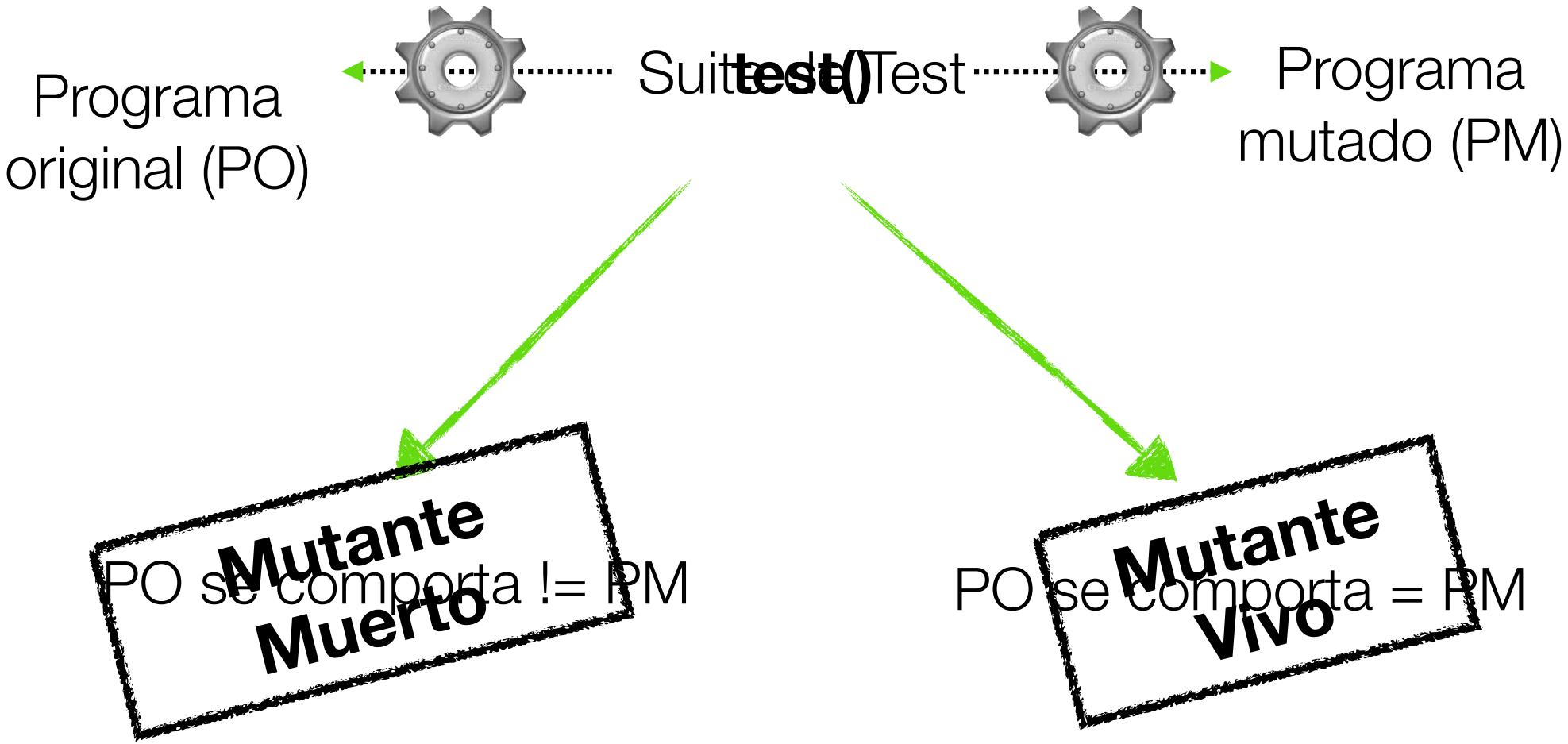
```
public static boolean bisiesto(int a) {
    boolean b = false;
    if ((a%4==0) || (a%100!= 0))
            b = true;
    if(a%400==0)
        b = true;
    return b;
}
```

Mutante

```
@Test
public void testBisiesto3() {
    int a =2005;
    boolean b  = SimpleExamples.bisiesto(a);
    assertEquals(false, b);

}
```

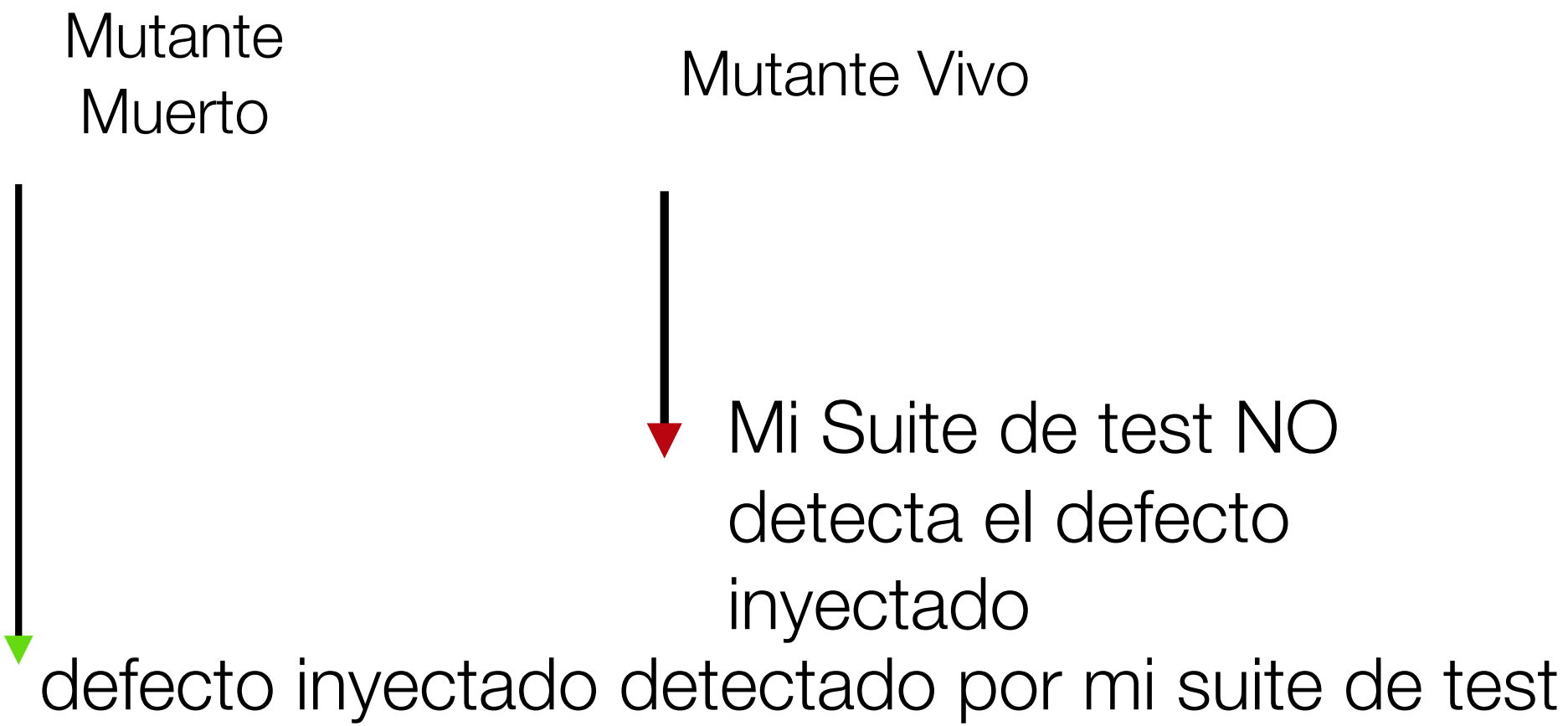
Testing de Mutación



Testing de Mutación

No matan al menos a un mutante son considerados no efectivos y pueden ser eliminados

Testing de Mutación



Testing de Mutación

Mutante Vivo

La suite de tests es inadecuada, es decir no es útil para detectar los errores insertados en el código, por ejemplo, porque no ejercita la parte del co´digo mutada.
El mutante es equivalente al programa original. La mutación nos deja un programa que se comporta exactamente igual al original.

Dos razones:

Mutante equivalente a programa original
Suite de Test inadecuada

Mutaciones equivalentes

Los siguientes programas son lógicamente equivalentes:

```
int i = 2;
if ( i >= 1 ) {
    return "foo";
}

//...
int i = 2;
if ( i > 1 ) {
    return "foo";
}
```

Suite de Test Inadecuada

Ningún test en la suite mata
el mutante

Modelo RIP

Defectos detectados por mutantes.

Reachability: debe alcanzarse la línea de código que contiene la mutación

Infection: después de ejecutar la mutación el programa mutante debe quedar en un estado de error.

Propagation: el programa mutante debe propagar el error, y producir un estado final incorrecto

Mutación Fuerte y Débil

Mutación débil “relaja” la definición de matar un mutante, exige **R**eachability y **I**nfection pero NO **P**ropagation.

Se chequea el estado interno inmediatamente después de la ejecución de la línea mutada.

Mutación Fuerte: exige que la falla provocada por la línea mutada se propague (**P**ropagation) a la salida.

Ejemplo

X=-6

```
public static boolean isEven(int x){
    if(x<0)
        x=0;
    if(x/2 == x/2.0f)
        return true;
    else
        return false;
}
```

Mutante debilmente muerto

Esperado =true, obtenido =true

```
public static boolean isEven(int x){
    if(x<0)
        x=0-x;
    if(x/2 == x/2.0f)
        return true;
    else
        return false;

}
```

Alcanzabilidad: X<0

Infección X =0

Programa original

La falla no se **propaga** a la salida

Ejemplo

```
public static boolean isEven(int x){
    if(x<0)
        x=0;
    if(x/2 == x/2.0f)
        return true;
    else
        return false;
}
```

Mutante Fuertemente muerto

Alcanzabilidad: X<0

Infección X =0

Propagación: X es impar

```
public static boolean isEven(int x){
    if(x<0)
        x=0-x;
    if(x/2 == x/2.0f)
        return true;
    else
        return false;

}
```

Programa original

La falla se **propaga** a la salida

Objetivo del Testing de Mutación

“matar” todos los mutantes(no equivalentes) usando tests

un mutante está muerto si existe un test de la suite que no falla en el original, pero si en el mutante.

Cobertura de Mutación

Para cada mutante *m*, la suite de test tiene al menos un test que mata al mutante.

Mutation Score

Porcentaje de Mutantes que una *suite de test* puede “distinguir” del programa original

El *Mutation Score* es un indicador de la efectividad, para detectar errores, de una *suite de test*.

Operadores de Mutación

Reglas que especifican variaciones sintácticas en un programa.

Operadores de Mutación

Mutaciones a nivel de **métodos**

producen cambios en el programa mediante inserción, reemplazo o eliminación de operadores primitivos del lenguaje: <, >, ==, etc.

Mutaciones a nivel de **clases**

producen cambios haciendo uso de características específicas de los lenguajes orientados a objetos: herencia, polimorfismo, etc.

Operadores de Mutación: Ejemplos

a + b -> a-b

a || b -> a && b

a== b -> a>=b , a==b->false/true

a> b -> a<b

Operadores de Mutación

Diseñados específicamente para cada lenguaje

Un número grande de operadores implica un número grande de mutantes.

Obtener menos mutantes:

- Seleccionar de manera Random algunos mutantes
- Aplicar solo operadores de mutación “efectivos”

Algunas Herramientas Automáticas para Java

- Jumble
- Judy
- Mujava
- Pitest
- Major

Una herramienta de mutación Pitest (http://pitest.org/)

Corre suites JUnit:

- mide cobertura de sentencia (para seleccionar tests dirigidos al código mutado)
- mide score (% de mutantes muertos)

Reporta los mutantes:

- Vivos cubiertos
- Vivos No cubiertos (ningún test ejercita el defecto introducido
- time-outs: mutantes que podrían causar loops

Material de consulta recomendado

Introducción to software testing, Ammann y Offutt, capítulo 1 (1.2). capítulo 5 (5.2.2)

http://pitest.org