Verificación y Validación de Software

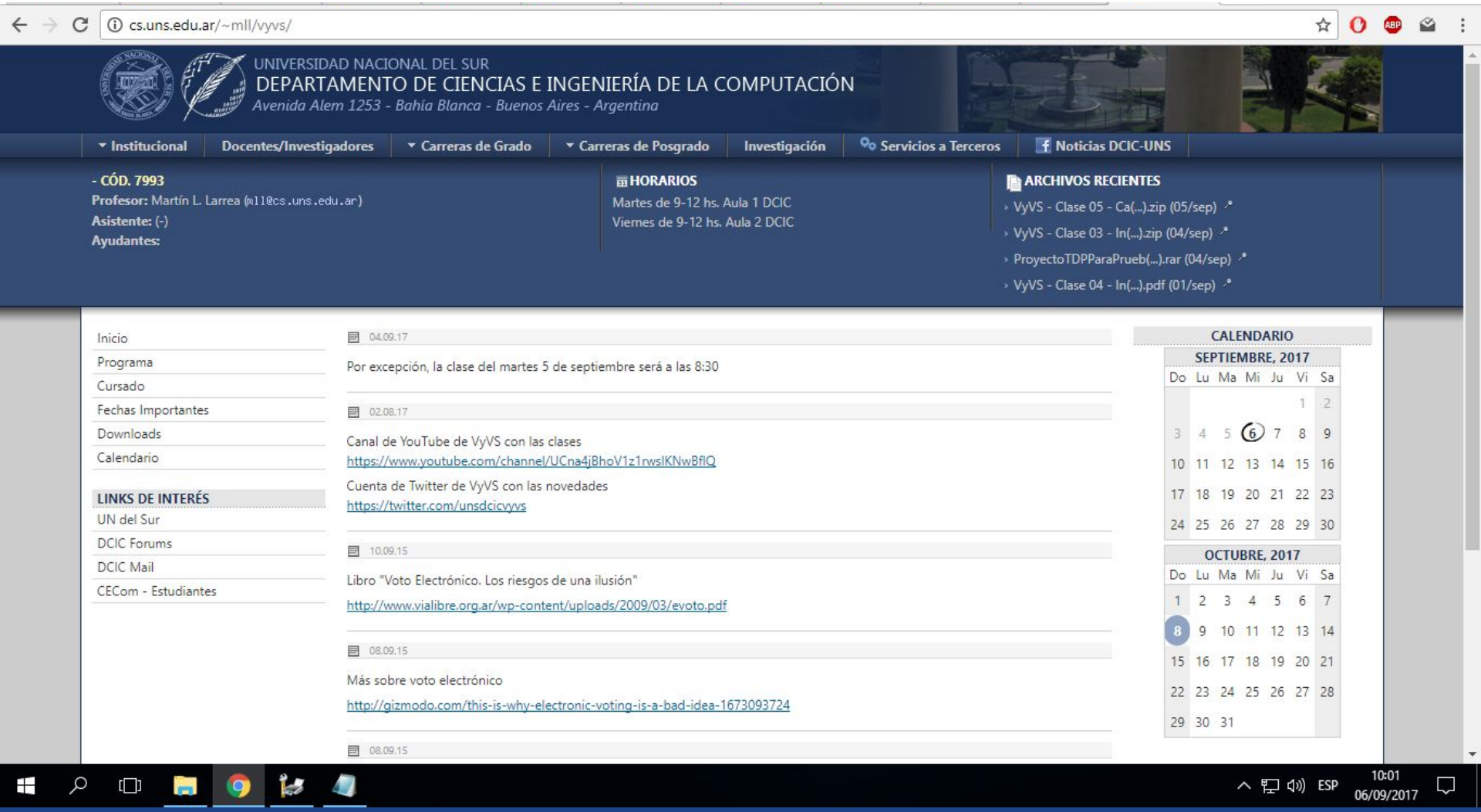
Tecnología de Programación
Martín Larrea

Verificación y Validación de Software

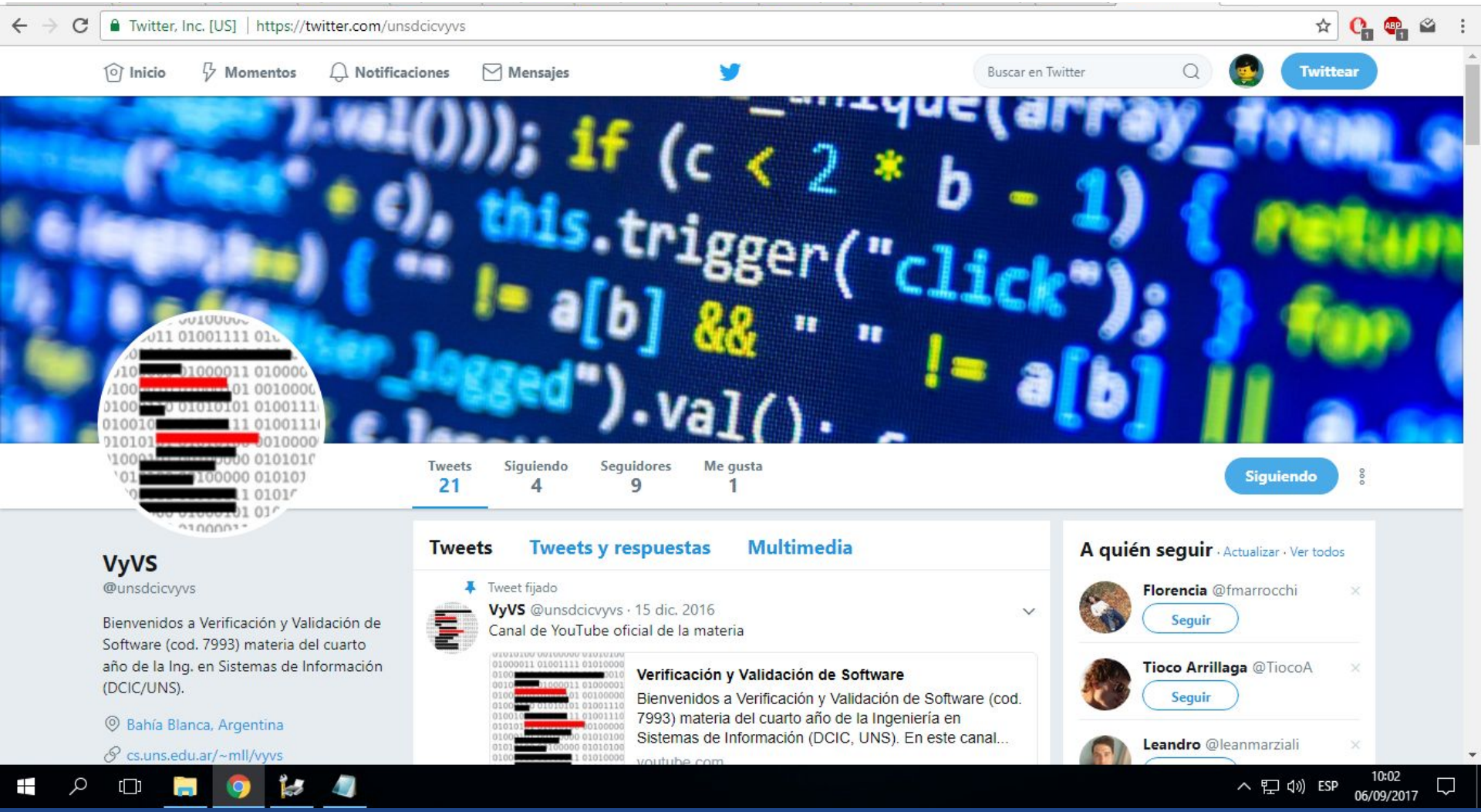
Martín L. Larrea, profesor de la materia "Verificación y Validación de Software" para la Ingeniería en Sistemas de Información

"Testing de Software" materia optativa para la Licenciatura en Ciencias de la Computación e Ingeniería en Computación

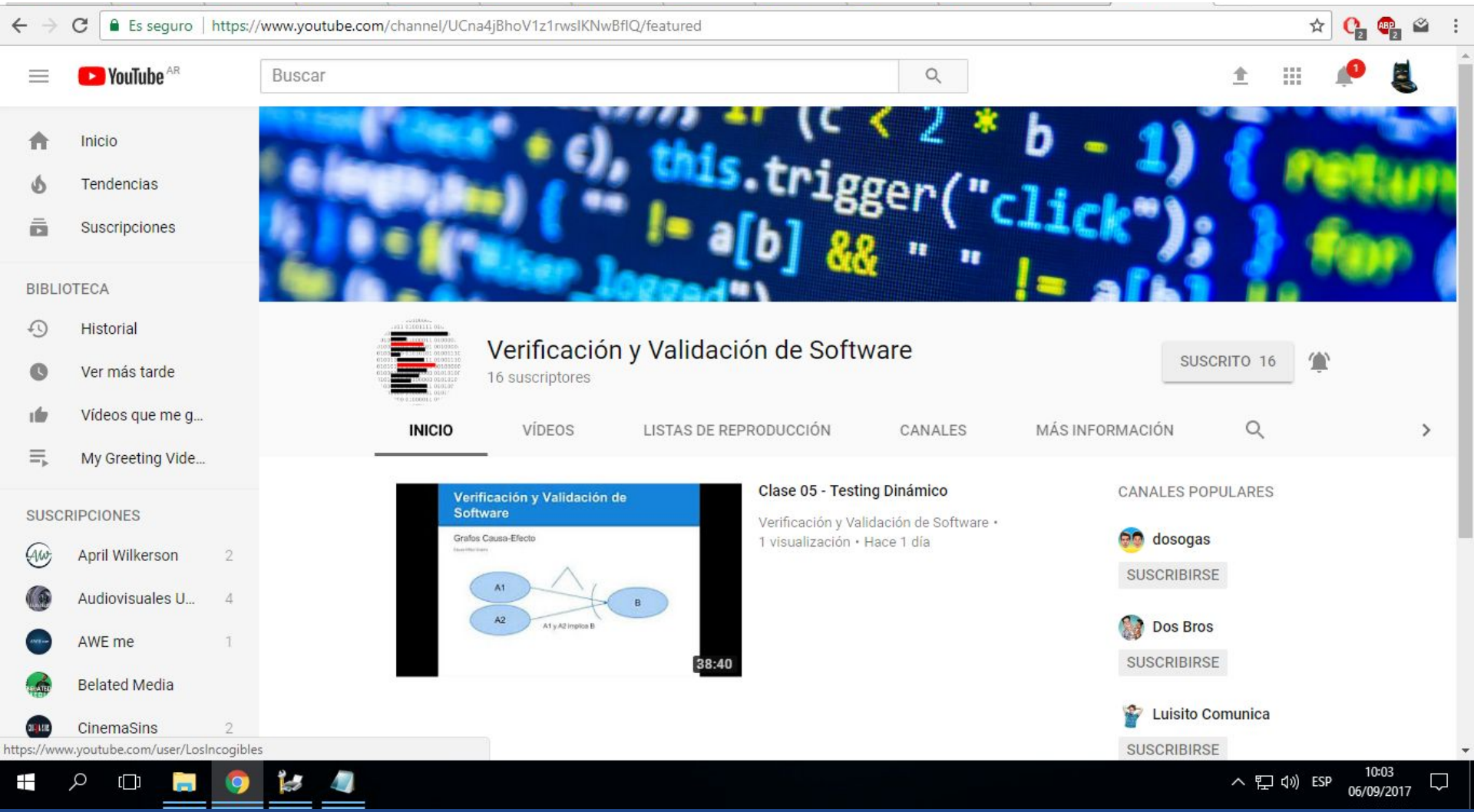
Verificación y Validación de Software



Verificación y Validación de Software



Verificación y Validación de Software

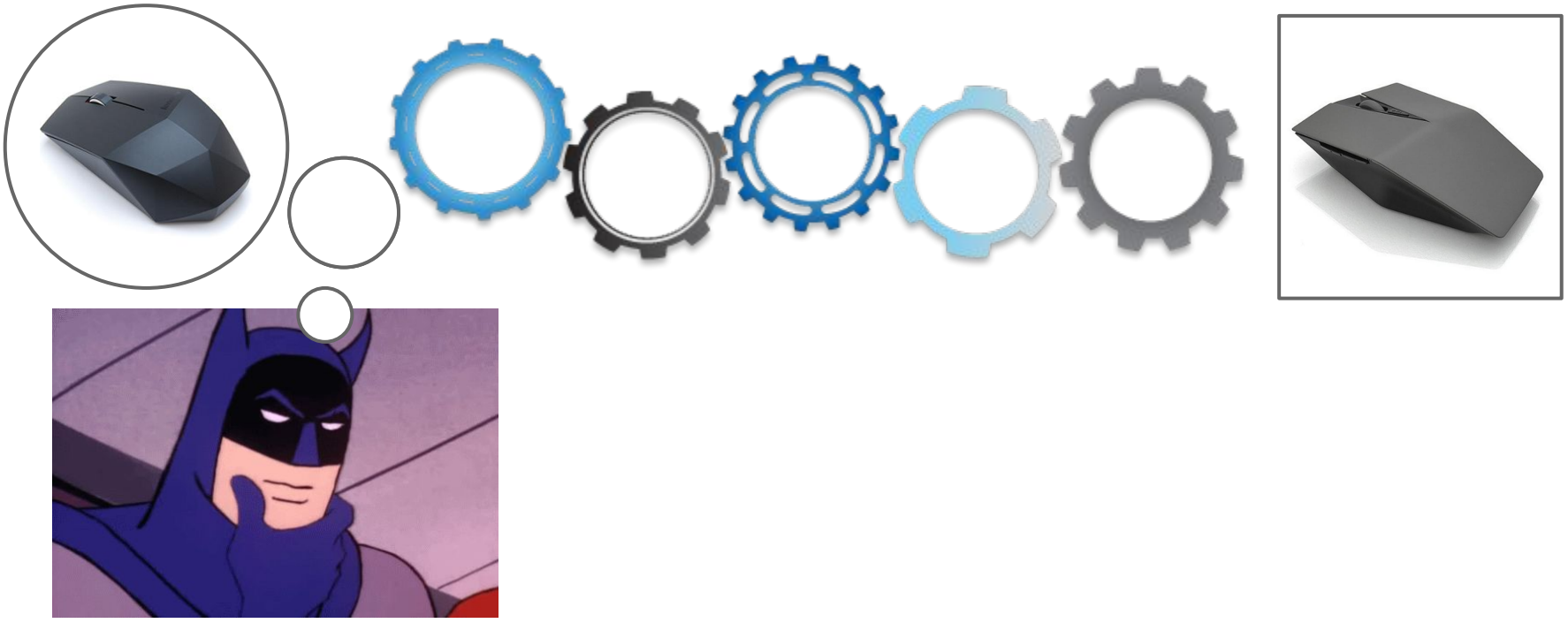


Verificación y Validación de Software

¿Qué es Verificación y Validación?

Verificación y Validación de Software

Verificación y Validación de Software



Verificación y Validación de Software

Verificación: ¿Construimos el mouse correctamente?

Validación: ¿Hicimos el mouse que el cliente quería?

Verificación y Validación de Software

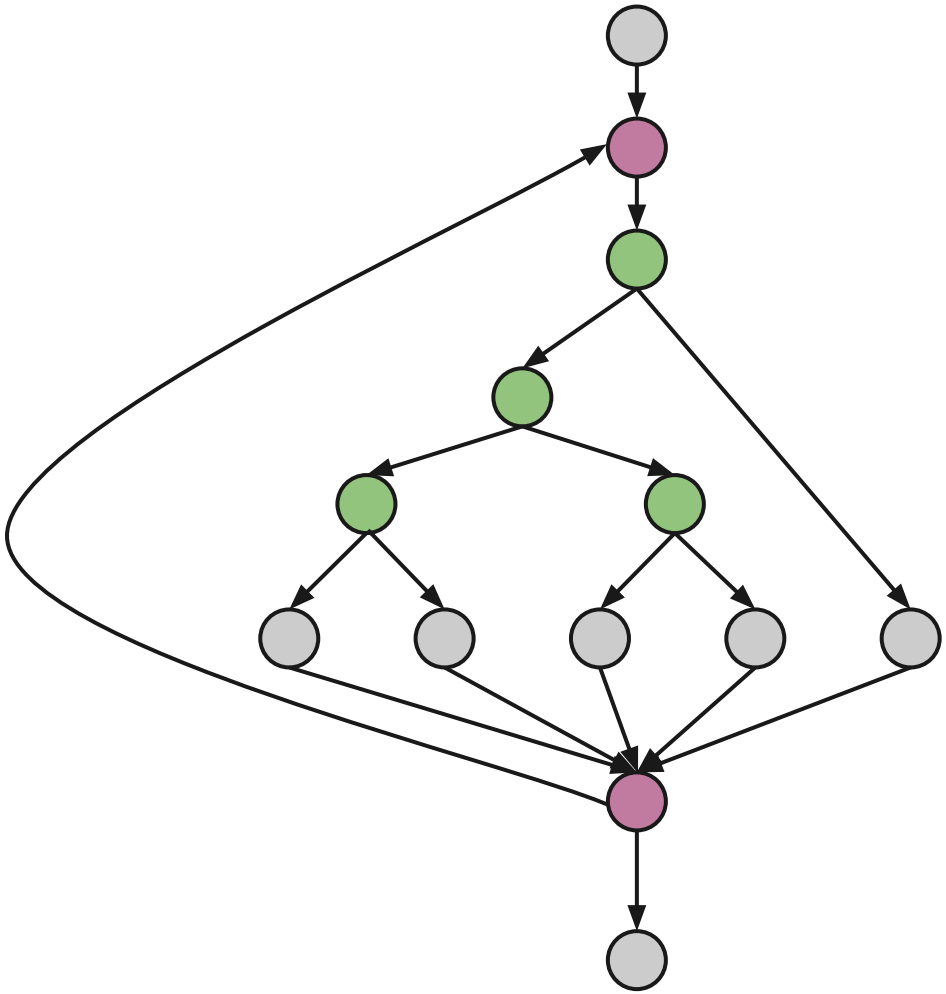
Verificación: ¿Construimos el mouse correctamente?

Podemos abusar de la terminología y llamar a todo esto **testing**

Validación: ¿Construimos el mouse que el cliente quería?

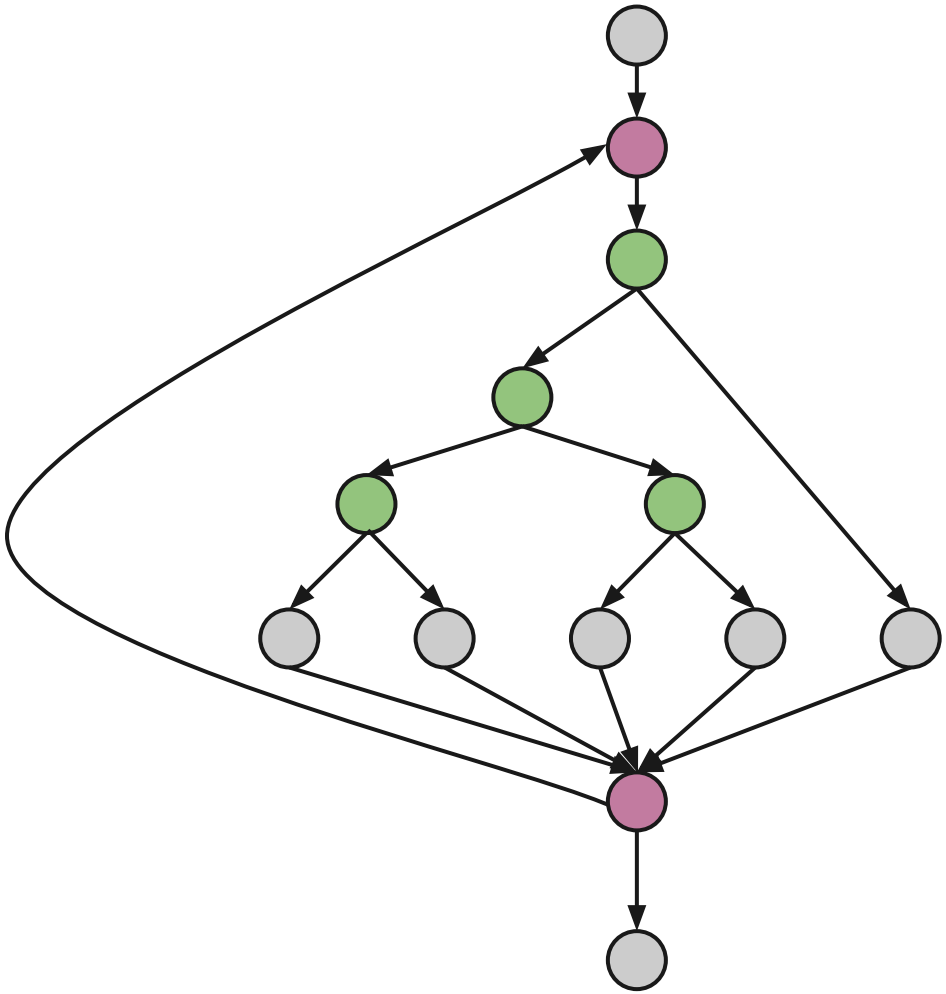
Testing de Software

Testing de Software



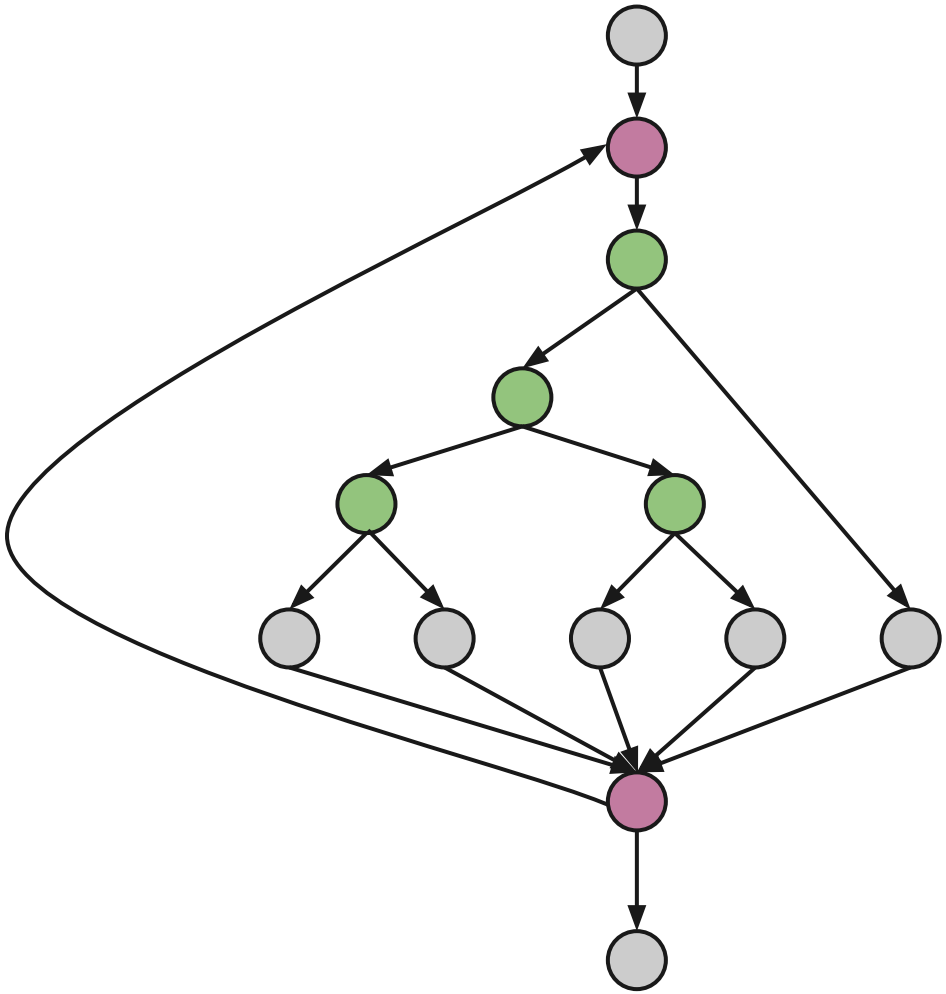
Un ejemplo

Testing de Software



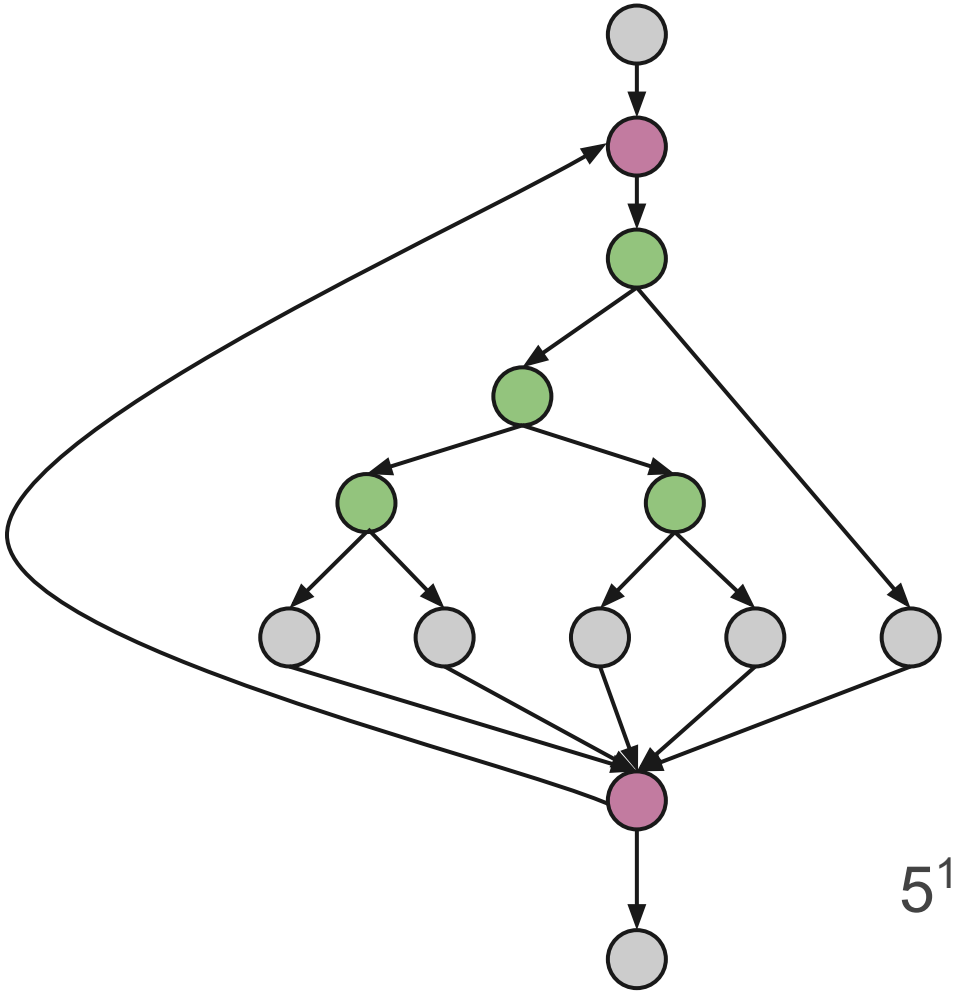
Probar este programa en forma exhaustiva significa probar cada posibilidad de camino.

Testing de Software



Si consideramos 1 iteración del bucle, tenemos 5 posibles caminos.

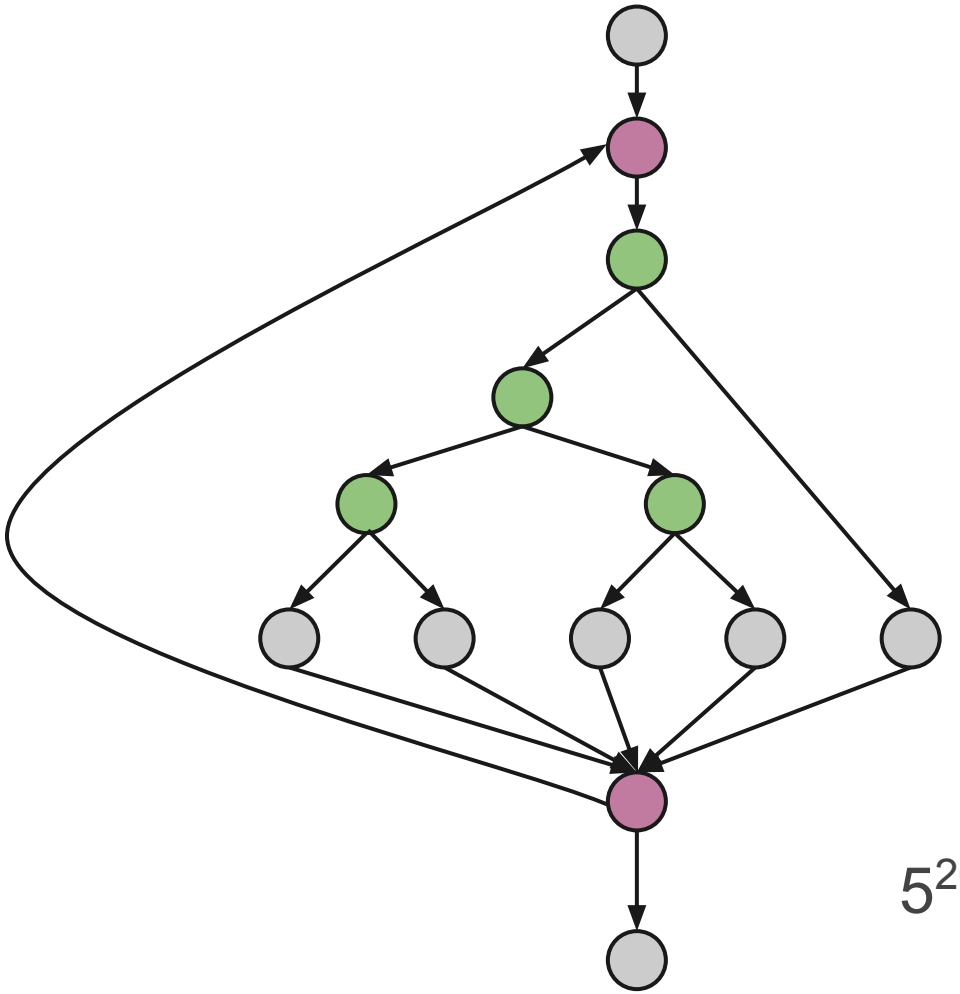
Testing de Software



Si consideramos 2 iteración del bucle, tenemos 25 posibles caminos.

$5^1 * 5^1 = 5^2 = 25$

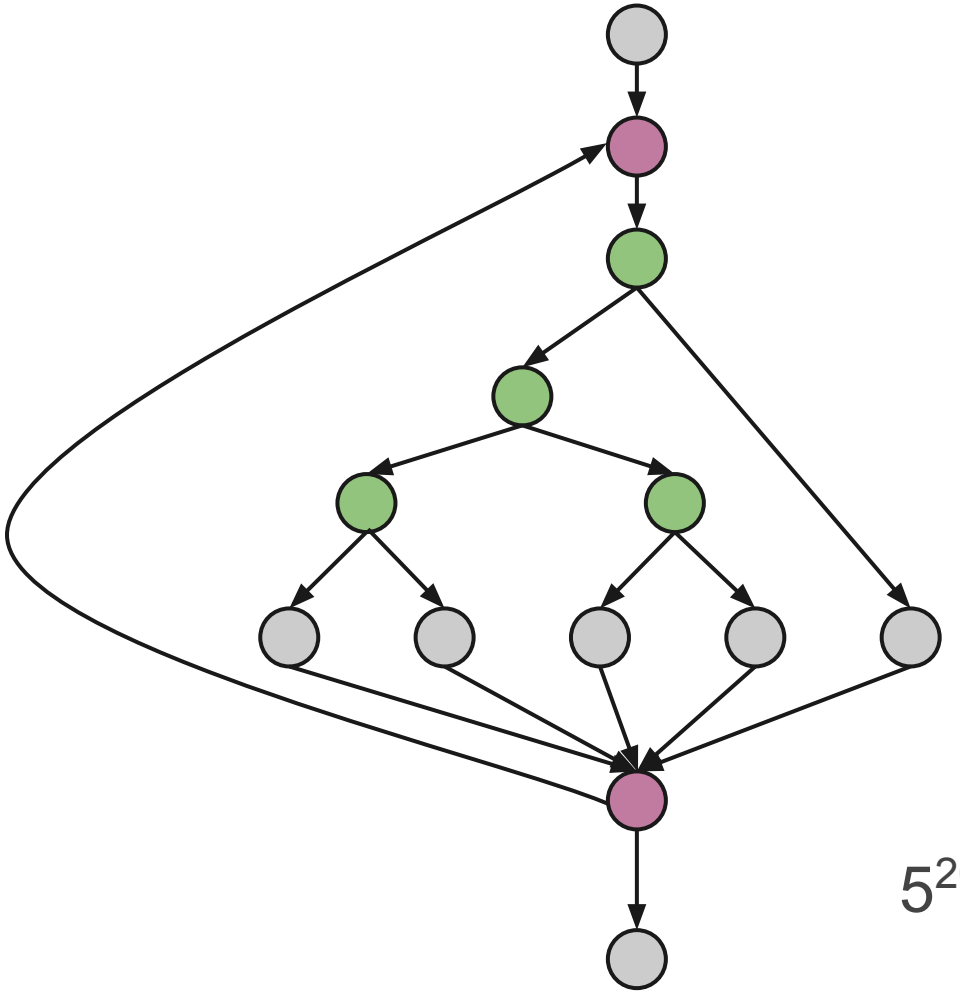
Testing de Software



Si consideramos 20 iteraciones del bucle, tenemos estos posibles caminos.

$5^{20}*5^{19}*...*5^{2}*5^{1}$

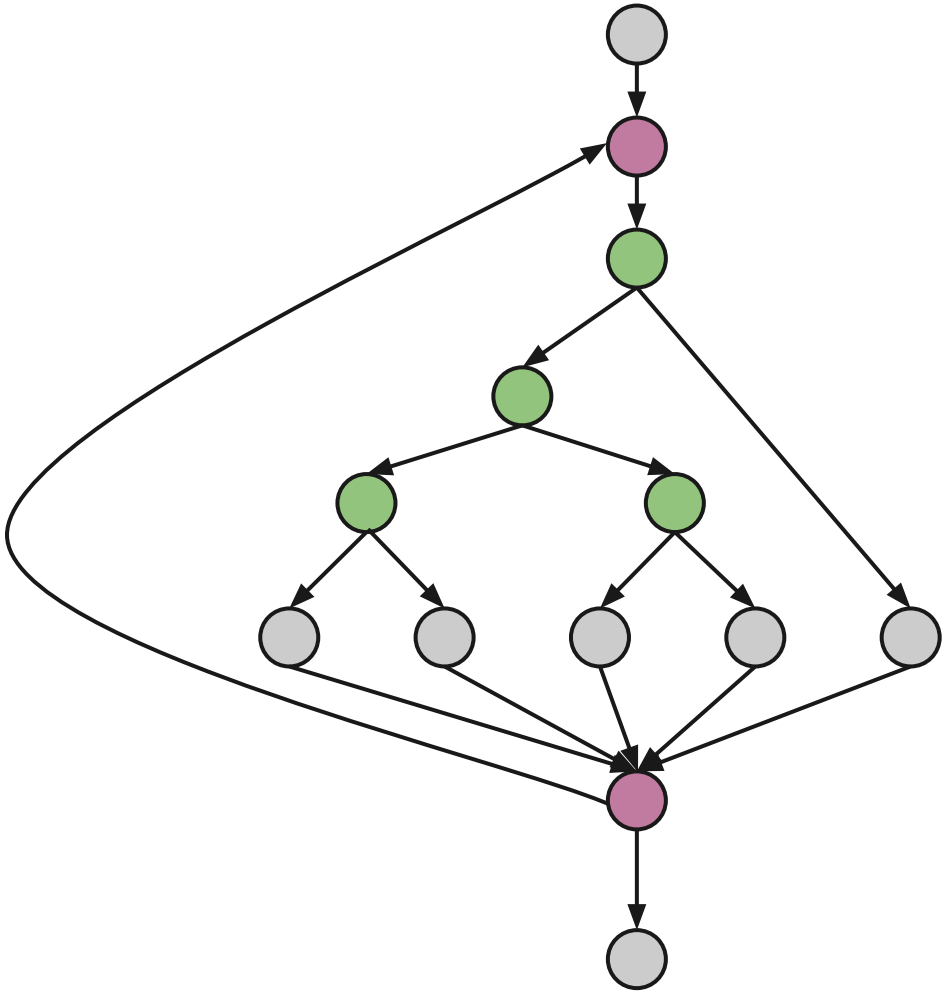
Testing de Software



Si consideramos 20 iteraciones del bucle, tenemos estos posibles caminos.

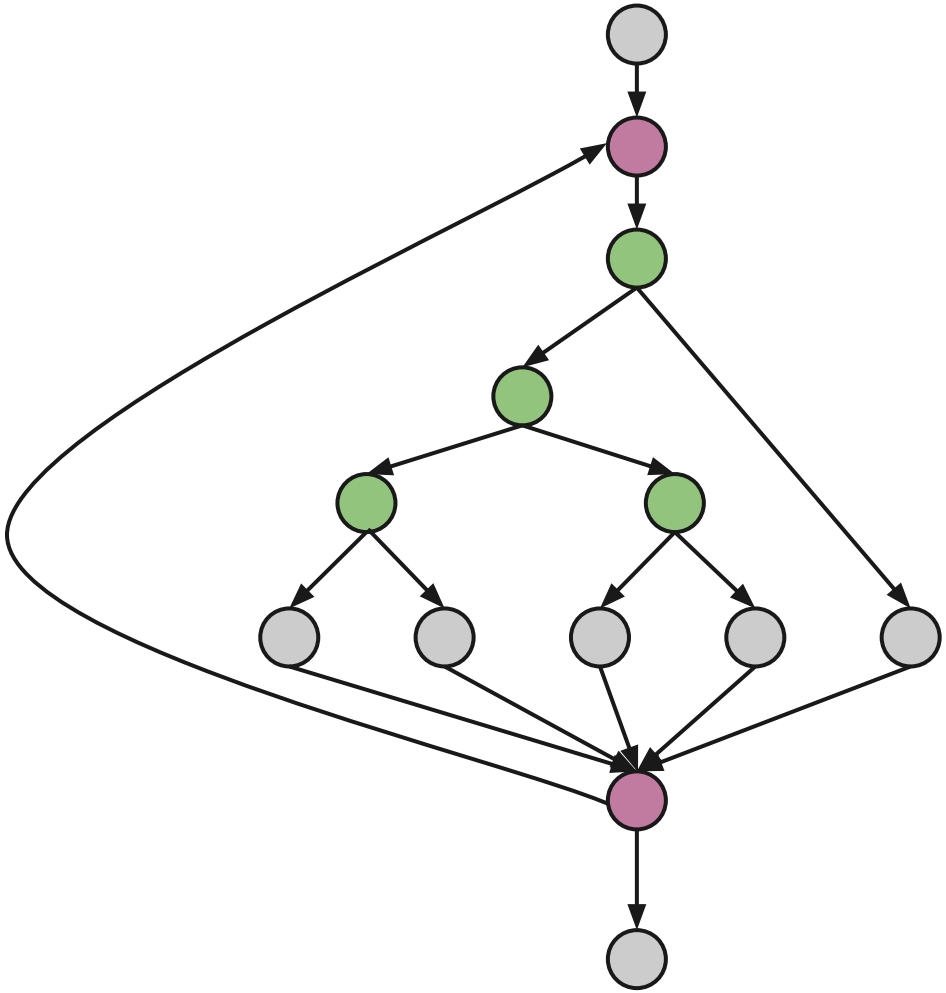
5^{20} = 9.5367432e+13

Testing de Software



Si los test se hicieran en forma manual, a 5 minutos por test. Se tardaría mil millones de años.

Testing de Software



Si los test se hicieran en 5 microsegundos por test. Se tardarían 19 años.

Testing de Software

Si el testing exhaustivo no es posible…
¿qué hacemos?

Testing de Software

Hacemos un testing inteligente

Testing de Software

"Implemente una función f que calcule el factorial de un número. El factorial de 1 es 1, y el factorial de n (n > 1) es n*f(n-1)"

Testing de Software

"Implemente una función f que calcule el factorial de un número. **El factorial de 1 es 1**, y el factorial de n (n > 1) es n*f(n-1)"

Hay que probar con: 1

Testing de Software

"Implemente una función f que calcule el factorial de un número. El factorial de 1 es 1, y el **factorial de n (n > 1) es n*f(n-1)**"

Hay que probar con: 1, 8

Testing de Software

"Implemente una función f que calcule

¿Y el cero?

f es 1, y el factorial de n (n > 1) es n*f(n-1)"

Hay que probar con: 1, 8, 0

Testing de Software

"Implemente una función f que calcule

¿Y el entero más grande posible?

f es 1, y el factorial de n (n > 1) es n*f(n-1)"

Hay que probar con: 1, 8, 0, MAXINT

Testing de Software

"Implemente una función f que calcule

¿Y un número negativo?

1 es 1, y el factorial de n (n > 1) es n*f(n-1)"

Hay que probar con: 1, 8, 0, MAXINT, -5

Testing de Software

"Implemente una función f que calcule

Hay que probar con: 1, 8, 0, MAXINT, -5

1 es 1, y el factorial de n (n > 1) es n*f(n-1)"

Testing de Software

Esto no es un testing inteligente

```
1  public class Factorial {
2      public static void main(String[] args) {
3          int numero = factorial(8);
4          System.out.println("El factorial de 8 es: " + numero);
5      }
6      public static int factorial(int n) {
7          if(n == 1) {
8              return 1;
9          } else {
10             return n * factorial(n-1);
11         }
12     }
13 }
```

Testing de Software

Siempre hacemos lo mismo, comparar lo observado con lo esperado. Esta es la esencia de Verificación y Validación de Software

Testing de Software

Si yo no sé qué esperar, entonces es poco lo que puedo testear.

Para saber si algo anda mal, necesito saber cuándo anda bien.

Testing de Software

Si yo no sé qué esperar, entonces es poco lo que puedo testear

La especificación del problema es crucial para el testing.

Para saber si algo anda mal, necesito saber cuándo anda bien.

Principios del Testing de Software

Testing shows the presence of defects, not their absence

Exhaustive testing is not possible

Testing activities should start as early as possible

Defects tend to cluster together

The pesticide paradox

Test is context dependent

The fallacy of assuming that no failures means a useful system

Tipos de Testing de Software

Existen diferentes tipos/categorías de testing

Tipos de Testing de Software

Estático Vs. Dinámico

Tipos de Testing de Software

Estático Vs. Dinámico

En el testing estático el objeto de test no se ejecuta, sólo se observa.

Tipos de Testing de Software

Estático Vs. Dinámico

Tipos de Testing de Software

Estático Vs. Dinámico

En el testing dinámico, el objeto de test se ejecuta. Se compara la salida del sistema observada con la esperada.

Testing Estático

Técnica: Examinación Grupal

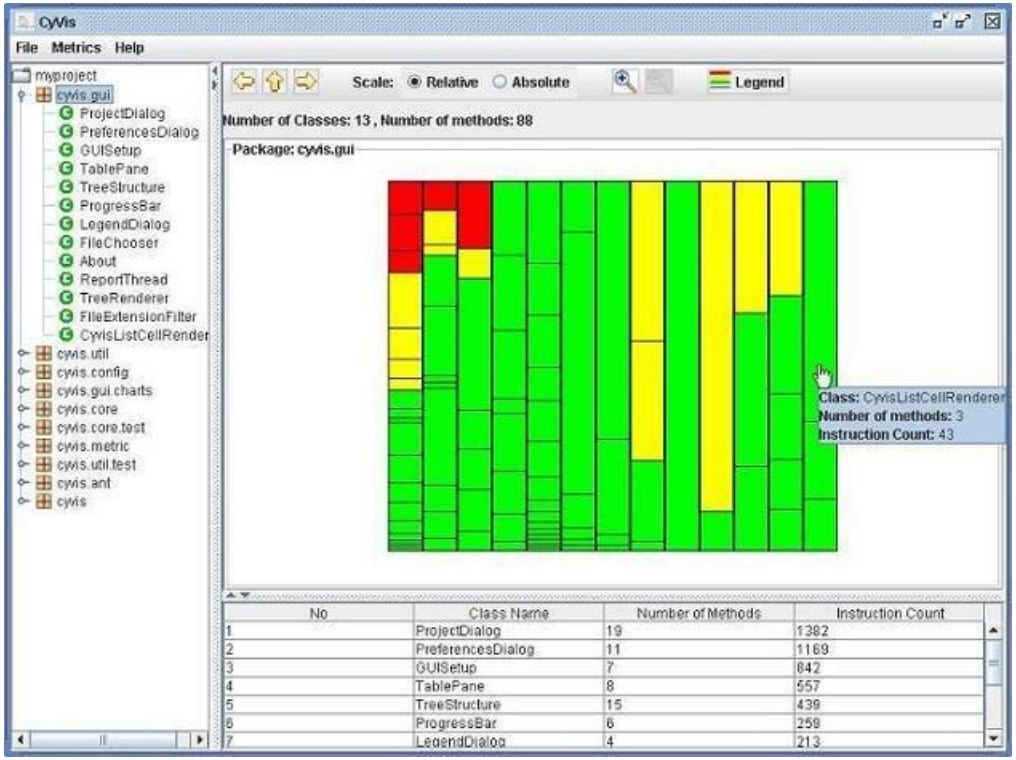
Este testing estático se basa en la capacidad humana para comprender, analizar y detectar errores en situaciones complejas.

Testing Estático

Técnica: Análisis Estático / Análisis de Complejidad

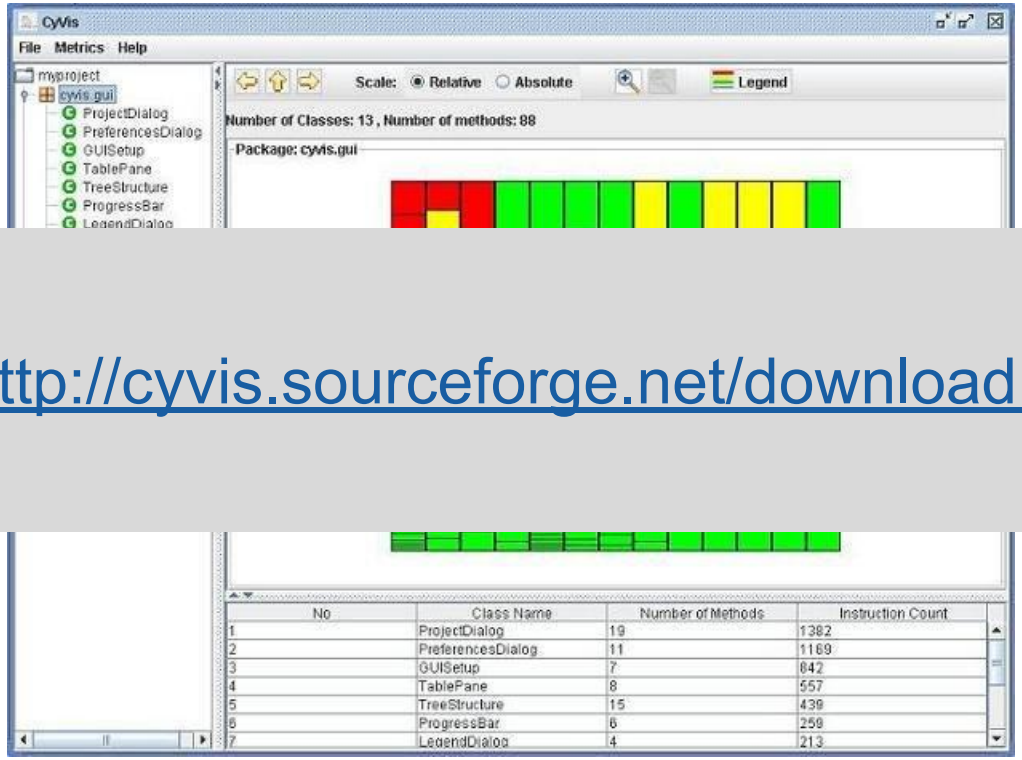
Testing Estático

Técnica: Análisis Estático / Análisis de Complejidad



Testing Estático

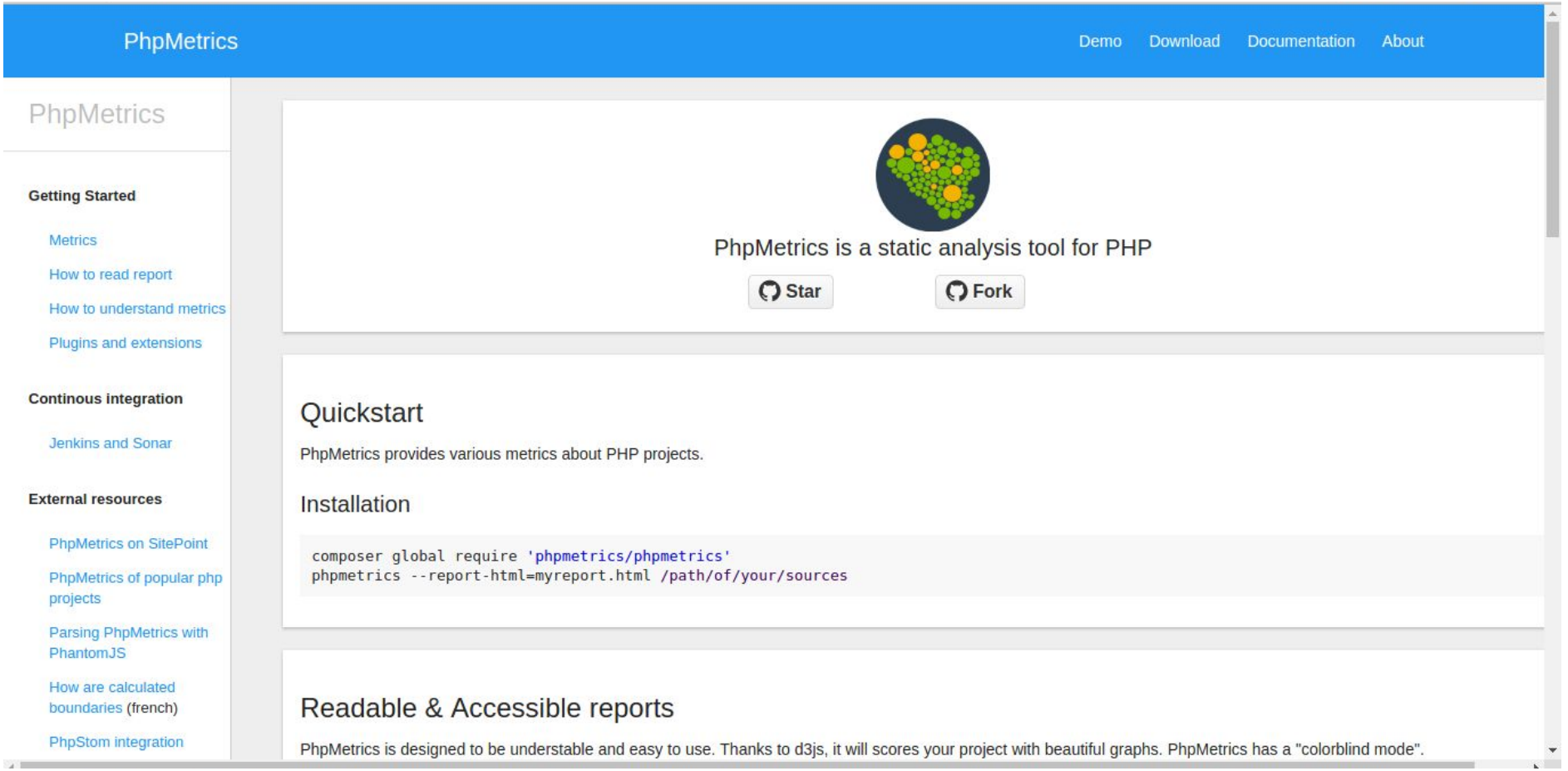
Técnica: Análisis Estático / Análisis de Complejidad



http://cyvis.sourceforge.net/download.html

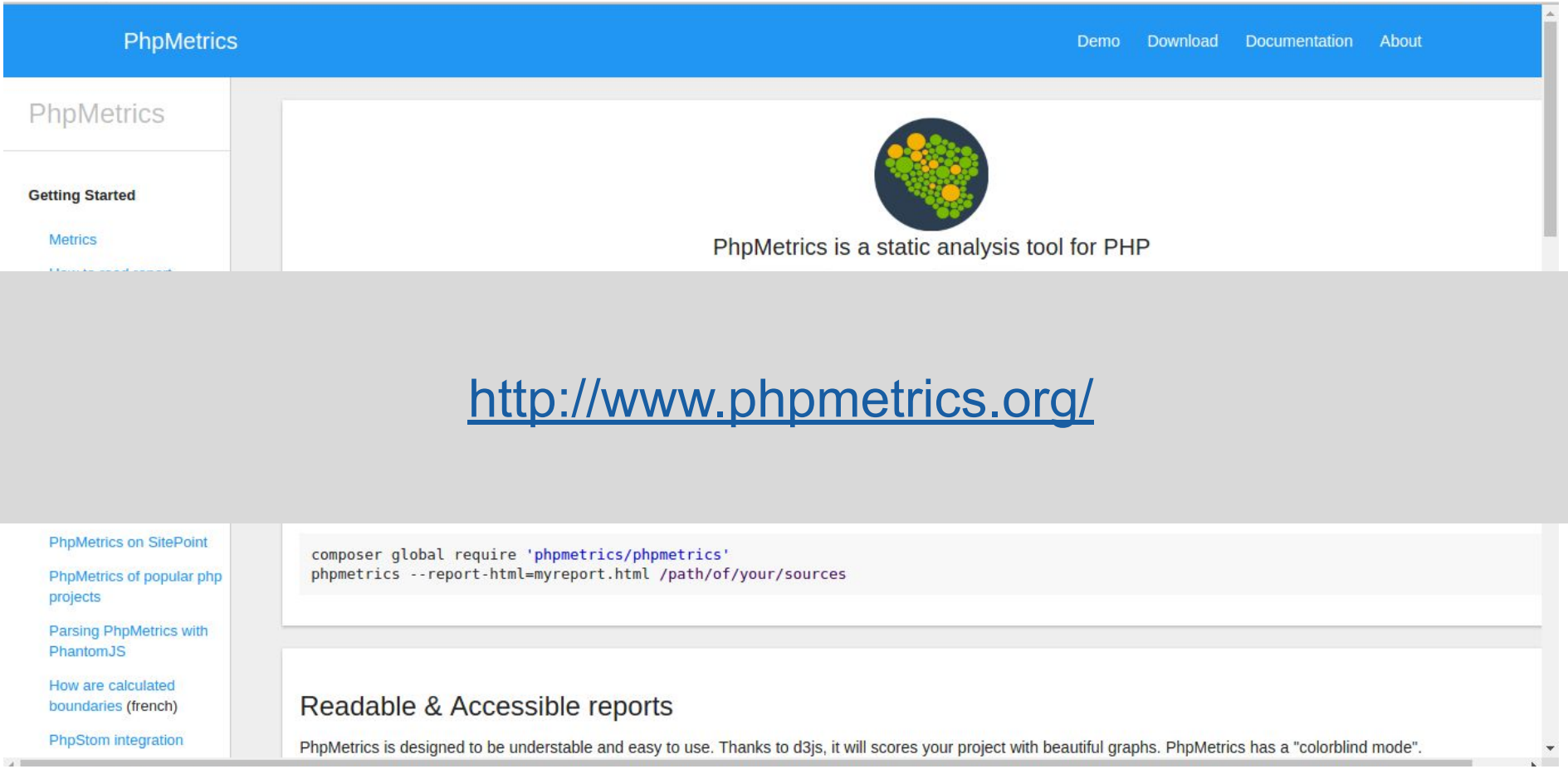
Testing Estático

Técnica: Análisis Estático / Análisis de Complejidad



Testing Estático

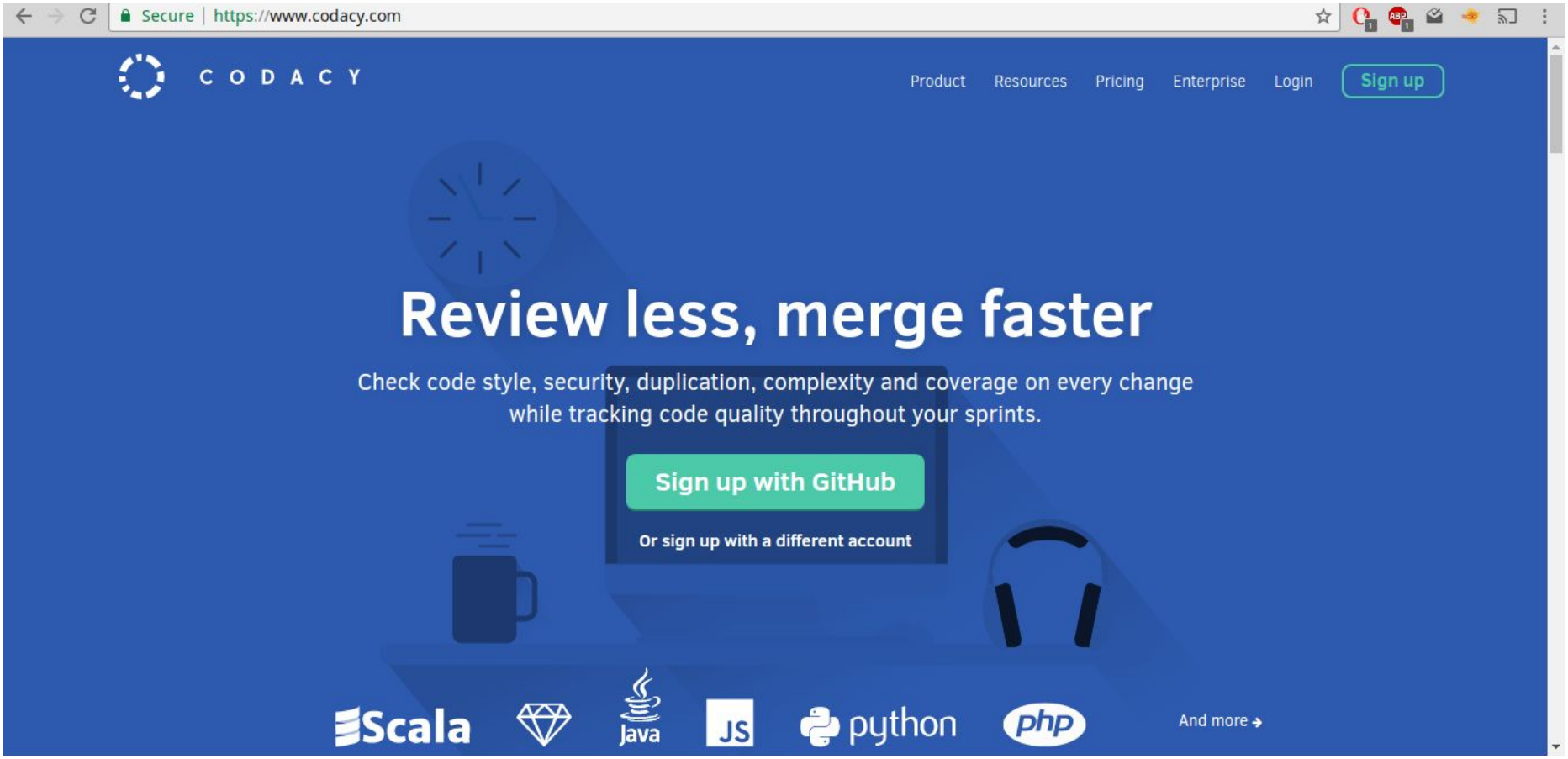
Técnica: Análisis Estático / Análisis de Complejidad



http://www.phpmetrics.org/

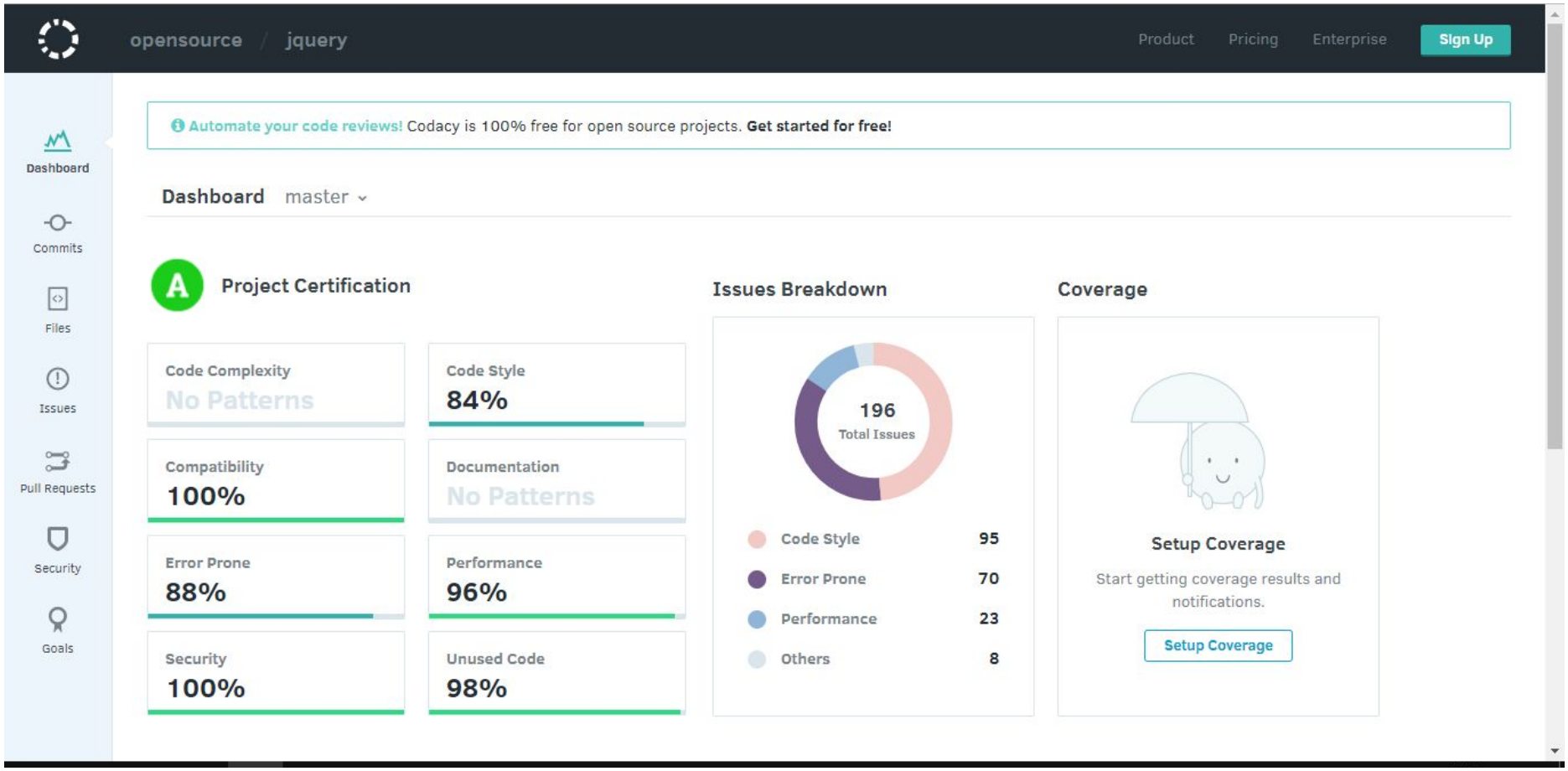
Testing Estático

Técnica: Análisis Estático



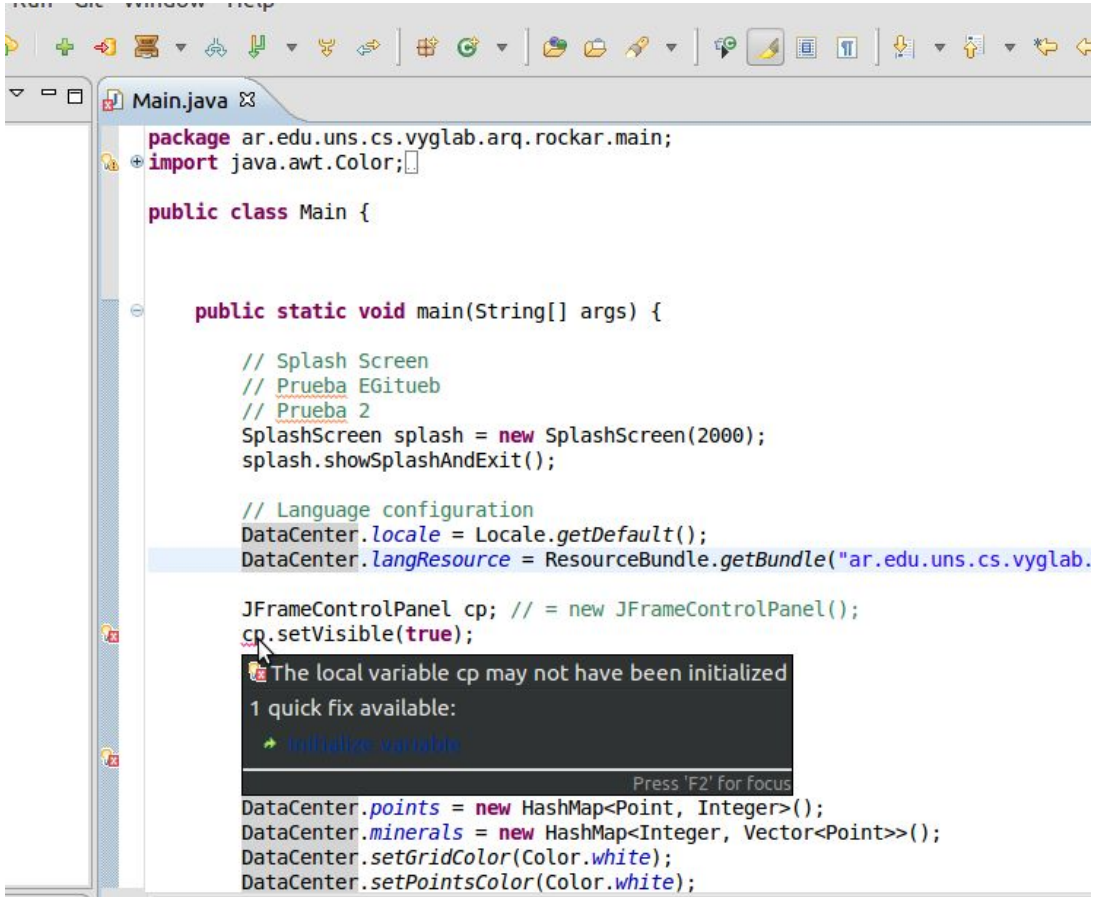
Testing Estático

Técnica: Análisis Estático



Testing Estático

Técnica: Análisis de Flujo de Datos



Testing Estático

Técnica: Análisis de Flujo de Datos

El Set-Use Pairs es un sistema de notación para el análisis de flujo de datos. El ciclo de vida de las variables se divide en tres etapas:

d: Esta etapa incluye la creación, definición o inicialización de una variable.

u: Esta etapa incluye el uso de la variable. El uso puede ser de cálculo o en una decisión.

k: Esta etapa incluye la destrucción, deallocate o salida del área de alcance.

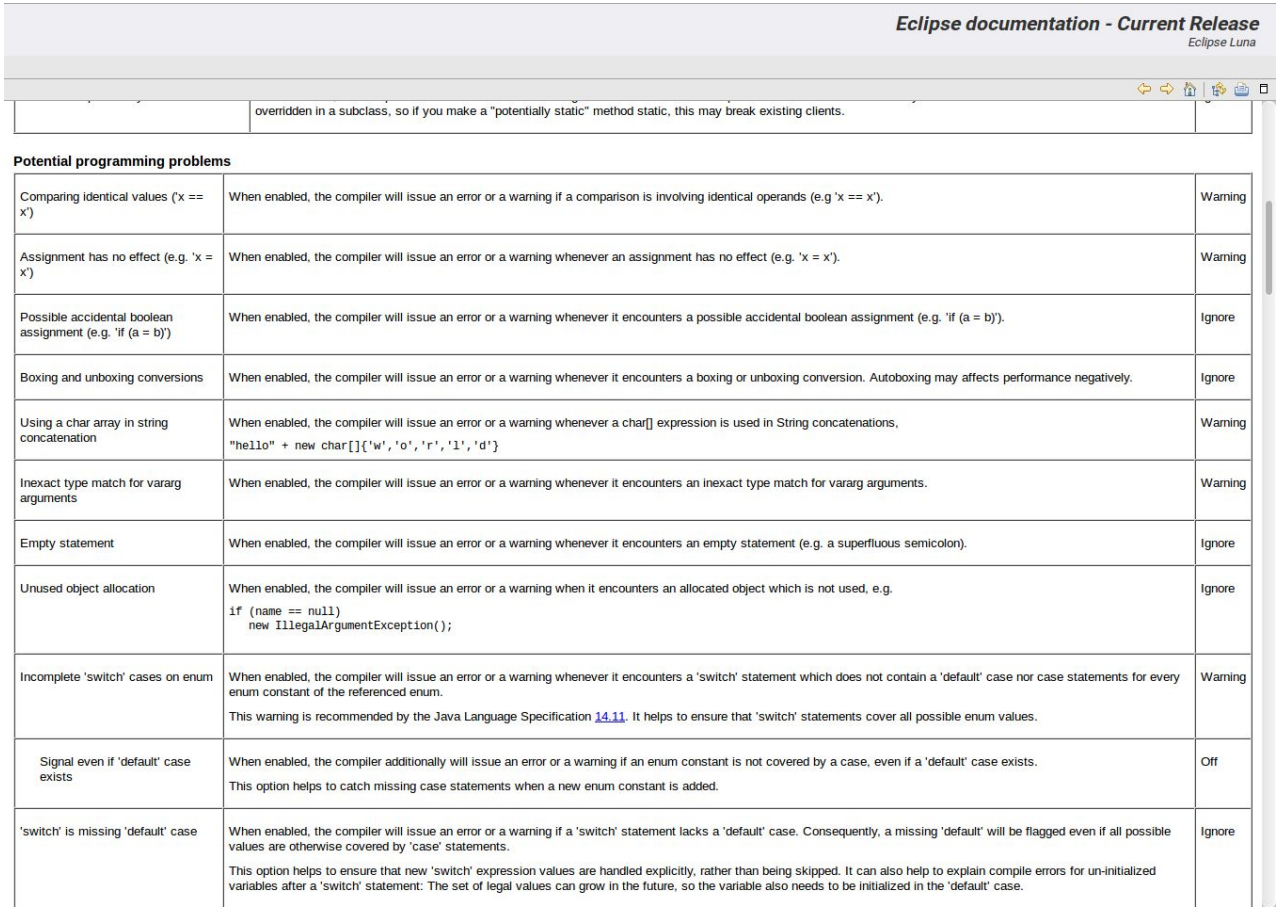
Testing Estático

Técnica: Análisis de Flujo de Datos

~d	~u	~k	dd	d~
du	ud	ku	uu	u~
dk	uk	kd	kk	k~

Testing Estático

Técnica: Análisis de Flujo de Datos



Eclipse documentation - Current Release
Eclipse Luna

overridden in a subclass, so if you make a "potentially static" method static, this may break existing clients.

Potential programming problems

Comparing identical values ('x == x')	When enabled, the compiler will issue an error or a warning if a comparison is involving identical operands (e.g 'x == x').	Warning
Assignment has no effect (e.g. 'x = x')	When enabled, the compiler will issue an error or a warning whenever an assignment has no effect (e.g. 'x = x').	Warning
Possible accidental boolean assignment (e.g. 'if (a = b)')	When enabled, the compiler will issue an error or a warning whenever it encounters a possible accidental boolean assignment (e.g. 'if (a = b)').	Ignore
Boxing and unboxing conversions	When enabled, the compiler will issue an error or a warning whenever it encounters a boxing or unboxing conversion. Autoboxing may affects performance negatively.	Ignore
Using a char array in string concatenation	When enabled, the compiler will issue an error or a warning whenever a char[] expression is used in String concatenations, `"hello" + new char[]{'w','o','r','l','d'}`	Warning
Inexact type match for vararg arguments	When enabled, the compiler will issue an error or a warning whenever it encounters an inexact type match for vararg arguments.	Warning
Empty statement	When enabled, the compiler will issue an error or a warning whenever it encounters an empty statement (e.g. a superfluous semicolon).	Ignore

Unused object allocation	When enabled, the compiler will issue an error or a warning when it encounters an allocated object which is not used, e.g.
	`if (name == null)`
	`  new IllegalArgumentException();`
		Ignore

Incomplete 'switch' cases on enum	When enabled, the compiler will issue an error or a warning whenever it encounters a 'switch' statement which does not contain a 'default' case nor case statements for every enum constant of the referenced enum. This warning is recommended by the Java Language Specification 14.11. It helps to ensure that 'switch' statements cover all possible enum values.	Warning
Signal even if 'default' case exists	When enabled, the compiler additionally will issue an error or a warning if an enum constant is not covered by a case, even if a 'default' case exists. This option helps to catch missing case statements when a new enum constant is added.	Off
'switch' is missing 'default' case	When enabled, the compiler will issue an error or a warning if a 'switch' statement lacks a 'default' case. Consequently, a missing 'default' will be flagged even if all possible values are otherwise covered by 'case' statements. This option helps to ensure that new 'switch' expression values are handled explicitly, rather than being skipped. It can also help to explain compile errors for un-initialized variables after a 'switch' statement: The set of legal values can grow in the future, so the variable also needs to be initialized in the 'default' case.	Ignore

Testing Estático

La principal ventaja del Testing Estático es su bajo costo y la facilidad de uso de las herramientas

Testing Dinámico

En el testing dinámico, el objeto de test se ejecuta. Se compara la salida del sistema observada con la esperada

Testing Dinámico

Caja Negra Vs. Caja Blanca

Testing Dinámico

Caja Negra Vs. Caja Blanca

También llamada “Basada en Especificación”. El testing se hace en base al comportamiento del sistema, sin tomar en cuenta los detalles de implementación para generar los casos de test

Testing Dinámico

Caja Negra Vs. Caja Blanca

Los casos de test se diseñan en base a cómo el sistema fue construido. El código es una documentación fundamental para este tipo de test.

Testing de Caja Negra

Partición de Equivalencias

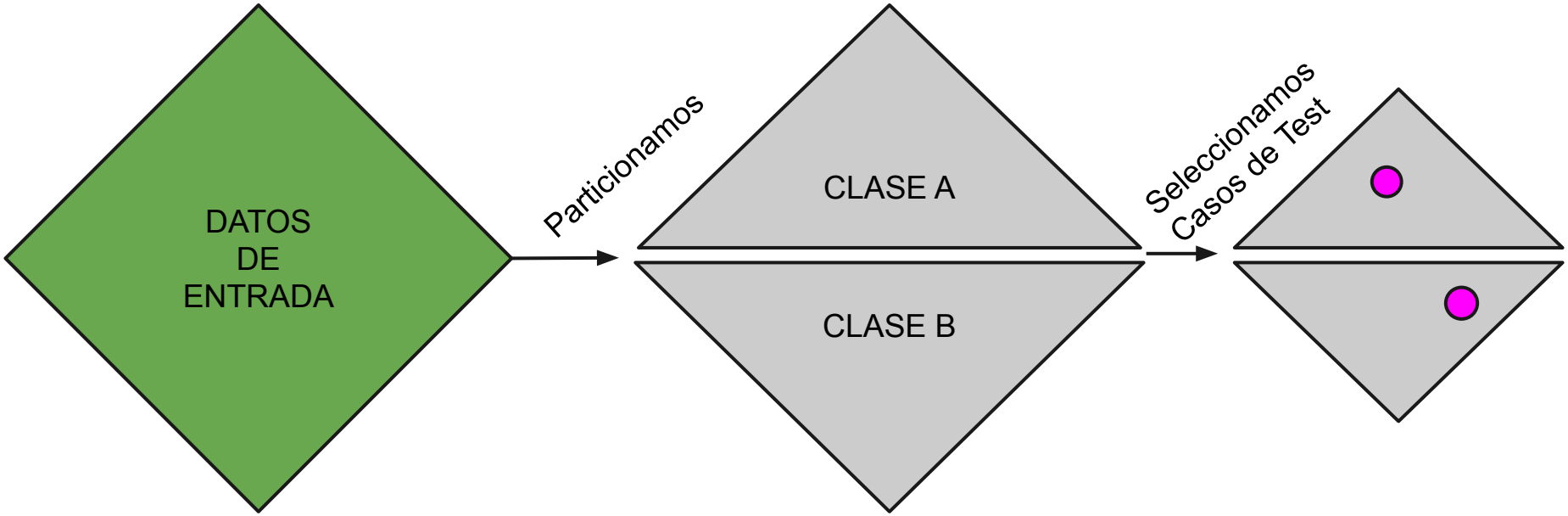
Se testean varios grupos los cuales se esperan que sean manejados de igual forma por el sistema.

El Modelo que se utiliza es el de clases de equivalencias. Clases de datos de entrada, salida, cálculos… cualquier cosa. Las clases son equivalentes porque el sistema las maneja igual.

Podemos tener clases de equivalencias válidas e inválidas. Las válidas son las que son manejadas por el sistema en forma normal. Las inválidas son rechazadas por el sistema.

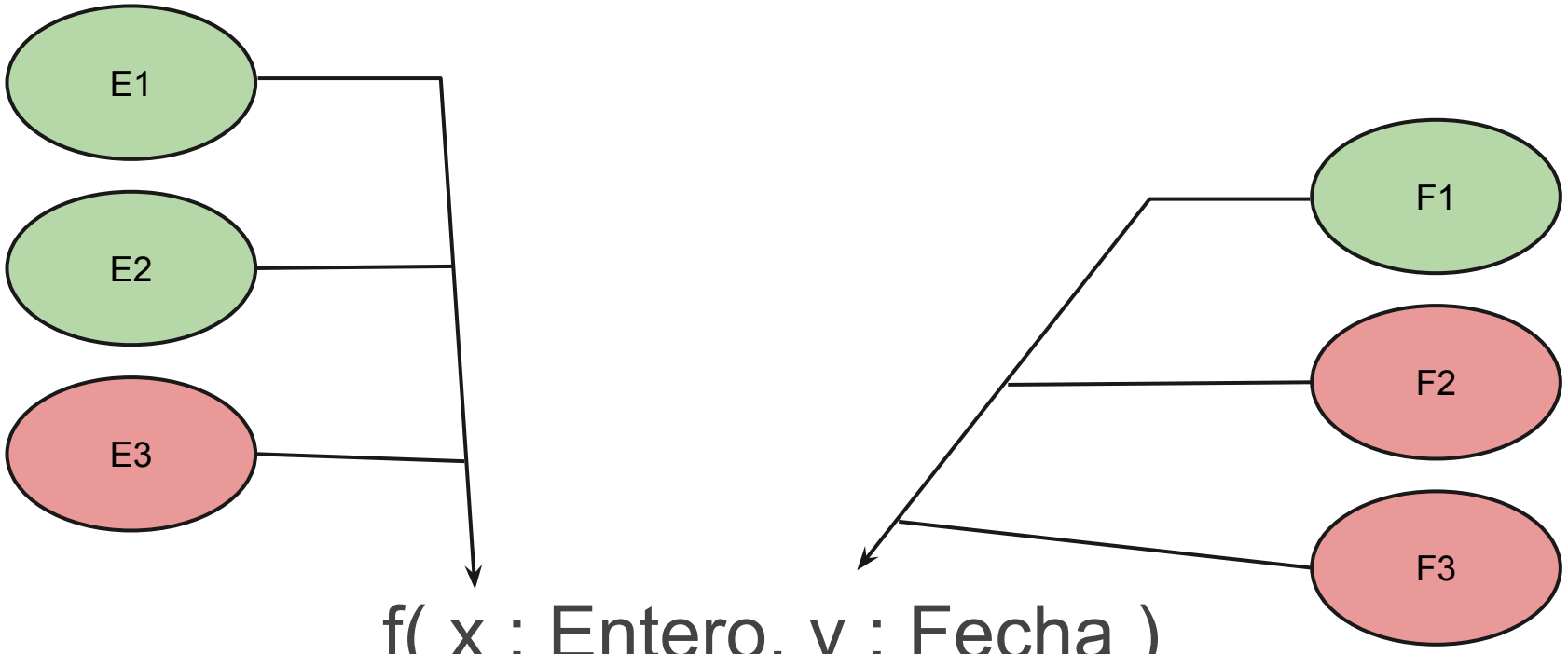
Testing de Caja Negra

Partición de Equivalencias



Testing de Caja Negra

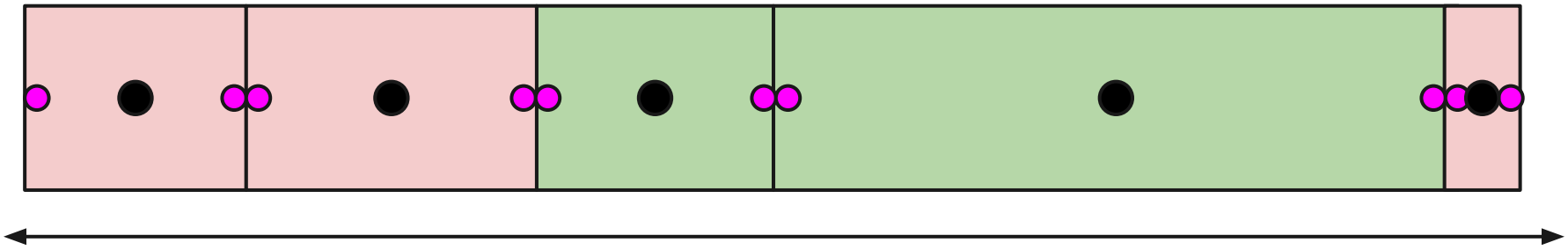
Partición de Equivalencias



Testing de Caja Negra

Partición de Equivalencias

Análisis de Valores Límites



Testing de Caja Negra

Estructuras de Datos | Trabajo Práctico N°1 | 2017

c) Implemente un programa en Java que lea dos números por teclado e imprima el cociente entre los mismos. En el caso en que el divisor sea cero, el programa debe capturar una ArithmeticException e imprimir el mensaje de error correspondiente.

Testing de Caja Negra

Estructuras de Datos Trabajo Práctico N°1 2017

c) Implemente un programa en Java que lea dos números por teclado e imprima el cociente entre los mismos. En el caso en que el divisor sea cero, el programa debe capturar una ArithmeticException e imprimir el mensaje de error correspondiente.

Dato de Entrada A

Enteros

Dato de Entrada B

Enteros

Testing de Caja Negra

Estructuras de Datos Trabajo Práctico N°1 2017

c) Implemente un programa en Java que lea dos números por teclado e imprima el cociente entre los mismos. En el caso en que el divisor sea cero, el programa debe capturar una ArithmeticException e imprimir el mensaje de error correspondiente.

Dato de Entrada A

Dato de Entrada B

Casos de Prueba: (-9,300)

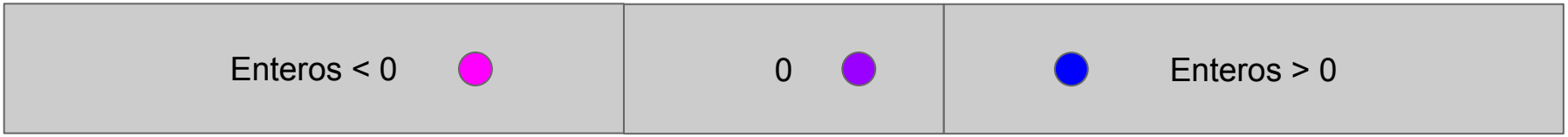
Testing de Caja Negra

Estructuras de Datos — Trabajo Práctico N°1 — 2017

c) Implemente un programa en Java que lea dos números por teclado e imprima el cociente entre los mismos. En el caso en que el divisor sea cero, el programa debe capturar una ArithmeticException e imprimir el mensaje de error correspondiente.

Dato de Entrada A

Dato de Entrada B



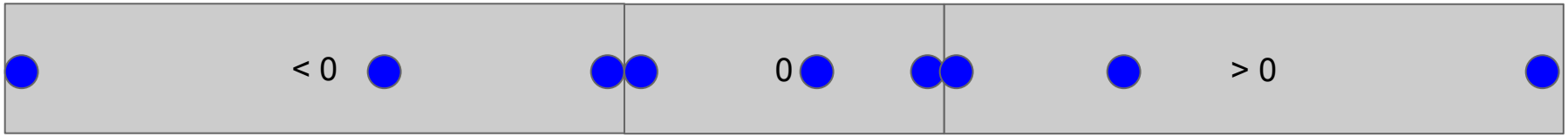
Casos de Prueba: (-9,300), (-10,0), (-5000, 50)

Testing de Caja Negra

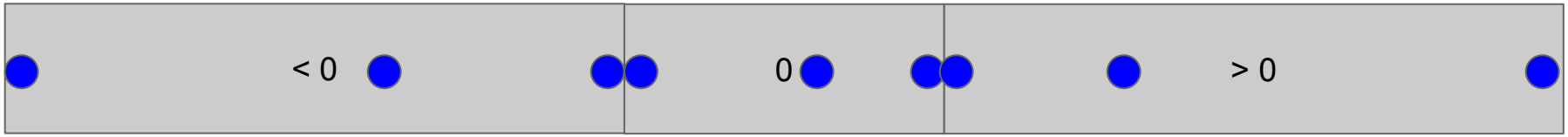
Estructuras de Datos Trabajo Práctico N°1 2017

c) Implemente un programa en Java que lea dos números por teclado e imprima el cociente entre los mismos. En el caso en que el divisor sea cero, el programa debe capturar una ArithmeticException e imprimir el mensaje de error correspondiente.

Dato de Entrada A



Dato de Entrada B



Testing Dinámico

Caja Blanca

Los casos de test se diseñan en base a cómo el sistema fue construido. El código es una documentación fundamental para este tipo de test.

Testing de Caja Blanca

Testing del Flujo de Control

El flujo de control de un programa, es la estructura que determina los “caminos” que se pueden tomar a lo largo del código. Los if, while, repeat, case, forman parte de la estructura de control.

Testing de Caja Blanca

Grafo de Flujo de Control

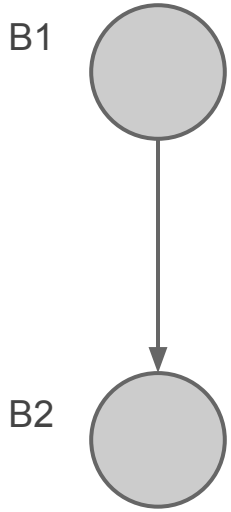
Control-Flow Graph

El Grafo de Flujo de Control es el modelo/representación que nos va a facilitar realizar/evaluar los diferentes cubrimiento.

Testing de Caja Blanca

Grafo de Flujo de Control

Control-Flow Graph



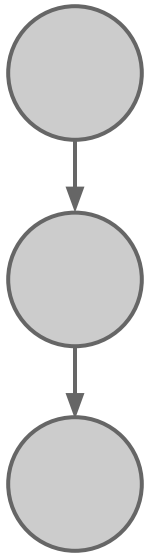
Dos sentencias de código en secuencia

Testing de Caja Blanca

Grafo de Flujo de Control

Control-Flow Graph

```
int x = 23;
x++;
f( x );
```

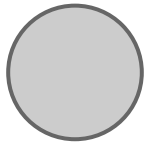


Testing de Caja Blanca

Grafo de Flujo de Control

Control-Flow Graph

```
int x = 23;
x++;
f( x );
```

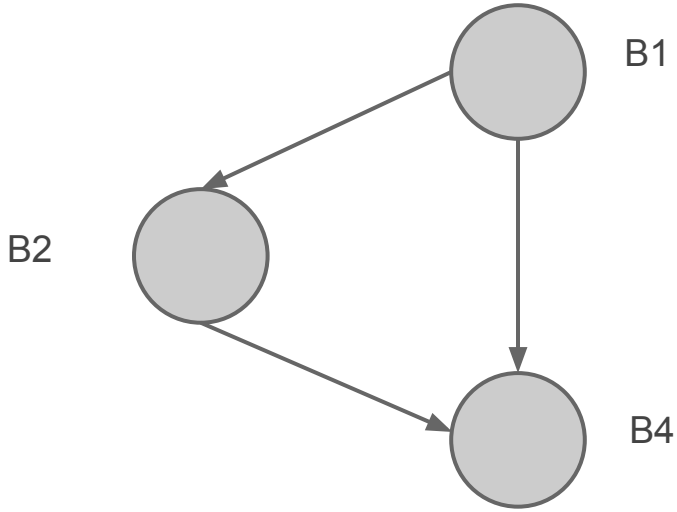


Como, en este ejemplo, no se modifica el flujo de control vamos a considerarlo un único nodo.

Testing de Caja Blanca

Grafo de Flujo de Control

Control-Flow Graph



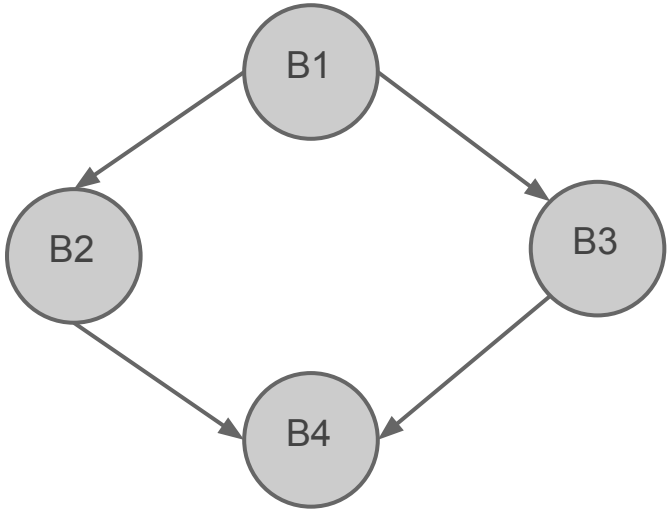
IF-THEN

```
IF B1 THEN {
      B2;
}
B4
```

Testing de Caja Blanca

Grafo de Flujo de Control

Control-Flow Graph



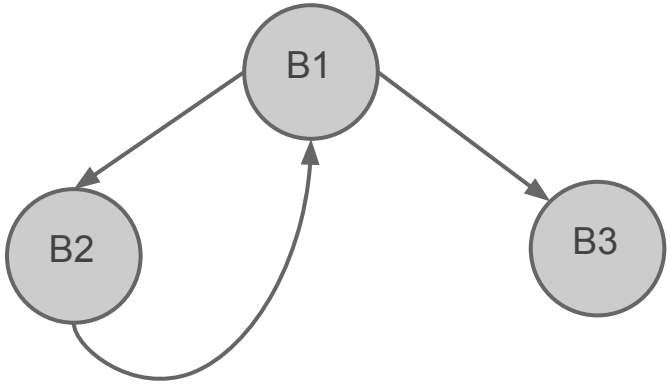

```
IF B1 THEN {
     B2; }
ELSE {
     B3; }
B4
```

IF-THEN-ELSE

Testing de Caja Blanca

Grafo de Flujo de Control

Control-Flow Graph




```
WHILE B1 {
    B2;
}
B3;
```

CICLO

Testing de Caja Blanca

Grafo de Flujo de Control

Control-Flow Graph

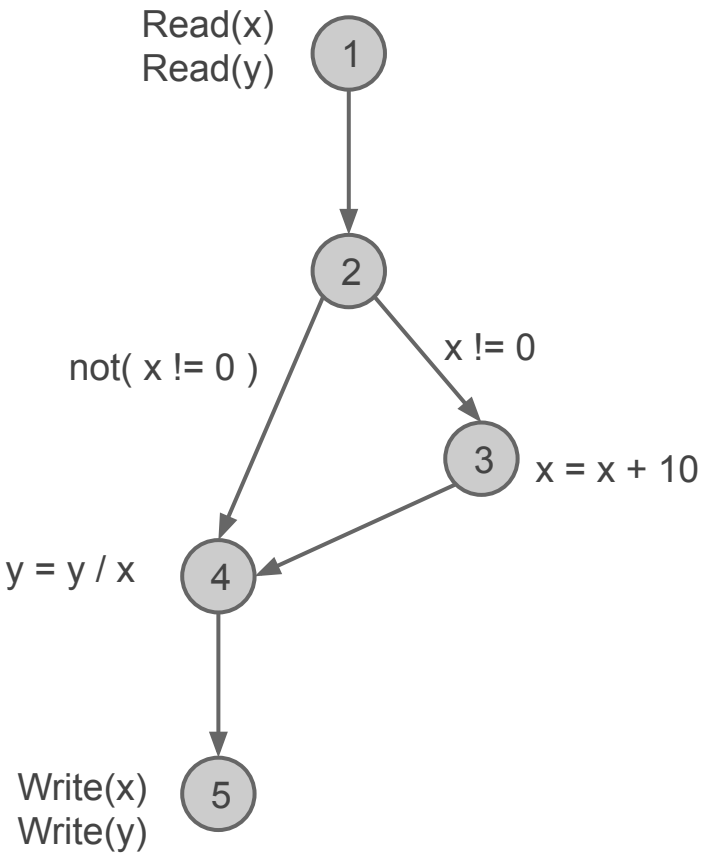
```
PROGRAM
VAR x,y:REAL
BEGIN
      Read(x);
      Read(y);
      IF (x != 0)
            THEN x = x + 10;
      y = y / 10;
      Write(x);
      Write(y);
END
```

Testing de Caja Blanca

Grafo de Flujo de Control

Control-Flow Graph

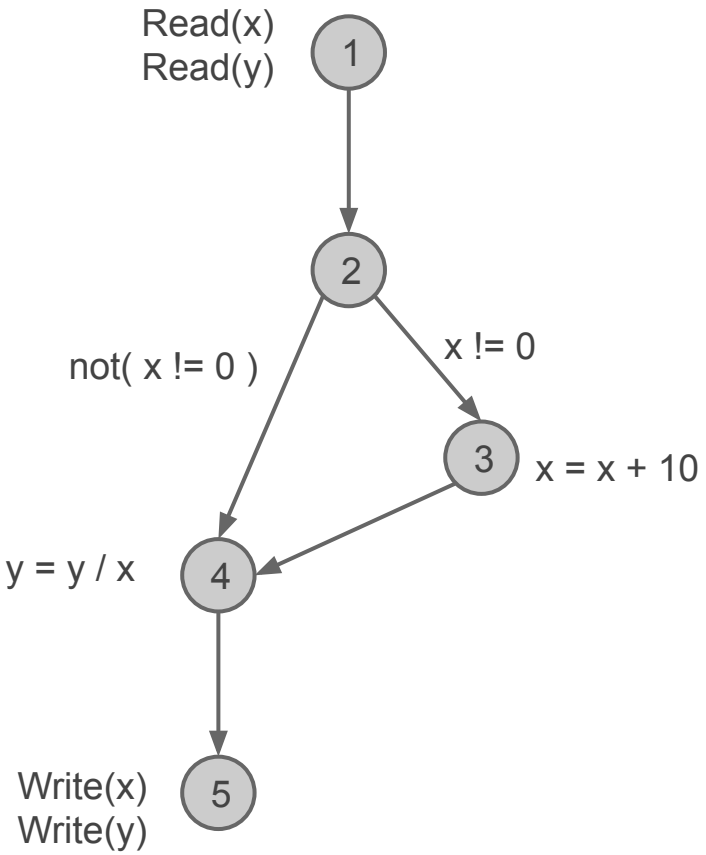
```
PROGRAM
VAR x,y:REAL
BEGIN
      Read(x);
      Read(y);
      IF (x != 0)
            THEN x = x + 10;
      y = y / 10;
      Write(x);
      Write(y);
END
```



Testing de Caja Blanca

Criterios de Cubrimiento

Criterio de Comandos



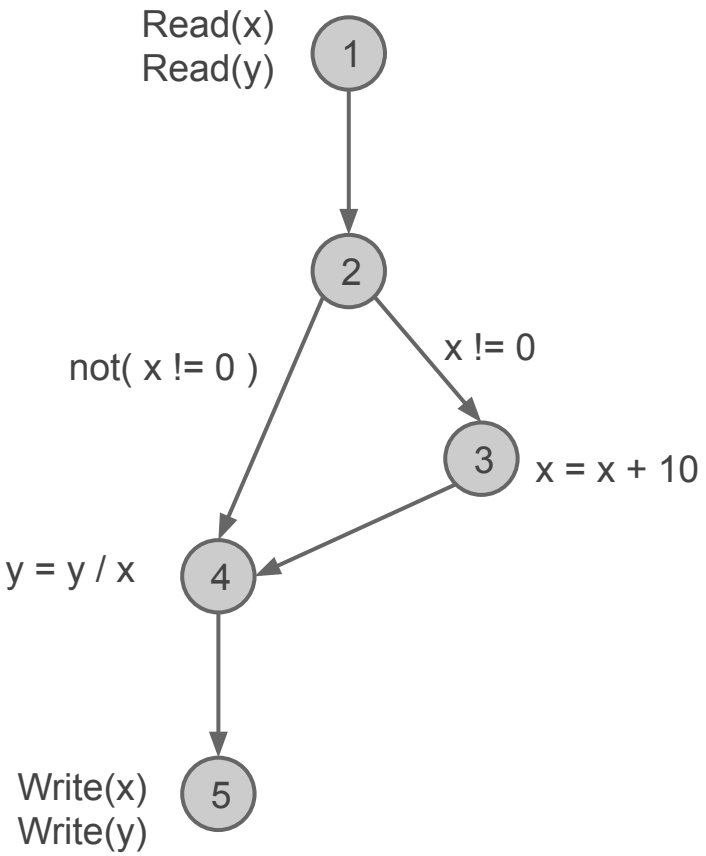
Testing de Caja Blanca

Criterios de Cubrimiento

Criterio de Comandos

Un test T satisface este criterio ssi cada comando (sentencia) ejecutable del programa es ejecutado al menos una vez.

Visito todos los nodos.



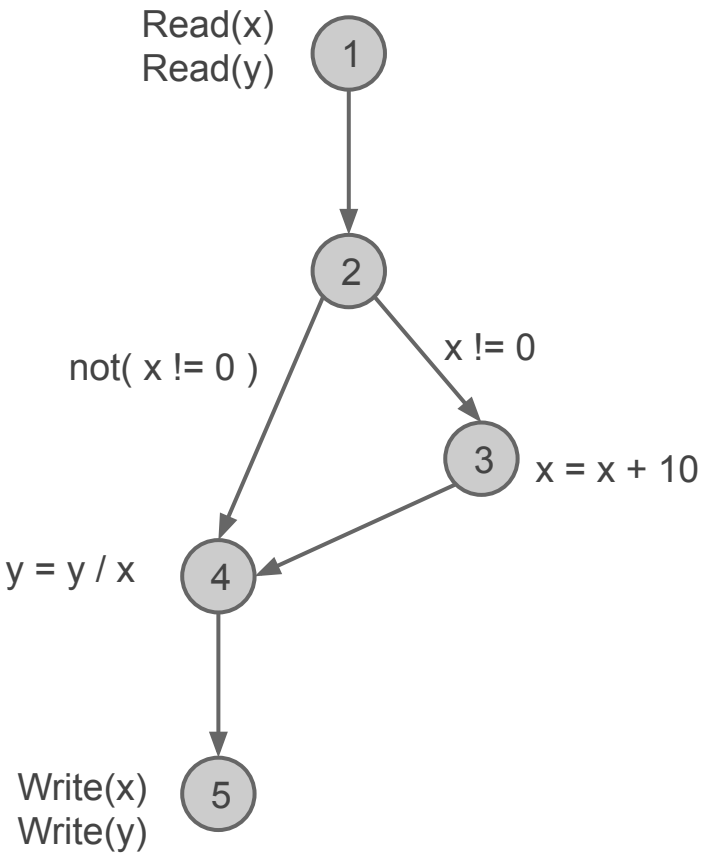
Testing de Caja Blanca

Criterios de Cubrimiento

Criterio de Comandos

Un test T satisface este criterio ssi cada comando (sentencia) ejecutable del programa es ejecutado al menos una vez.

El camino [1, 2, 3, 4, 5] visita todos los nodos del grafo. Si pudiese encontrar un par (x, y) (o más pares) tal que al usarlos como inputs se ejecutase dicho camino, entonces podríamos decir que el test T = {(x,y)} satisface el Criterio de Comandos.



Testing de Caja Blanca

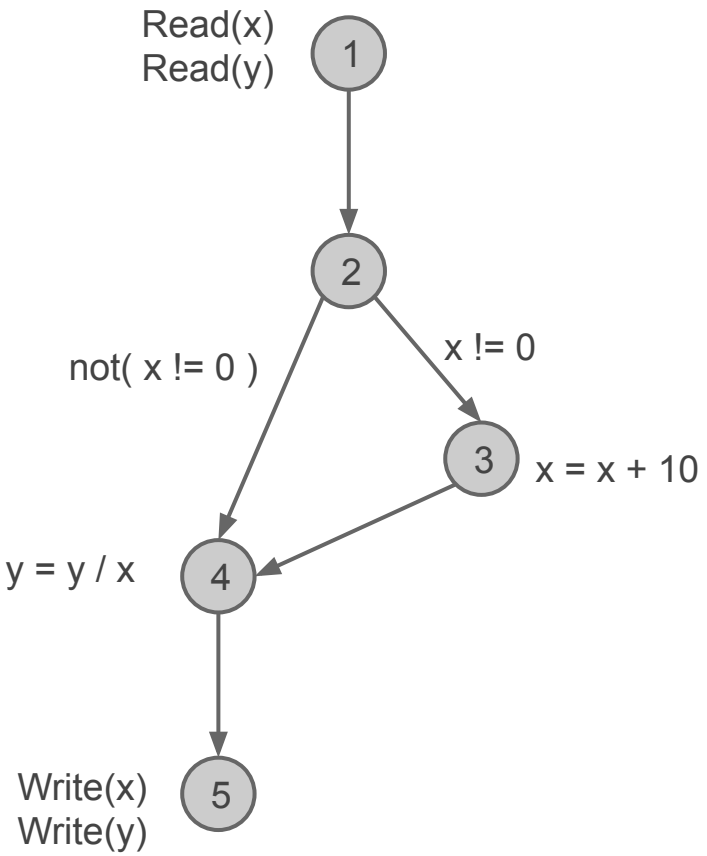
Criterios de Cubrimiento

Criterio de Comandos

Un test T satisface este criterio ssi cada comando (sentencia) ejecutable del programa es ejecutado al menos una vez.

El camino [1, 2, 3, 4, 5] visita todos los nodos del grafo. Si pudiese encontrar un par (x, y) (o más pares) tal que al usarlos como inputs se ejecutase dicho camino, entonces podríamos decir que el test T = {(x,y)} satisface el Criterio de Comandos.

T_{CC}= {(x=20, y=30)}



Testing de Caja Blanca

Criterios de Cubrimiento

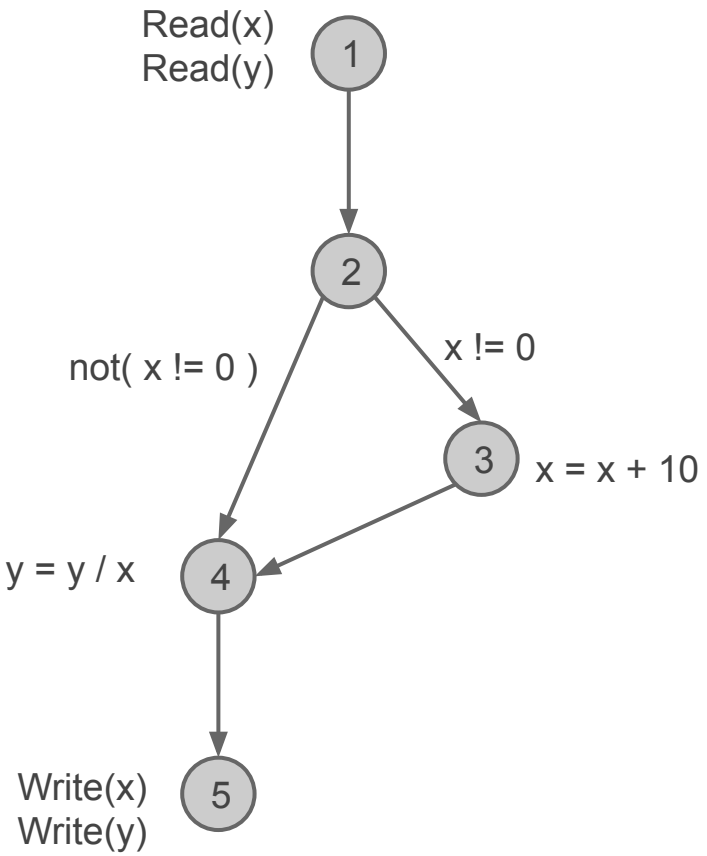
Criterio de Comandos

Un test T satisface este criterio ssi cada comando (sentencia) ejecutable del programa es ejecutado al menos una vez.

El camino [1, 2, 3, 4, 5] visita todos los nodos del grafo. Si pudiese encontrar un par (x, y) (o más pares) tal que al usarlos como inputs se ejecutase dicho camino, entonces podríamos decir que el test T = {(x,y)} satisface el Criterio de Comandos.

T_{CC}= {(x=20, y=30)}

Puedo llegar a necesitar más pares...



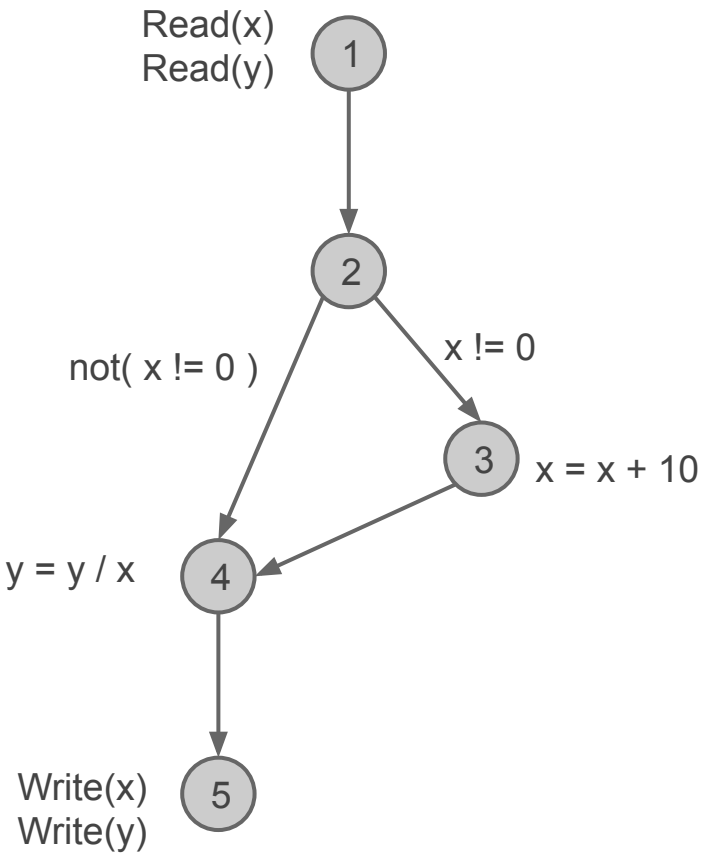
Testing de Caja Blanca

Criterios de Cubrimiento

Criterio de Comandos

Un test T satisface este criterio ssi cada comando (sentencia) ejecutable del programa es ejecutado al menos una vez.

Con T_{CC}= {(x=20, y=30)} cubro todas las líneas ejecutables del programa. ¿Es un buen test? Si el test da bien, ¿me puedo quedar tranquilo de que no hay errores?



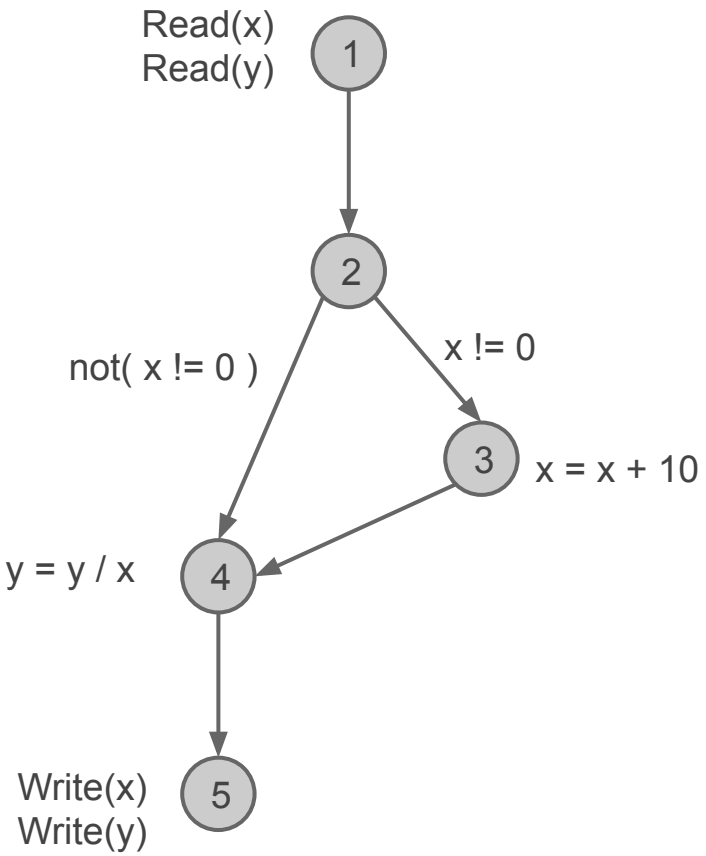
Testing de Caja Blanca

Criterios de Cubrimiento

Criterio de Decisiones

Un test T satisface este criterio ssi cada arco del grafo de flujo de control es visitado al menos una vez.

Visito todos los arcos.



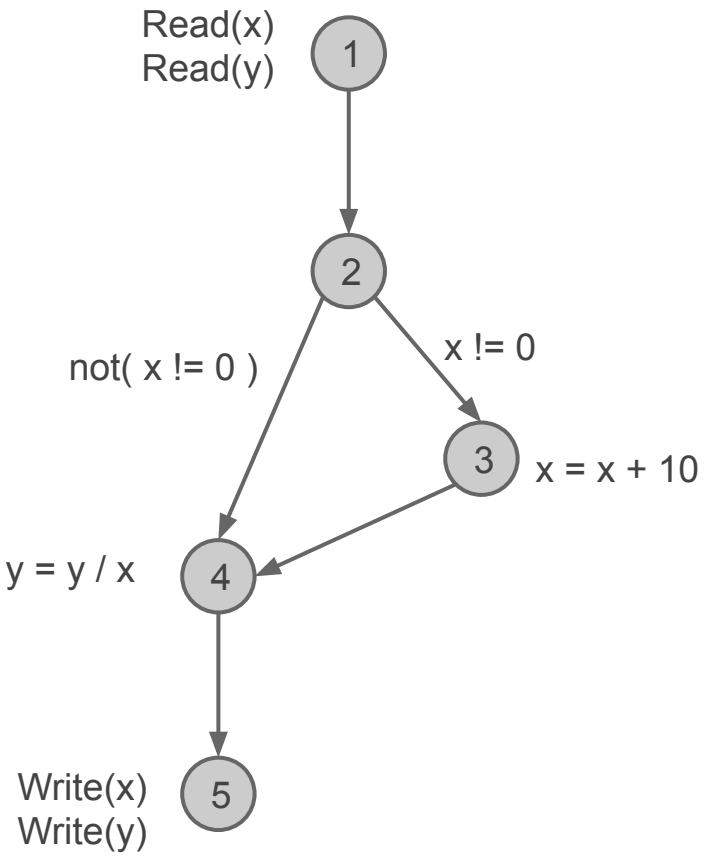
Testing de Caja Blanca

Criterios de Cubrimiento

Criterio de Decisiones

Un test T satisface este criterio ssi cada arco del grafo de flujo de control es visitado al menos una vez.

El camino [1, 2, 3, 4, 5] **no** visita todos los arcos del grafo. Por lo que T_{CC}= {(x=20, y=30)} no satisface el Criterio de Decisiones.



Testing de Caja Blanca

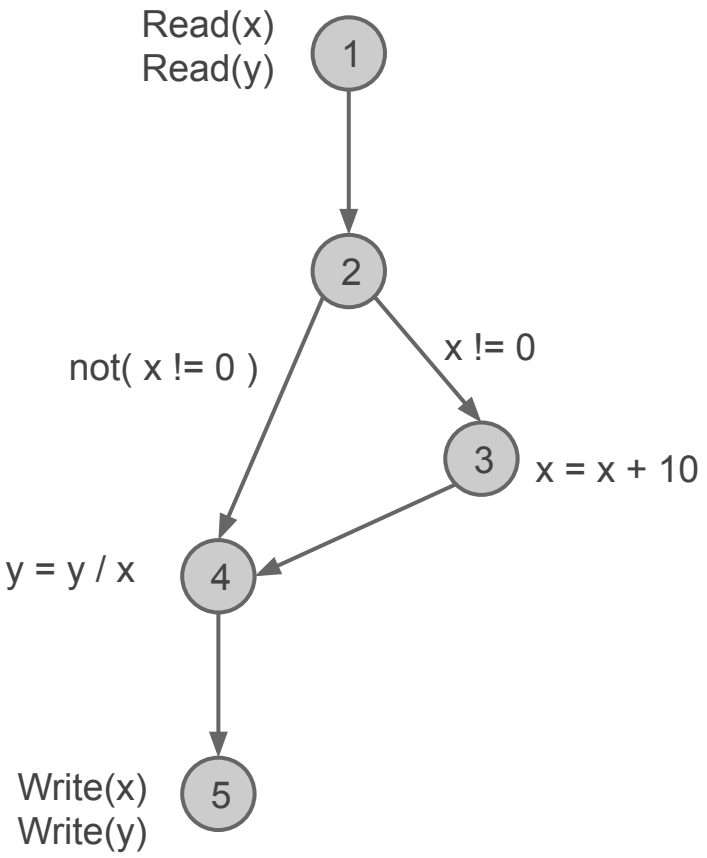
Criterios de Cubrimiento

Criterio de Decisiones

Un test T satisface este criterio ssi cada arco del grafo de flujo de control es visitado al menos una vez.

El camino [1, 2, 3, 4, 5] **no** visita todos los arcos del grafo. Por lo que T_{CC}= {(x=20, y=30)} no satisface el Criterio de Decisiones.

El conjunto de caminos ([1, 2, 3, 4, 5], [1, 2, 4, 5]) **sí** visita todos los arcos. Si puedo encontrar (x_1, y_1) y (x_2, y_2) que recorren dichos caminos entonces podría decir que T = {(x_1,y_1), (x_2,y_2)} satisface el Criterio de Decisiones.



Testing de Caja Blanca

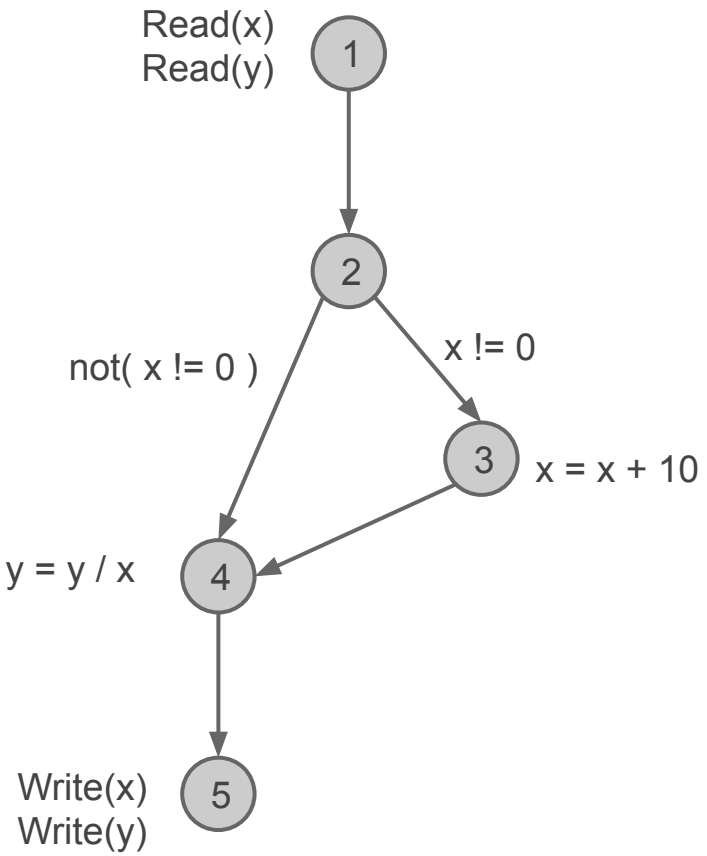
Criterios de Cubrimiento

Criterio de Decisiones

Un test T satisface este criterio ssi cada arco del grafo de flujo de control es visitado al menos una vez.

El conjunto de caminos ([1, 2, 3, 4, 5], [1, 2, 4, 5]) **sí** visita todos los arcos. Si puedo encontrar (x_1, y_1) y (x_2, y_2) que recorren dichos caminos entonces podría decir que T = {(x_1,y_1), (x_2,y_2)} satisface el Criterio de Decisiones.

T_{CD}= {(x=20, y=30), (x=0, y=30)}



Testing de Caja Blanca

Criterios de Cubrimiento

Criterio de Condiciones y Decisiones

Un test T satisface este criterio ssi cada condición simple (atómica) que aparece en cada decisión del programa es evaluada al menos una vez a TRUE y al menos una vez a FALSE para diferentes datos de test.

Una condición es simple (atómica) cuando no presenta operadores lógicos como AND, OR y NOT. Puede tener sí operadores relacionales como >, >=, <, <= e =.

Testing de Caja Blanca

Criterios de Cubrimiento

Criterio de Condiciones y Decisiones

Un test T satisface este criterio ssi cada condición simple (atómica) que aparece en cada decisión del programa es evaluada al menos una vez a TRUE y al menos una vez a FALSE para diferentes datos de test.

```
boolean cond1;
boolean cond2;
…
if( cond1 AND cond2 ) THEN {
… }
```

Testing de Caja Blanca

Criterios de Cubrimiento

Criterio de Condiciones y Decisiones

Un test T satisface este criterio ssi cada condición simple (atómica) que aparece en cada decisión del programa es evaluada al menos una vez a TRUE y al menos una vez a FALSE para diferentes datos de test.

```
boolean cond1;
boolean cond2;
…
if( cond1 AND cond2 ) THEN {
… }
```

Condiciones Simples		Condición
cond1	cond2	cond1 AND cond2

Testing de Caja Blanca

Criterios de Cubrimiento

Criterio de Condiciones y Decisiones

Un test T satisface este criterio ssi cada condición simple (atómica) que aparece en cada decisión del programa es evaluada al menos una vez a TRUE y al menos una vez a FALSE para diferentes datos de test.

```
boolean cond1;
boolean cond2;
…
if( cond1 AND cond2 ) THEN {
… }
```

Condiciones Simples		Condición
cond1	cond2	cond1 AND cond2
TRUE	TRUE	TRUE
FALSE	FALSE	FALSE

Testing de Caja Blanca

Criterios de Cubrimiento

Criterio de Condiciones Múltiples

Un test T satisface este criterio ssi cada condición simple (atómica) que aparece en cada decisión del programa es evaluada a TRUE y a FALSE en todas las combinaciones de la tabla de verdad, para diferentes datos de test.

Testing de Caja Blanca

Criterios de Cubrimiento

Criterio de Condiciones Múltiples

Un test T satisface este criterio ssi cada condición simple (atómica) que aparece en cada decisión del programa es evaluada a TRUE y a FALSE en todas las combinaciones de la tabla de verdad, para diferentes datos de test.

```
boolean
boolean
…
if( cond1 AND cond2 ) THEN {
… }
```

Condiciones Simples		Condición
cond1	cond2	cond1 AND cond2
TRUE	TRUE	TRUE
FALSE	FALSE	FALSE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE

Testing de Caja Blanca

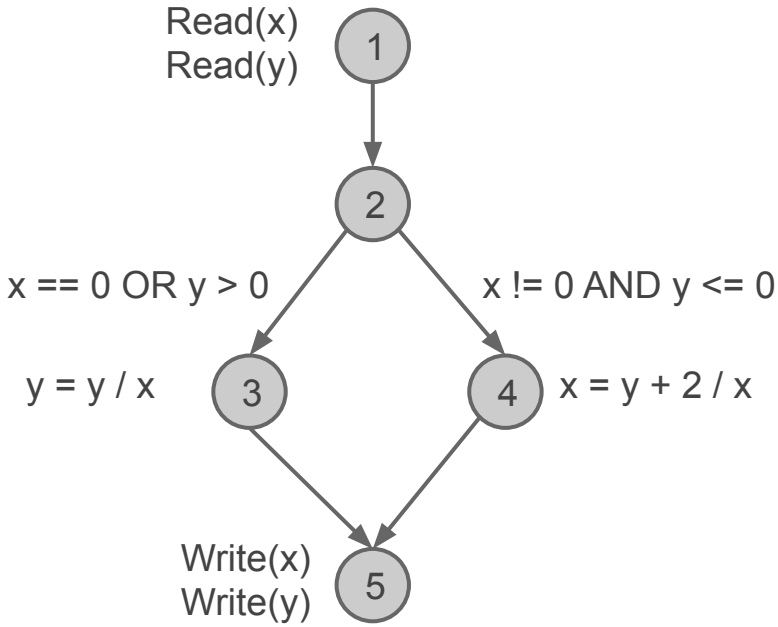
Criterios de Cubrimiento

```
PROGRAM
VAR x,y:REAL
BEGIN
Read(x);
Read(y);
IF (x == 0 OR y > 0 )
THEN
        y = y / x;
ELSE
        x = y + 2 / x;
Write(x);
Write(y);
END
```

Testing de Caja Blanca

Criterios de Cubrimiento

```
PROGRAM
VAR x,y:REAL
BEGIN
Read(x);
Read(y);
IF (x == 0 OR y > 0 )
THEN
        y = y / x;
ELSE
        x = y + 2 / x;
Write(x);
Write(y);
END
```

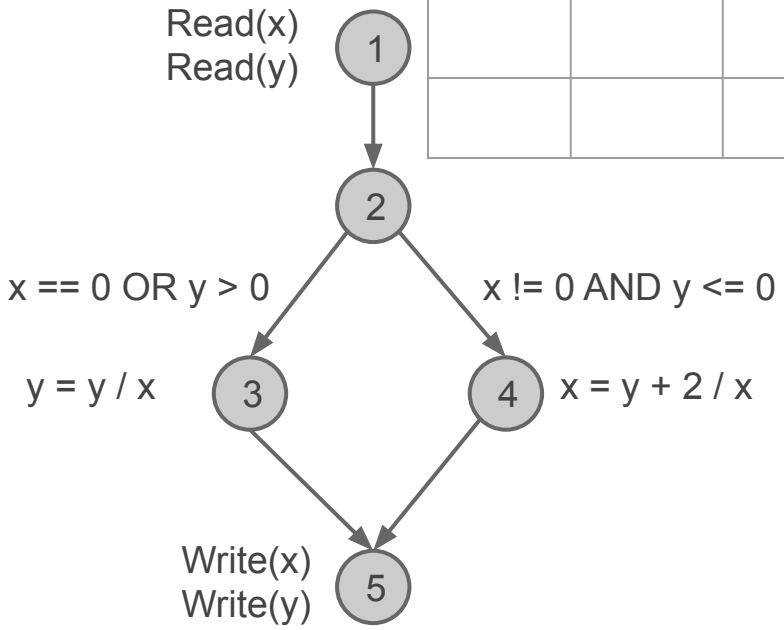


Testing de Caja Blanca

Criterios de Cubrimiento

Criterio de Condiciones y Decisiones

x	y	x == 0	y > 0	x == 0 OR y > 0

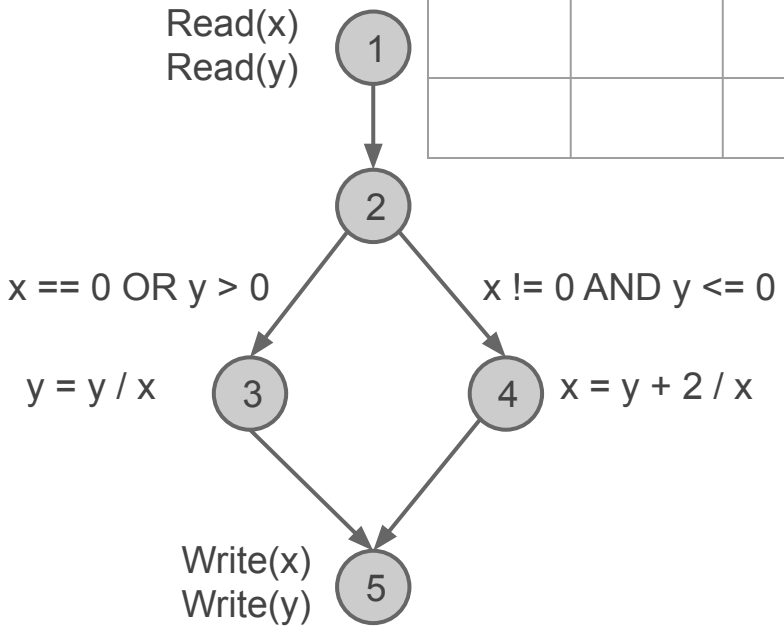


Testing de Caja Blanca

Criterios de Cubrimiento

Criterio de Condiciones y Decisiones

x	y	**x == 0**	**y > 0**	x == 0 OR y > 0

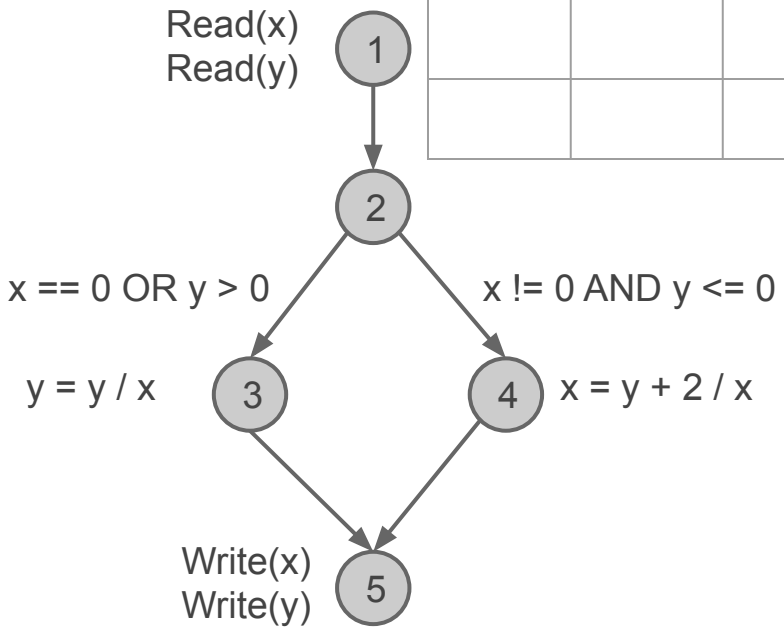


Testing de Caja Blanca

Criterios de Cubrimiento

Criterio de Condiciones y Decisiones

x	y	**x == 0**	**y > 0**	x == 0 OR y > 0
		TRUE	**TRUE**	
		FALSE	**FALSE**	

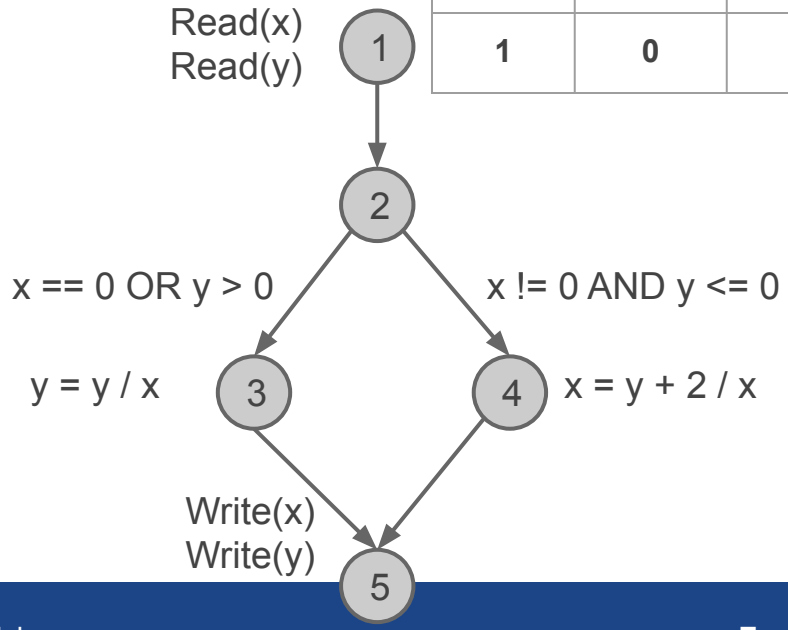


Testing de Caja Blanca

Criterios de Cubrimiento

Criterio de Condiciones y Decisiones

x	y	x == 0	y > 0	x == 0 OR y > 0
0	**1**	**TRUE**	**TRUE**	
1	**0**	**FALSE**	**FALSE**	

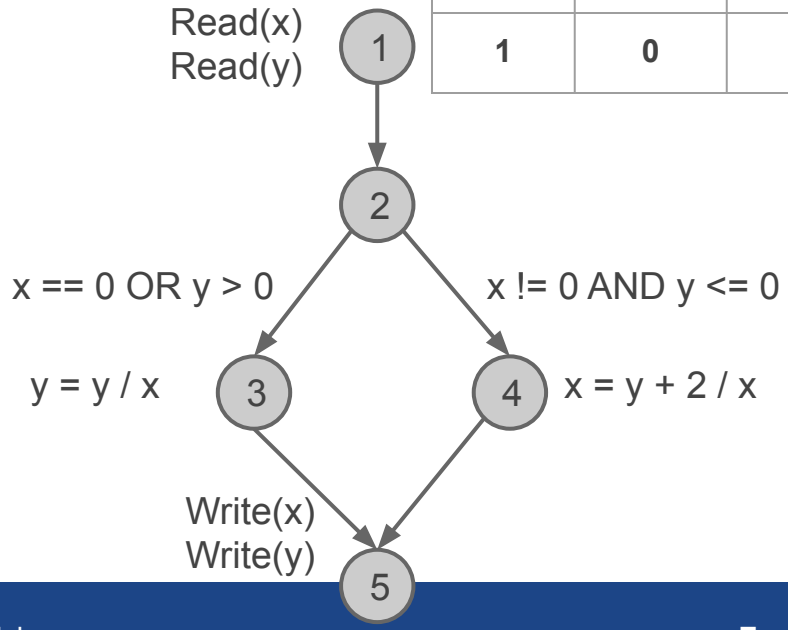


Testing de Caja Blanca

Criterios de Cubrimiento

Criterio de Condiciones y Decisiones

x	y	x == 0	y > 0	x == 0 OR y > 0
0	1	TRUE	TRUE	TRUE
1	0	FALSE	FALSE	FALSE

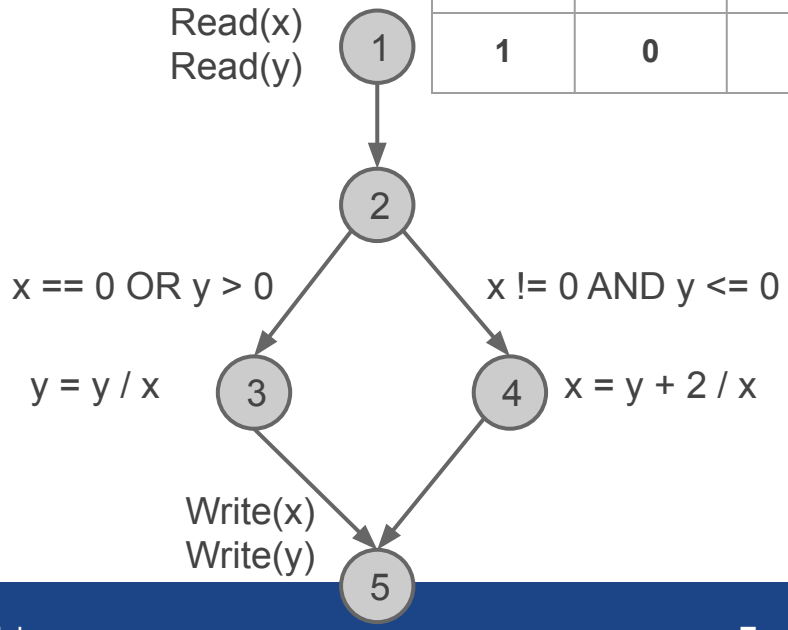


Testing de Caja Blanca

Criterios de Cubrimiento

Criterio de Condiciones y Decisiones

x	y	x == 0	y > 0	x == 0 OR y > 0
0	1	TRUE	TRUE	TRUE
1	0	FALSE	FALSE	FALSE

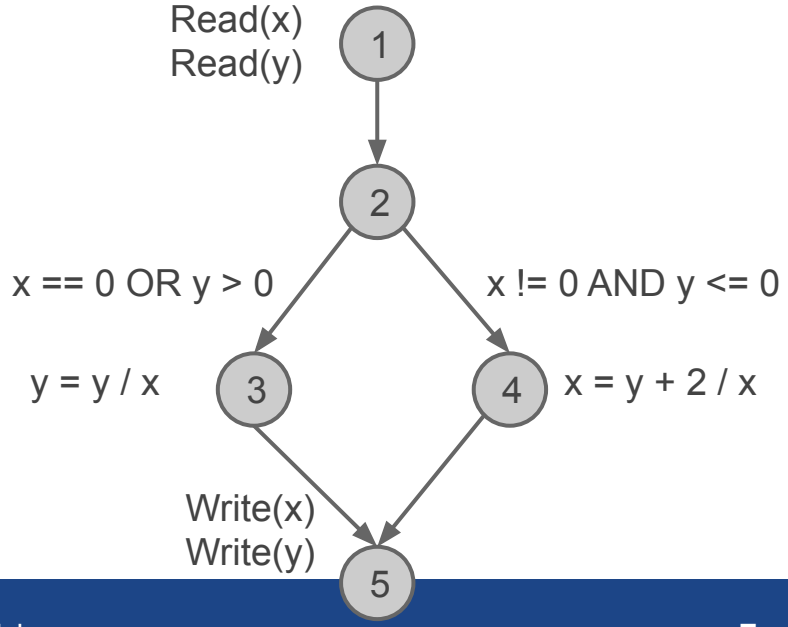


T_{CCD}= {(x=1, y=0), (x=0, y=1)}

Testing de Caja Blanca

Criterios de Cubrimiento

Criterio de Condiciones Múltiples

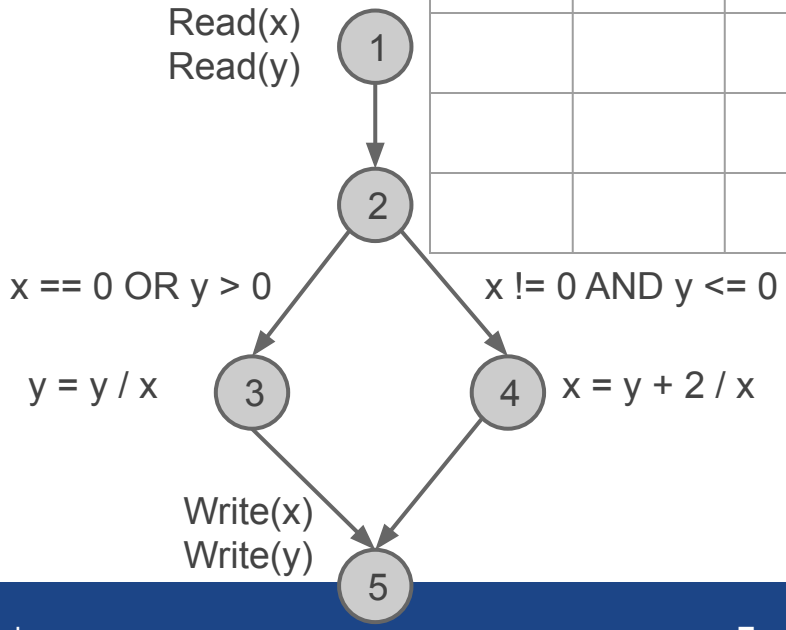


Testing de Caja Blanca

Criterios de Cubrimiento

Criterio de Condiciones Múltiples

x	y	x == 0	y > 0	x == 0 OR y > 0

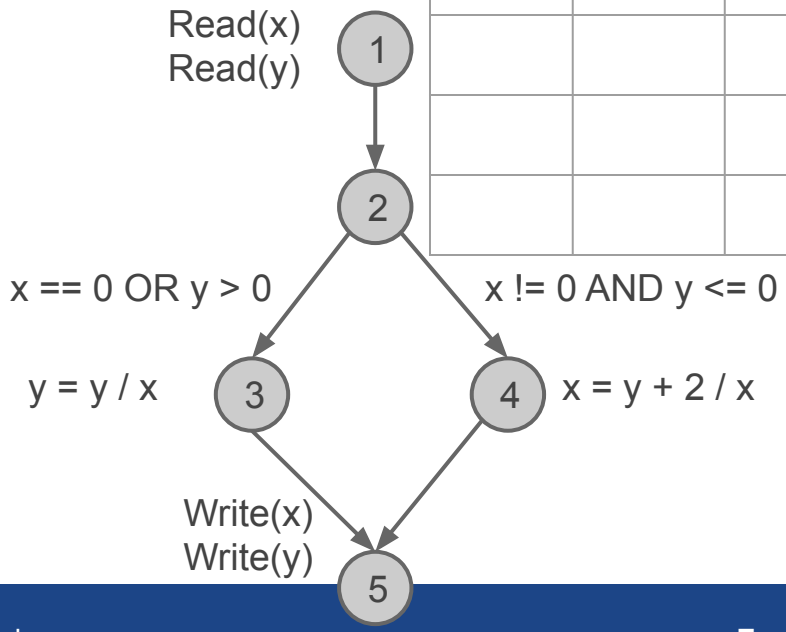


Testing de Caja Blanca

Criterios de Cubrimiento

Criterio de Condiciones Múltiples

x	y	**x == 0**	**y > 0**	x == 0 OR y > 0
		TRUE	**TRUE**	
		TRUE	**FALSE**	
		FALSE	**TRUE**	
		FALSE	**FALSE**	

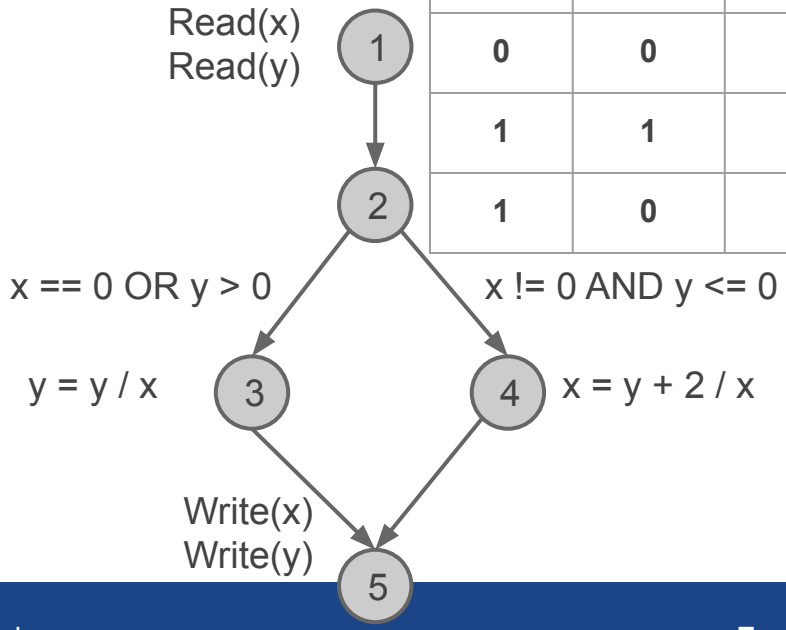


Testing de Caja Blanca

Criterios de Cubrimiento

Criterio de Condiciones Múltiples

x	y	x == 0	y > 0	x == 0 OR y > 0
0	**1**	**TRUE**	**TRUE**	
0	**0**	**TRUE**	**FALSE**	
1	**1**	**FALSE**	**TRUE**	
1	**0**	**FALSE**	**FALSE**	

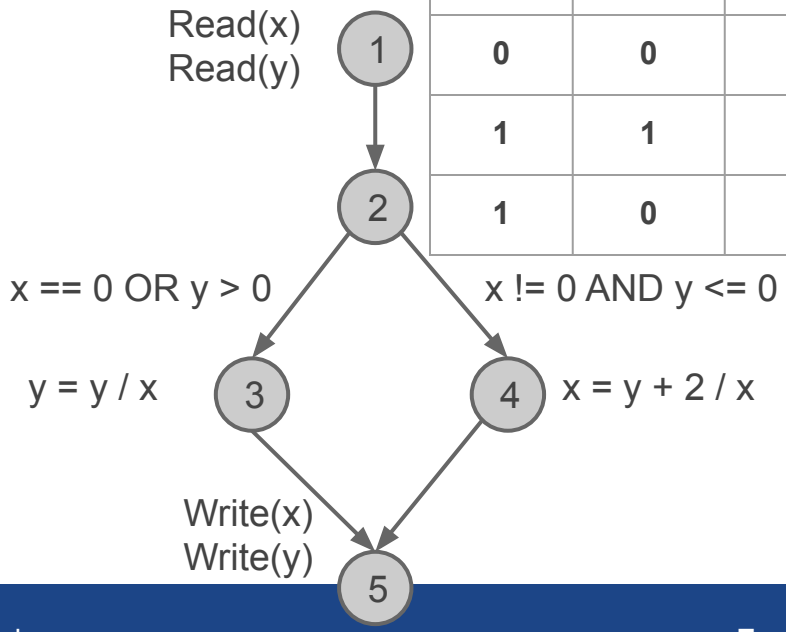


Testing de Caja Blanca

Criterios de Cubrimiento

Criterio de Condiciones Múltiples

x	y	x == 0	y > 0	x == 0 OR y > 0
0	1	TRUE	TRUE	TRUE
0	0	TRUE	FALSE	TRUE
1	1	FALSE	TRUE	TRUE
1	0	FALSE	FALSE	FALSE

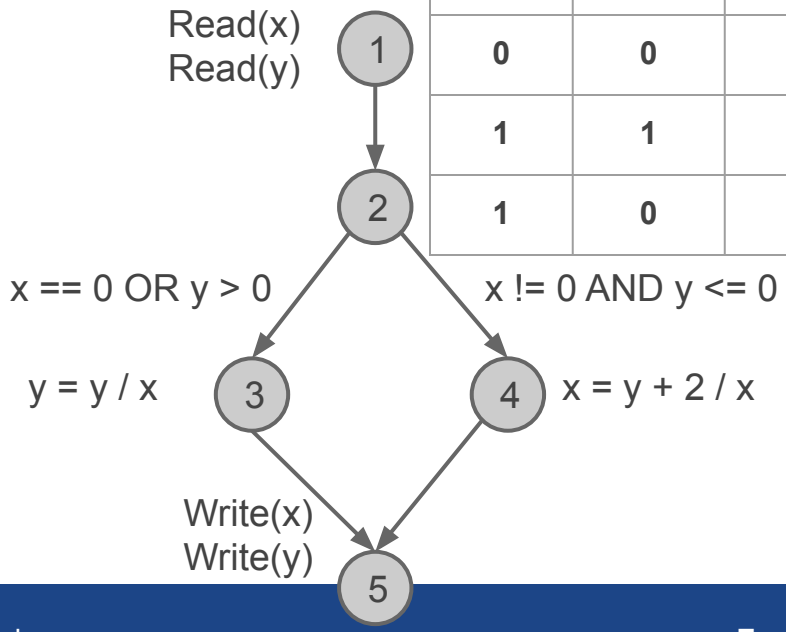


Testing de Caja Blanca

Criterios de Cubrimiento

Criterio de Condiciones Múltiples

x	y	x == 0	y > 0	x == 0 OR y > 0
0	1	TRUE	TRUE	TRUE
0	0	TRUE	FALSE	TRUE
1	1	FALSE	TRUE	TRUE
1	0	FALSE	FALSE	FALSE



T_{CCM}= {(x=1, y=0), (x=0, y=1), (x=1, y=1), (x=0, y=0)}

Testing de Caja Blanca

Criterios de Cubrimiento

Criterio de Caminos

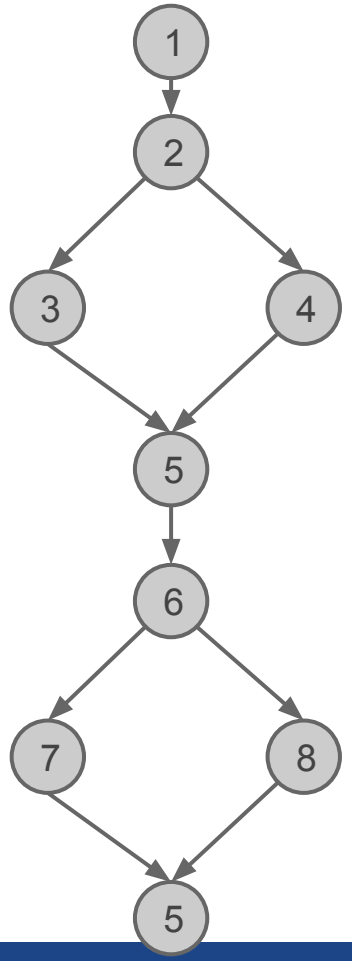
Un test T satisface este criterio ssi todos los caminos desde el nodo inicial del grafo, al nodo final son atravesados.

Testing de Caja Blanca

Criterios de Cubrimiento

Criterio de Caminos

Un test T satisface este criterio ssi todos los caminos desde el nodo inicial del grafo, al nodo final son atravesados.



Testing de Caja Blanca

Criterios de Cubrimiento

Criterio de Caminos

Un test T satisface este criterio ssi todos los caminos desde el nodo inicial del grafo, al nodo final son atravesados.

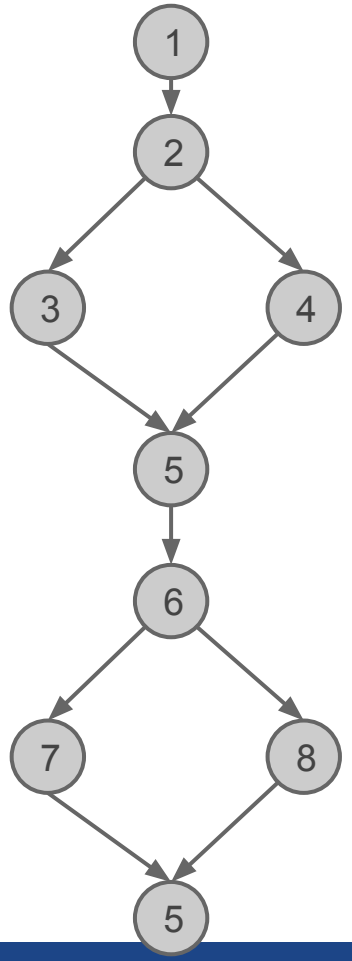
Todos los caminos:

[1, 2, 3, 5, 6, 7, 5]

[1, 2, 3, 5, 6, 8, 5]

[1, 2, 4, 5, 6, 7, 5]

[1, 2, 4, 5, 6, 8, 5]



Testing de Caja Blanca

Criterios de Cubrimiento

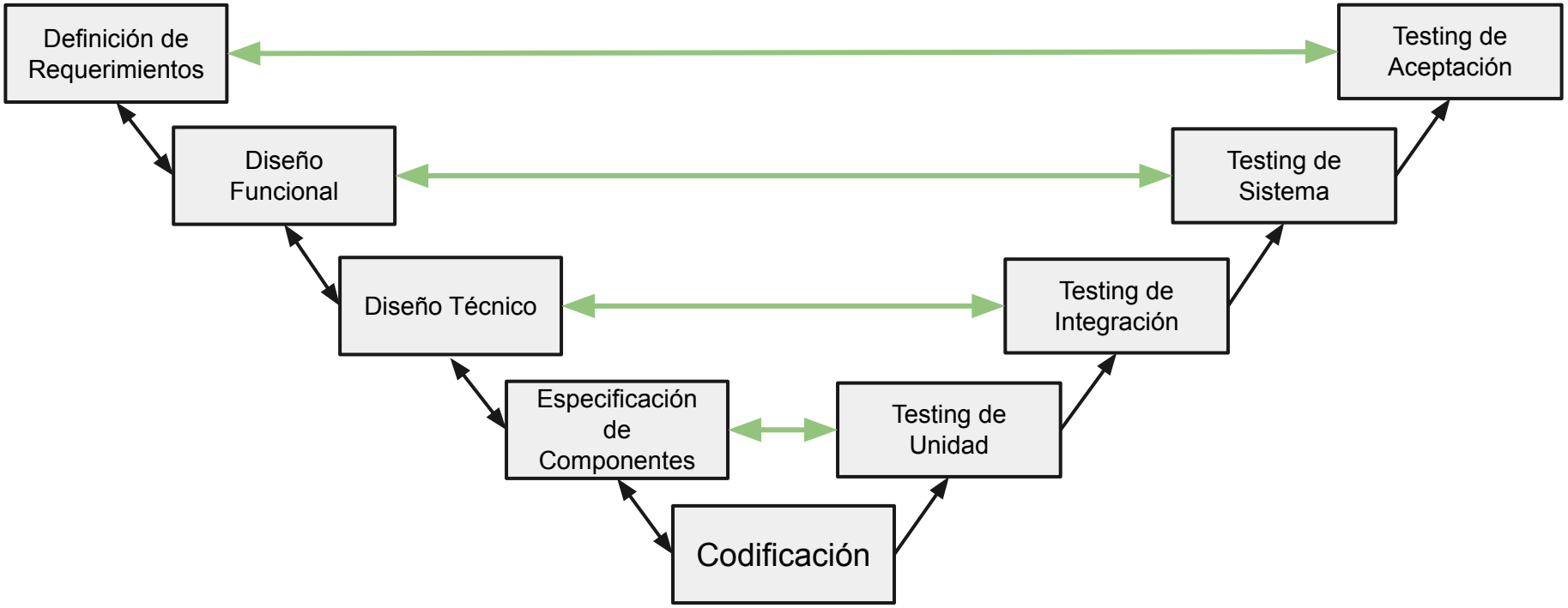
Criterio de Bucles

Los bucles son casos especiales. Se deberían cubrir los siguientes casos, cuando sea posible.

- Ejecutar 0 veces el bucle, o el mínimo número de veces.
- Ejecutar el caso máximo posible de veces, si es determinable.
- Ejecutar un número promedio de veces, si es un valor calculable.

Testing en el Ciclo de Vida

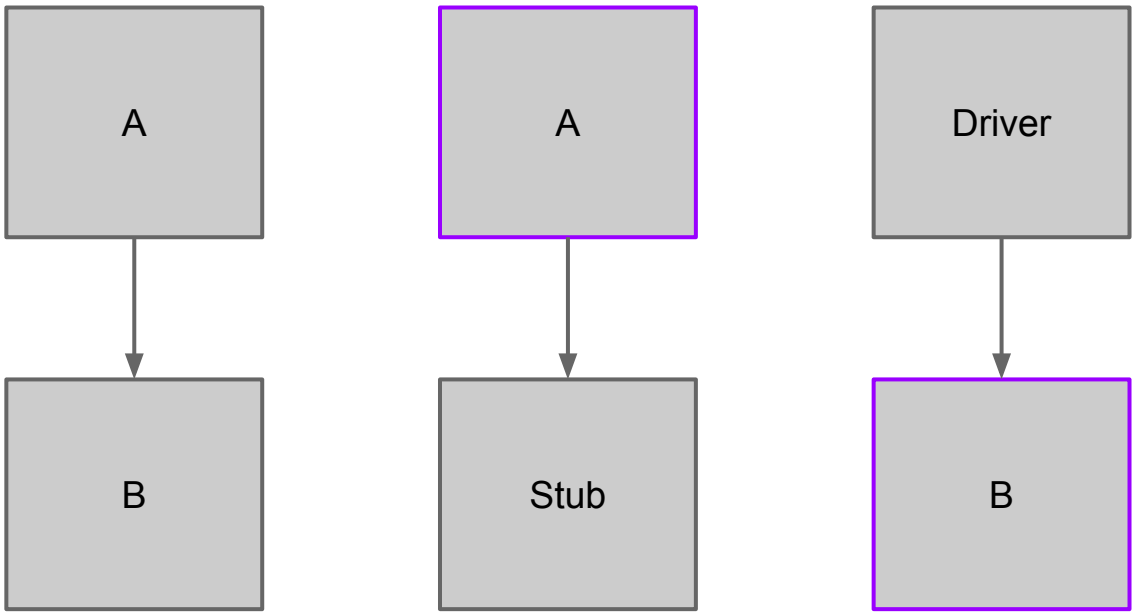
En este modelo, el Testing toma el mismo nivel de importancia que el Diseño y el Desarrollo.



Testing en el Ciclo de Vida

Testing de Unidad

Component Testing



Testing en el Ciclo de Vida

Testing de Integración

Integration Testing

Segunda etapa del testing. En esta etapa se asume que las componentes (unidades) fueron testeadas y sus defectos fueron resueltos.

Testing en el Ciclo de Vida

Testing de Integración

Integration Testing

Se define como “integración” a la conexión entre componentes para formar un sistema o unidad mayor.

Testing en el Ciclo de Vida

Testing de Integración

Integration Testing

Se define como “integración” a la conexión entre componentes para formar un sistema o unidad mayor.

El objetivo en esta etapa es detectar fallas en las interfaces e interacciones entre componentes. También se pueden incluir test de integración con sistemas externos.

Testing en el Ciclo de Vida

Testing de Integración

Integration Testing

Algunos de los errores que se pueden encontrar son:

- Formatos de interfaces erróneos
- Errores en el intercambio de información

Testing en el Ciclo de Vida

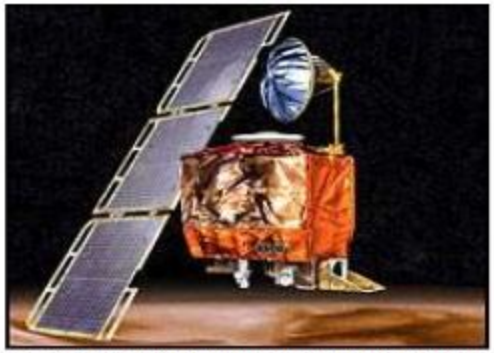
Testing de Integración

Integration Testing

Metric Mishap Caused Loss of NASA Orbiter

* By Kathy Sawyer & Robin Lloyd (Washington Post Staff Writer & CNN Interactive Senior Writer)

NASA's Mars Climate Orbiter was lost in space last week because engineers failed to make a simple conversion from English units to metric, an embarrassing lapse that sent the $125 million craft fatally close to the Martian surface.



Scientists do not yet know what caused the Mars Orbiter to lose contact with Earth. (AP)

http://mytoe.org/docs/Mars_closereading_art.pdf

Testing en el Ciclo de Vida

Testing de Integración

Integration Testing

Se define como “integración” a la conexión entre componentes para formar un sistema o unidad mayor.

El objetivo en esta etapa es detectar fallas en las interfaces e interacciones entre componentes.

Testing en el Ciclo de Vida

Testing de Integración

Integration Testing

No todas las unidades se terminan al mismo tiempo, esto obliga a manejar diferentes tiempos de testing. El scheduling del desarrollo de componentes es un input importante para la planificación del testing de integración.

Testing en el Ciclo de Vida

Testing

Integration Testing

Stubs .NET

Stubs is a lightw
classes, type-sa
Stubs are part of

Stubs is a lightw
may be used on
minimal overhea
rewriter and enc

Download
Read more

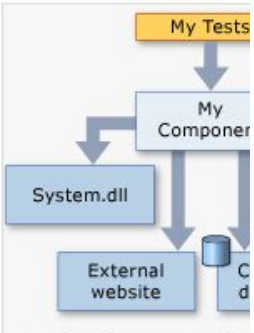
Isolating Code Under Test with Microsoft Fakes

Visual Studio 2013 | Other Versions

Microsoft Fakes help you isolate the code you are testing by replacing other parts of the application with *stubs* or *shims*. These are s
there and not somewhere else. Stubs and shims also let you test your code even if other parts of your application are not working ye

Fakes come in two flavors

- A stub replaces a c
  classes that are de
- A shim modifies th



work under the control of the tests.

Moles - Isolation framework for .NET

The Fakes Framework in Visual Studio 2012 is the next generation of Moles & Stubs. Fakes is different from Moles, however, so moving from Moles to Fakes will require some modifications to your code. The Moles framework will not be supported in Visual Studio 2012.

Testing en el Ciclo de Vida

Testing de Integración

Integration Testing

Existen diferentes estrategias de integración:

Top-Down, Bottom-Up, Ad-hoc, Backbone, Big-Bang.

Testing en el Ciclo de Vida

Testing de Integración

Integration Testing

Existen diferentes estrategias de integración:

Top-Down, Bottom-Up, Ad-hoc, Backbone, Big-Bang.

No requiere Test Drivers / Requiere Stubs

Testing en el Ciclo de Vida

Testing de Integración

Integration Testing

Existen diferentes estrategias de integración:

Top-Down, Bottom-Up, Ad-hoc, Backbone, Big-Bang.

No requiere Stubs / Requiere Test Drivers

Testing en el Ciclo de Vida

Testing de Integración

Integration Testing

Existen diferentes estrategias de integración:

Top-Down, Bottom-Up, Ad-hoc, Backbone, Big-Bang.

Se ahorra tiempo / Requiere Stubs y Drivers

Testing en el Ciclo de Vida

Testing de Integración

Integration Testing

Existen diferentes estrategias de integración:

Top-Down, Bottom-Up, Ad-hoc, Backbone, Big-Bang.

El orden no es importante / Buscar el Backbone

Testing en el Ciclo de Vida

Testing de Integración

Integration Testing

Existen diferentes estrategias de integración:

Top-Down, Bottom-Up, Ad-hoc, Backbone, Big-Bang.

Verificación y Validación de Software

El objetivo del testing es encontrar errores, diferencias entre lo observado y lo esperado.

Verificación y Validación de Software

El objetivo del testing es encontrar errores, diferencias entre lo observado y lo esperado.

Si consideramos que un conjunto de test T es adecuado y un programa P pasa T, entonces podemos decir que el programa está libre de errores*.

Verificación y Validación de Software

Si un programa tiene errores, entonces el test debería señalarlos.

Verificación y Validación de Software

Entonces...

Verificación y Validación de Software

Entonces...

Sea *P* un programa

Verificación y Validación de Software

Entonces...

Sea *P* un programa, para este ejemplo *P* es un programa que determina si un número es par o no.

```
1
2  public class Main {
3
4      public boolean esPar(int x) {
5          return (x % 2 == 0);
6      }
7
8  }
9
```

Verificación y Validación de Software

Entonces...

Sea *P* un programa, para este ejemplo *P* es un programa que determina si un número es par o no.

Sea *T* un conjunto de test para *P*. En particular supongamos que *T* adecuado y que *P* pasa T.

P no tiene errores.

Verificación y Validación de Software

Entonces...

$T = T^{SI} \cup T^{NO}$

$T^{SI} = \{ 6, 12, 24 \}$

$T^{NO} = \{1, 7, 11\}$

```
1
2  public class Main {
3
4      public boolean esPar(int x) {
5          return (x % 2 == 0);
6      }
7
8  }
9
```

Verificación y Validación de Software

Entonces...

T = T^{SI} U T^{NO}

T^{SI} = { 6, 12, 24 }

T^{NO} = {1, 7, 11}

T nos asegura que *P* no tiene errores

```
1
2  public class Main {
3
4      public boolean esPar(int x) {
5          return (x % 2 == 0);
6      }
7
8  }
9
```

Verificación y Validación de Software

ENTONCES...

Si *T* es tan bueno, si yo ingresara un error en *P* a propósito, *T* debería ser capaz de detectarlo.

$T = T^{SI} \cup T^{NO}$

$T^{SI} = \{ 6, 12, 24 \}$

$T^{NO} = \{1, 7, 11\}$

```
1
2  public class Main {
3
4      public boolean esPar(int x) {
5          return (x % 2 == 0);
6      }
7
8  }
9
```

Verificación y Validación de Software

ENTONCES...

Si *T* es tan bueno, si yo ingresara un error en *P* a propósito, *T* debería ser capaz de detectarlo.

$T = T^{SI} \cup T^{NO}$

$T^{SI} = \{ 6, 12, 24 \}$

$T^{NO} = \{1, 7, 11\}$

```
1
2  public class Main {
3
4      public boolean esPar(int x) {
5          return (x % 2 == 0);
6      }
7
8  }
9
```

Verificación y Validación de Software

ENTONCES...

Sea P^1 el resultado de introducir un cambio sintáctico en *P.*
Introduje un error, por lo tanto T debería detectarlo.

$T = T^{SI} \cup T^{NO}$

$T^{SI} = \{ 6, 12, 24 \}$

$T^{NO} = \{1, 7, 11\}$

```
1
2  public class Main {
3
4      public boolean esPar(int x) {
5          return (x % 3 == 0);
6      }
7
8  }
9
```

Verificación y Validación de Software

ENTONCES...

Sea P^1 el resultado de introducir un cambio sintáctico en *P.*
Introduje un error, por lo tanto T debería detectarlo.

T no reconoce el error

$T = T^{SI} \cup T^{NO}$

$T^{SI} = \{ 6, 12, 24 \}$

$T^{NO} = \{1, 7, 11\}$

```
1
2  public class Main {
3
4      public boolean esPar(int x) {
5          return (x % 3 == 0);
6      }
7
8  }
9
```

Verificación y Validación de Software

ENTONCES...

Sea P^1 el resultado de introducir un cambio sintáctico en P.

Introduje u

o.

ce el error

T = T^{SI} U T

T^{SI} = { 6, 1

T^{NO} = {1, 7

```
(int x) {
    return (x % 3 == 0);
}

}
```

Verificación y Validación de Software

ENTONCES...

Esta experiencia me sirve para ver que *T* no es un buen conjunto de test y además me sirve para mejorar *T*. Si analizo el error que introduje, puedo mejorar los casos de test.

$T = T^{SI} \cup T^{NO}$

$T^{SI} = \{ 6, 12, 24 \}$

$T^{NO} = \{1, 7, 11\}$

```
1
2  public class Main {
3
4      public boolean esPar(int x) {
5          return (x % 3 == 0);
6      }
7
8  }
9
```

Verificación y Validación de Software

ENTONCES...

Esta experiencia me sirve para ver que *T* no es un buen conjunto de test y además me sirve para mejorar *T*. Si analizo el error que introduje, puedo mejorar los casos de test.

$D = D^{SI} \cup D^{NO}$

$D^{SI} = \{ 6, 12, 24, 2 \}$

$D^{NO} = \{1, 7, 11, 9\}$

```
1
2  public class Main {
3
4      public boolean esPar(int x) {
5          return (x % 3 == 0);
6      }
7
8  }
9
```

Verificación y Validación de Software

ENTONCES...

El nuevo conjunto de test *D* reconoce que P^1 tiene errores, por lo que es un mejor testeo que *T*.

D = D^{SI} U D^{NO}

D^{SI} = { 6, 12, 24, 2 }

D^{NO} = {1, 7, 11, 9}

```
1
2  public class Main {
3
4      public boolean esPar(int x) {
5          return (x % 3 == 0);
6      }
7
8  }
9
```

Verificación y Validación de Software

Testing por Mutación

Mutation Testing

Verificación y Validación de Software

Testing por Mutación

Mutation Testing

Una mutación es un pequeño cambio en un programa. Estos cambios buscar reflejar los pequeños errores que se producen al programar.

Verificación y Validación de Software

Testing por Mutación

Mutation Testing

El Testing por Mutación es una técnica de caja blanca que permite evaluar/mejorar la calidad de nuestra suite de test.

Verificación y Validación de Software

Testing por Mutación

Mutation Testing

Dado un programa *P* y una suite de test *T* el proceso es:

Aplicamos mutaciones sobre *P* para obtener una secuencia $P^1 ... P^n$ de mutantes de *P*.

Verificación y Validación de Software

Testing por Mutación

Mutation Testing

Dado un programa *P* y una suite de test *T* el proceso es:

Aplicamos mutaciones sobre *P* para obtener una secuencia $P^1 ... P^n$ de mutantes de *P*.

Ejecutamos *T* sobre cada mutante. Se dice que *T* mata al mutante P^j si *T* detecta un error en P^j.

Verificación y Validación de Software

Testing por Mutación

Mutation Testing

Dado un programa *P* y una suite de test *T* el proceso es:

Aplicamos mutaciones sobre *P* para obtener una secuencia $P^1 ... P^n$ de mutantes de *P*.

Ejecutamos *T* sobre cada mutante. Se dice que *T* mata al mutante P^j si *T* detecta un error en P^j.

Si *T* mata a *k* mutantes entonces se dice que la eficacia (adequacy) de *T* es *k / n. T* es **mutation adequacy** si *k* = *n*.

Verificación y Validación de Software

Testing por Mutación

Mutation Testing

Para poder aprovechar esta técnica debemos poder sistematizar el proceso de mutación. Esto implica definir reglas para la aplicación de mutaciones.

Verificación y Validación de Software

Testing por Mutación

Mutation Testing

Para poder aprovechar esta técnica debemos poder sistematizar el proceso de mutación. Esto implica definir reglas para la aplicación de mutaciones.

Obviamente que, para una simple línea de código podemos pensar en muchas mutaciones. Esto llevado a un programa complejo implica que la cantidad de mutantes, si queremos mutar todo lo posible, es astronómica.

Verificación y Validación de Software

Testing por Mutación

Mutation Testing

Mutación de Valor

Mutación de Decisión

Mutación de Sentencia

Verificación y Validación de Software

Testing por Mutación

Mutation Testing

Mutación de Valor

Modificamos el valor de una constante

Mutación de Decisión

Mutación de Sentencia

Verificación y Validación de Software

Testing por Mutación

Mutation Testing

Mutación de Valor

Mutación de Decisión

Modificamos condiciones en decisiones

Mutación de Sentencia

Verificación y Validación de Software

Testing por Mutación

Mutation Testing

Mutación de Valor

Mutación de Decisión

Mutación de Sentencia

Eliminamos sentencias, las duplicamos o intercambiamos

Verificación y Validación de Software

Testing por Mutación

Mutation Testing

Mutación de Valor

El objetivo es reflejar errores en el razonamiento de la solución por parte del programador.

Recorrer un arreglo a partir de 1 en lugar de 0.

El caso base de factorial en 1 retorna 0.

Cubrimiento: Modificar cada constante del programa por lo menos una vez (o dos).

Verificación y Validación de Software

Testing por Mutación

Mutation Testing

Mutación de Decisión

El objetivo es reflejar errores en el razonamiento de las decisiones en la solución, por parte del programador.

Controlar un if por < *cuando debería ser* <=

Ciclar sobre un && *cuando debería ser* ||

Cubrimiento: Modificar cada condición del programa una vez. Modificar una vez cada operador lógico y de relación una vez.

Verificación y Validación de Software

Testing por Mutación

Mutation Testing

Mutación de Sentencia

El objetivo es reflejar errores en la edición del código, muchas veces por copiar y pegar

Borrar, duplicar e intercambiar una sentencia

Cubrimiento: Aplicar cada una de estas acciones en cada una de las sentencias del programa.

Verificación y Validación de Software

Testing por Mutación

Mutation Testing

Esta técnica es práctica sólo si podemos automatizar el proceso de mutación y testeo.

Verificación y Validación de Software

Testing por Mutación

Mutation Testing

An Analysis and Survey of the Development of Mutation Testing

Yue Jia *Student Member, IEEE,* and Mark Harman *Member, IEEE*

***Abstract*— Mutation Testing is a fault–based software testing technique that has been widely studied for over three decades. The literature on Mutation Testing has contributed a set of approaches, tools, developments and empirical results. This paper provides a comprehensive analysis and survey of Mutation Testing. The paper also presents the results of several development trend analyses. These analyses provide evidence that Mutation Testing techniques and tools are reaching a state of maturity and applicability, while the topic of Mutation Testing itself is the subject of increasing interest.**

***Index Terms*— mutation testing, survey**

Besides using Mutation Testing at the software implementation level, it has also been applied at the design level to test the specifications or models of a program. For example, at the design level Mutation Testing has been applied to Finite State Machines [20], [28], [88], [111], State charts [95], [231], [260], Estelle Specifications [222], [223], Petri Nets [86], Network protocols [124], [202], [216], [238], Security Policies [139], [154], [165], [166], [201] and Web Services [140], [142], [143], [193], [245], [259].

Mutation Testing has been increasingly and widely studied since it was first proposed in the 1970s. There has been much research work on the various kinds of techniques seeking to

Verificación y Validación de Software

Testing por Mutación

Mutation Testing

MuJava : An Automated Class Mutation System

Yu-Seung Ma[1]*, Jeff Offutt[2]†, and Yong Rae Kwon[1]

[1]Division of Computer Science
Department of Electrical Engineering and Computer Science
Korea Advanced Institute of Science and Technology, Korea
{*ysma, kwon*}*@salmosa.kaist.ac.kr*

[2]Department of Information and Software Engineering
George Mason University
ofut@ise.gmu.edu

Abstract

Several module and class testing techniques have been applied to object-oriented programs, but researchers have only recently begun developing test criteria that evaluate the use of key OO features such as inheritance, polymorphism, and encapsulation. Mutation testing is a powerful testing technique for generating software tests and evaluating the quality of software. However, the cost of mutation testing has traditionally been so high it cannot be applied without full automated tool support.

This paper presents a method to reduce the execution cost of mutation testing for OO programs by

Verificación y Validación de Software

Testing por Mutación

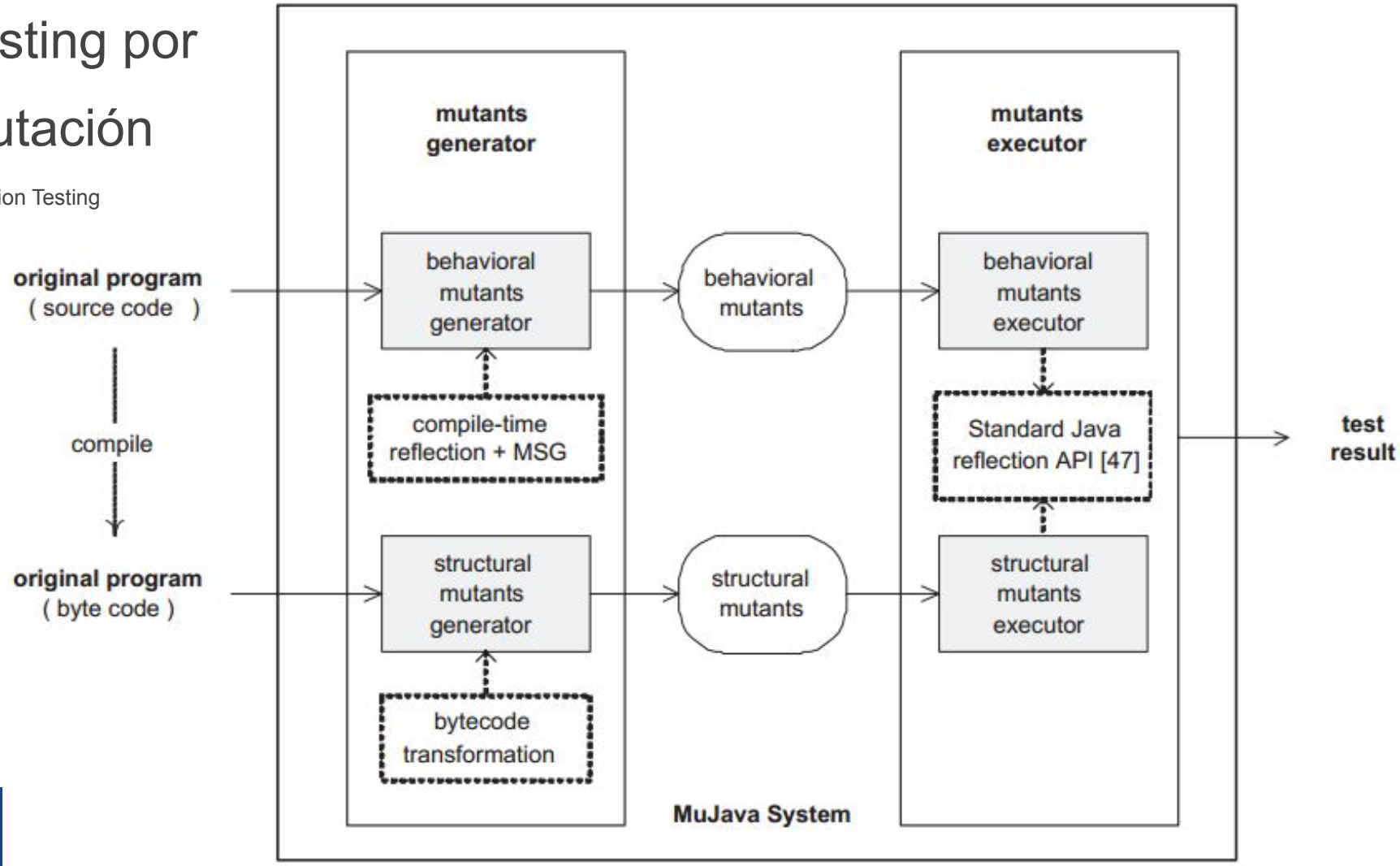
Mutation Testing

Language Feature	Operator	Description
Access Control	AMC	Access modifier change
Inheritance	IHD	Hiding variable deletion
	IHI	Hiding variable insertion
	IOD	Overriding method deletion
	IOP	overriding method calling position change
	IOR	Overriding method rename
	ISK	*super* keyword deletion
	IPC	Explicit call of a parent's constructor deletion
Polymorphism	PNC	*new* method call with child class type
	PMD	Instance variable declaration with parent class type
	PPD	Parameter variable declaration with child class type
	PRV	Reference assignment with other comparable type
Overloading	OMR	Overloading method contents change
	OMD	Overloading method deletion
	OAO	Argument order change
	OAN	Argument number change
Java-Specific Features	JTD	*this* keyword deletion
	JSC	*static* modifier change
	JID	Member variable initialization deletion
	JDC	Java-supported default constructor creation
Common Programming Mistakes	EOA	Reference assignment and content assignment replacement
	EOC	Reference comparison and content comparison replacement
	EAM	Accessor method change
	EMM	Modifier method change

Verificación y Validación de Software

Testing por Mutación

Mutation Testing



Verificación y Validación de Software

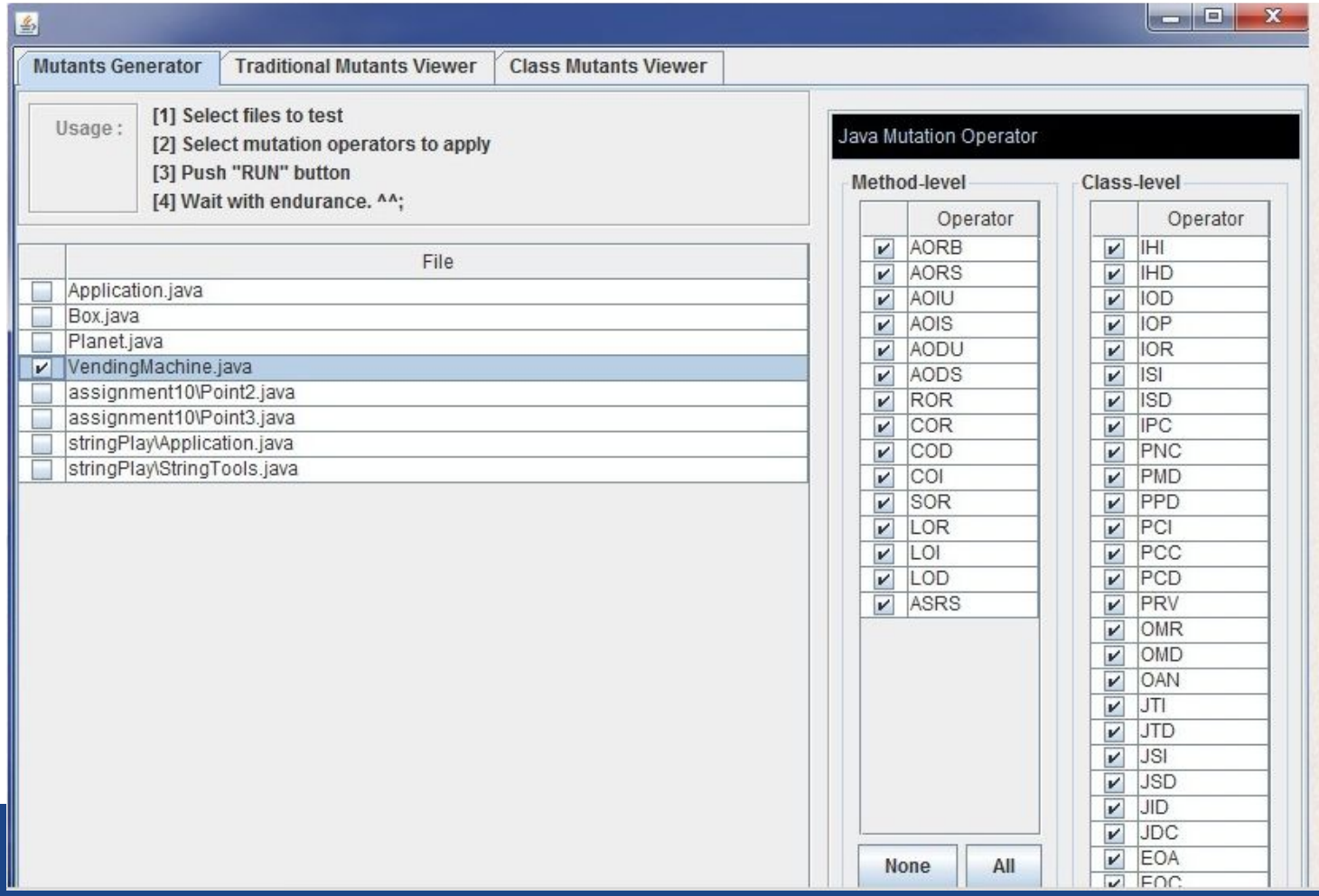
Testing por Mutación

Mutation Testing

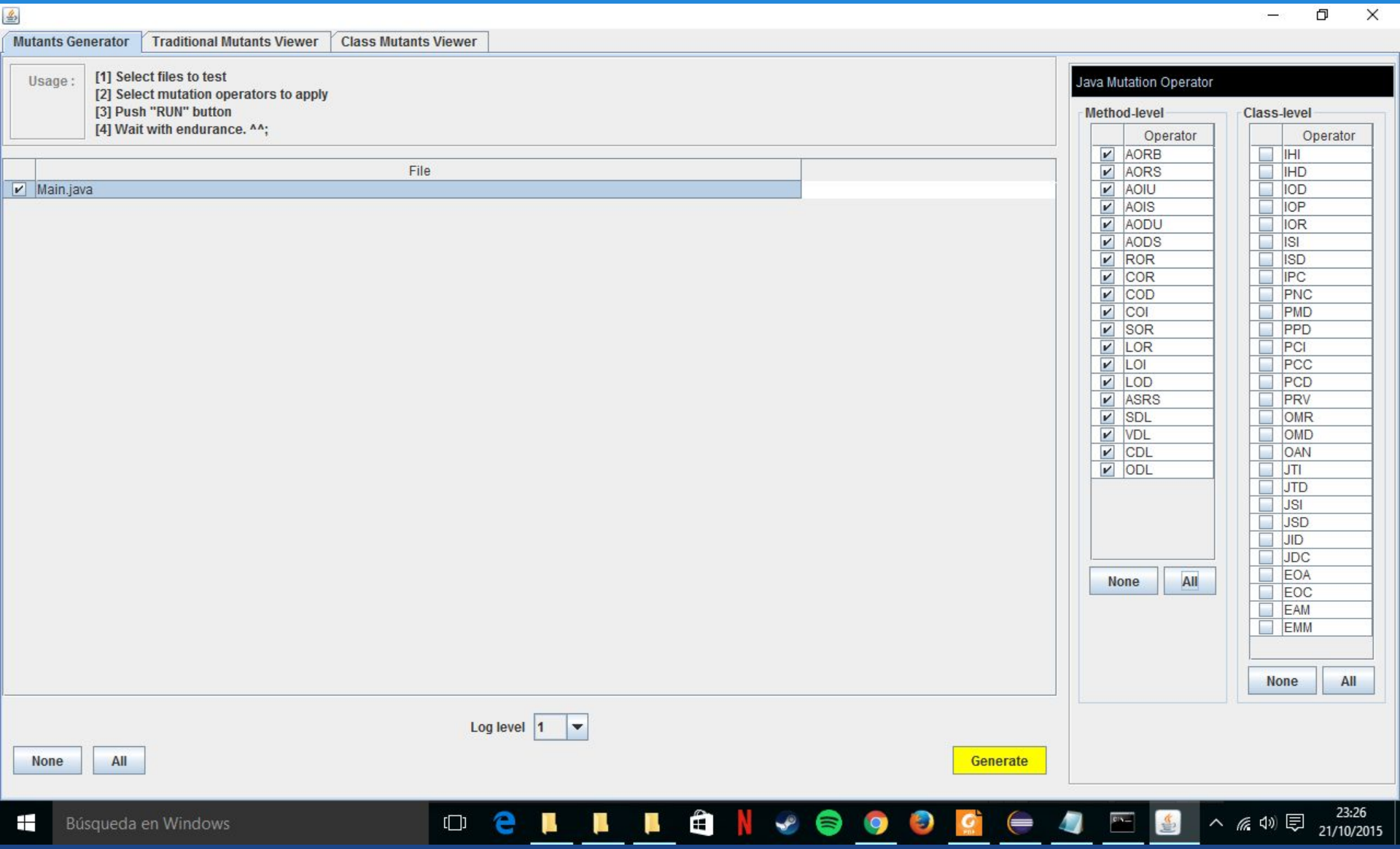
Verificación y Validación de Software

Testing por Mutación

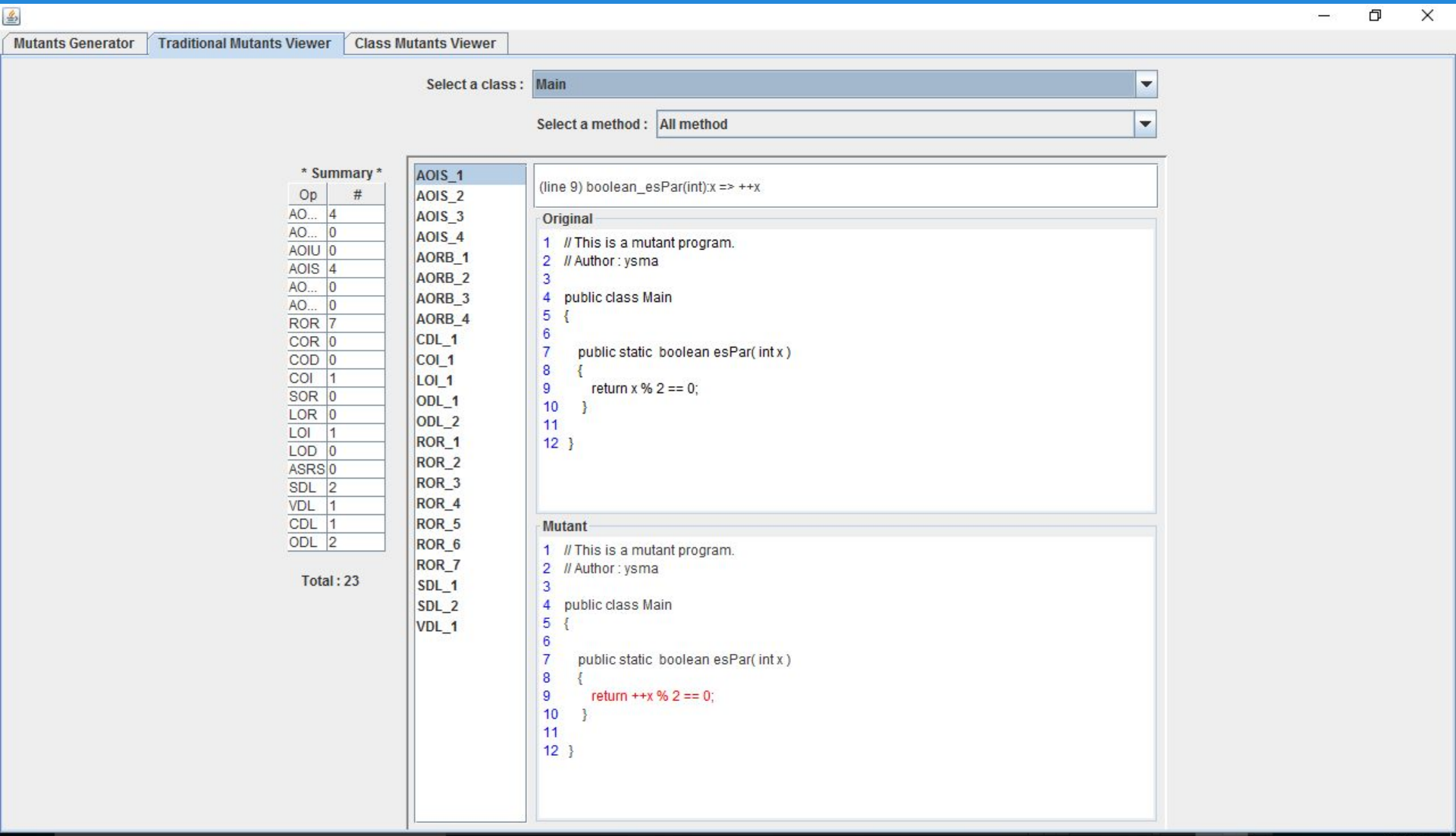
Mutation Testing



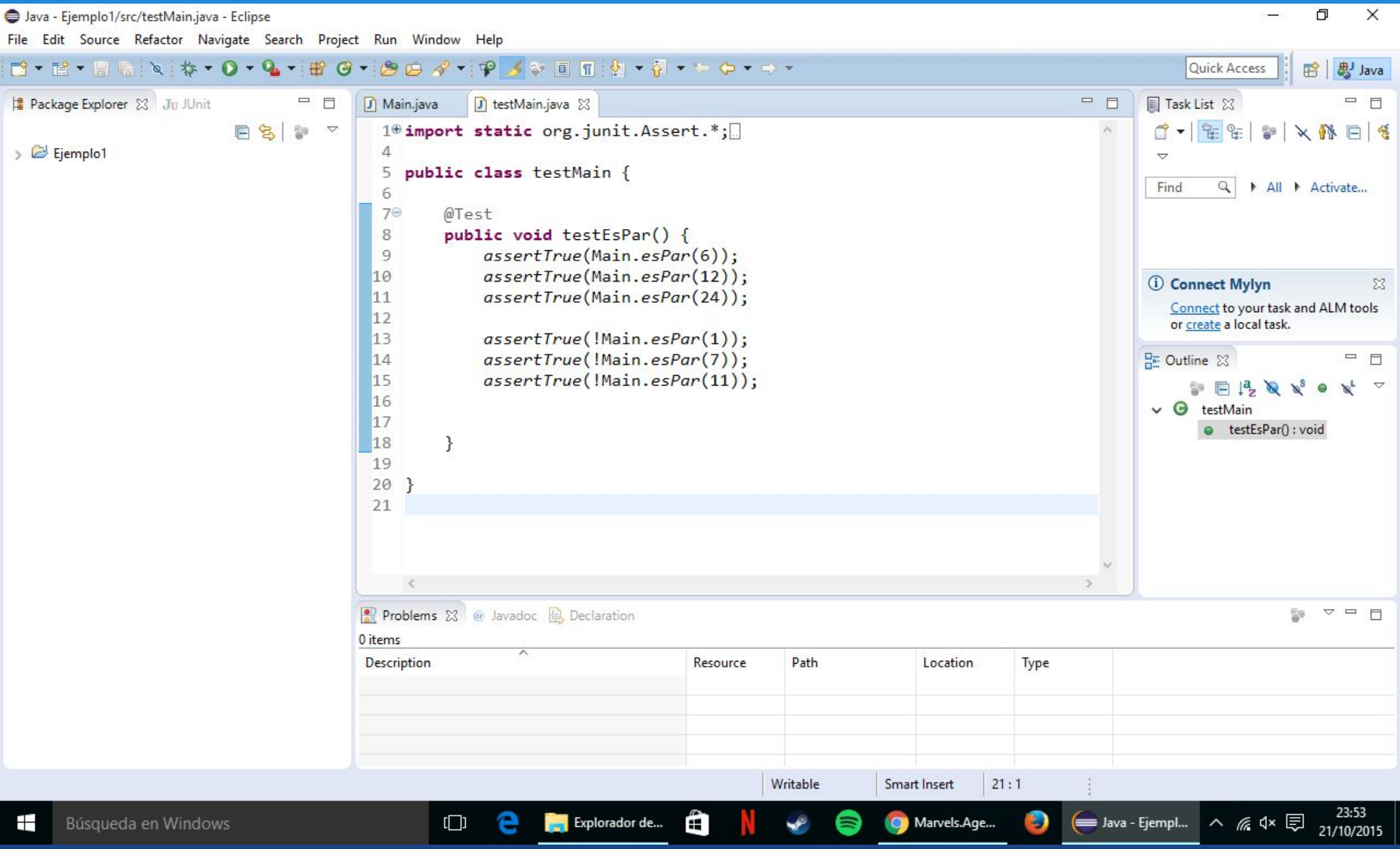
Verificación y Validación de Software



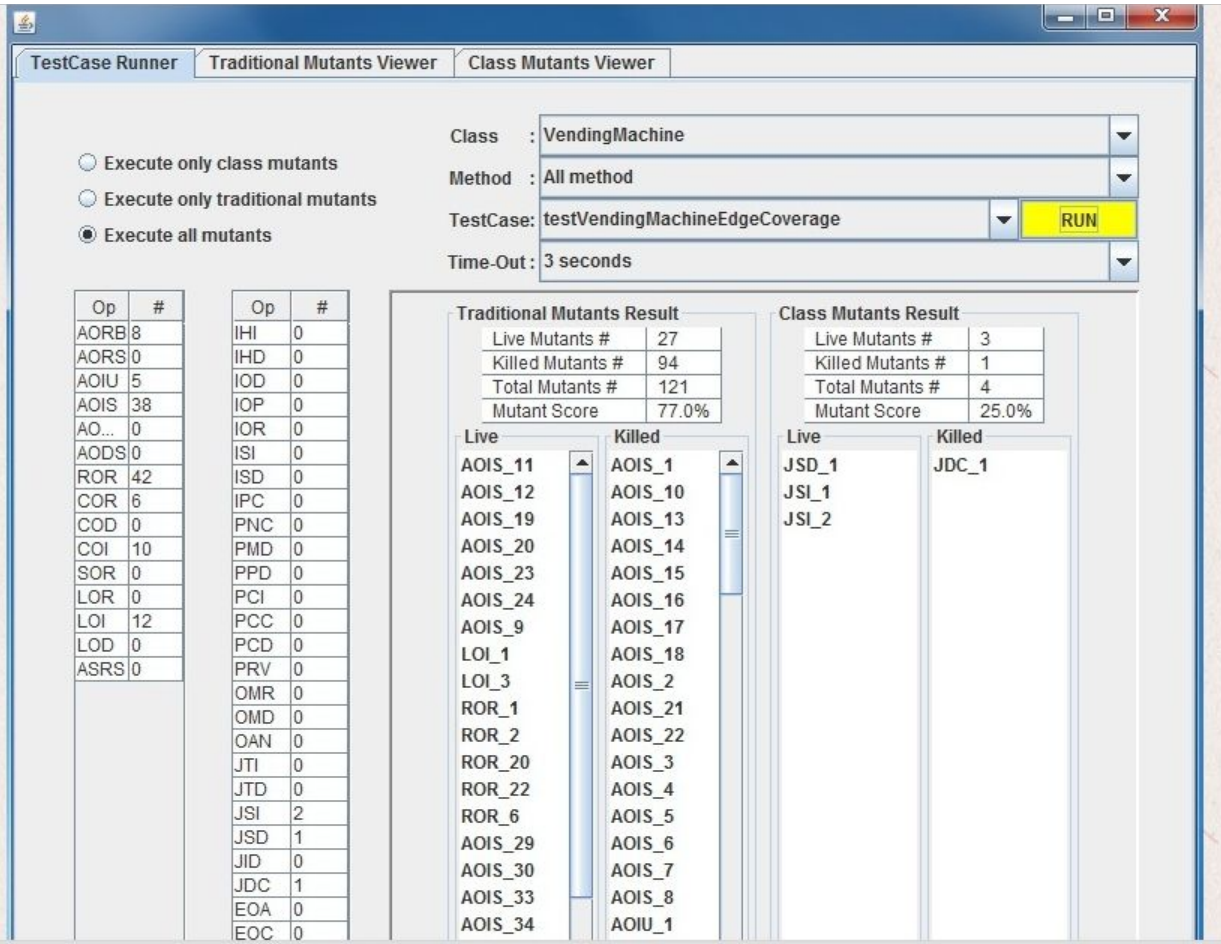
Verificación y Validación de Software



Verificación y Validación de Software



Verificación y Validación de Software



Verificación y Validación de Software

Estructuras de Datos
Martín Larrea