Verificación

Taller de Programación

Error, Defecto y Falla

un error humano

puede generar

Un defecto (interna)

que puede generar

?!

una falla (externa)

- El software **falla** cuando
  - No hace lo requerido o
  - Hace algo que no debería
- Se intenta detectar y corregir los **defectos** antes de la liberación del producto

Prueba

- Proceso de ejecutar software con el fin de provocar fallas (Myers)
  - Caso de prueba
    - Datos de entrada, condiciones de ejecución y resultado esperado
  - Conjunto de casos de prueba

Caja Negra y Caja Blanca

- Caja negra
  - No se usa la implementación
  - Los casos de prueba se generan sólo con la especificación
- Caja blanca
  - Se basa en la implementación
  - Los datos de prueba se generan a partir de la implementación
    - Luego se usa la especificación para obtener el resultado esperado

Ejercicio del Triángulo

- Un programa lee tres números enteros, los que son interpretados como representaciones de las longitudes de los lados de un triángulo. El programa escribe un mensaje que informa si el triángulo es escaleno, isósceles o equilátero.

- Escriban casos de prueba para esta especificación.

Discusión del Ejercicio

- Triángulo escaleno válido
- Triángulo equilátero válido
- Triángulo isósceles
- 3 permutaciones de triángulos isósceles 3,3,4 – 3,4,3 – 4,3,3
- Un caso con un lado con valor nulo
- Un caso con un lado con valor negativo
- Un caso con 3 enteros mayores que 0 tal que la suma de 2 es igual a la del 3 (esto no es un triángulo válido)
- Las permutaciones del anterior
- Suma de dos lados menor que la del tercero (todos enteros positivos)
- Permutaciones
- Todos los lados iguales a cero
- Valores no enteros
- Numero erróneo de valores (2 enteros en lugar de 3)

¿Todos los casos tienen especificada la salida esperada?

Técnicas de Caja Negra

- Existen diversas técnicas de caja negra
  - Partición en clases de equivalencia
  - Valores límite
  - Tablas de decisión
  - Grafos causa efecto
  - Pairwise testing
  - Basado en máquinas de estado
    - Método W
    - Todas las transiciones
    - Etc.
  - Basado en casos de uso
  - Etc.

Clases de Equivalencia

- Se **parte la entrada** en clases de equivalencia
- considerando que, para **cualquier dato** de la clase,
- el software se **comporta igual**

- Es el método más intuitivo

- Un método recibe un entero representando la edad de una persona y devuelve true si es mayor de edad y false si es menor de edad
  - Identificar clases de equivalencia

Ejercicio

- ¿Qué es ser mayor de edad?
  - Problema en la especificación que se puede detectar al momento de intentar generar casos de prueba (en otros momentos también)
- Supongamos que con 18 años o más se es mayor de edad
- Dividir en clases válidas e inválidas
  - Alfanuméricos (si se permite su ingreso)
  - Edades negativas
  - Edades positivas
    - Menores que 18
    - Iguales o mayores a 18
  - ¿Mayores que 140?

Valores Límite

- Usar los extremos de las clases de equivalencia
- Es normal encontrar más defectos en los extremos que en otros lados

- Edades negativas (-1, minint)
- Edades positivas
  - Menores que 18 (0, 17)
  - Iguales o mayores que 18 (18, maxint)

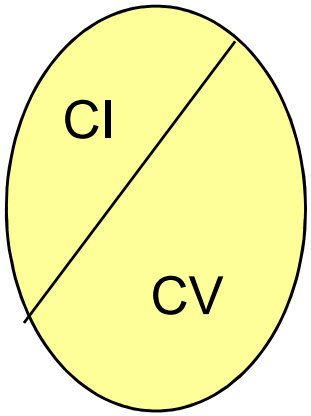
Múltiples Entradas Independientes

- Sistema para registro de horas trabajadas
  - Entrada: Número de cargo, tarea, horas
  - Salida: El registro se graba en la BD

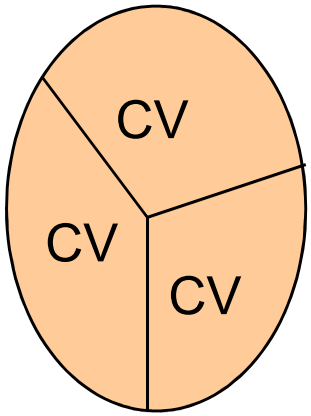
- Se parte cada uno de estos dominios en clases de equivalencia
- Se arman datos de prueba considerando
  - Sólo cubrir una clase inválida a la vez
  - Cubrir la mayor cantidad de clases válidas

Partición Sistema de Registro Horas

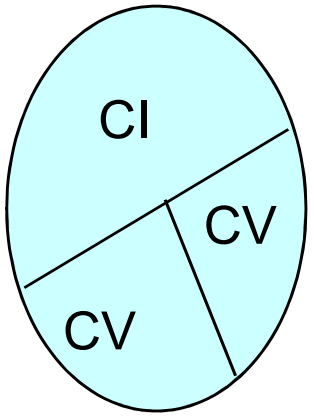
Partición de cada dominio de entrada en clases válidas e inválidas



Nro. Cargo



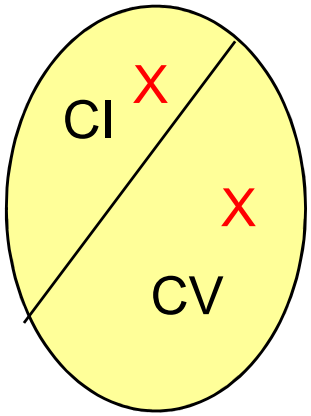
Tarea



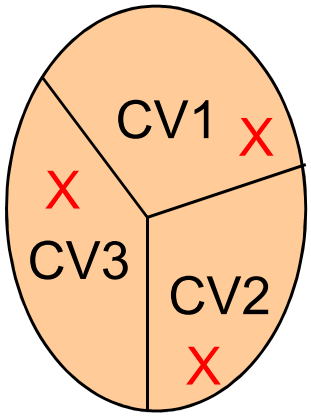
Horas

Partición Sistema de Registro Horas

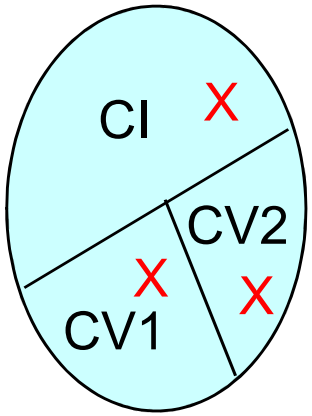
Selección de datos



Nro. Cargo



Tarea



Horas

Caso	Nro.Cargo	Tarea	Horas
1	CI	CV1	CV1
2	CV	CV2	CI
3	CV	CV1	CV1
4	CV	CV2	CV2
5	CV	CV3	CV2

Múltiples Entradas Dependientes

- En este caso las entradas no son independientes en lo que respecta al tratamiento de los datos por parte del sistema
- El triángulo es un caso claro
  - No importa sólo partir cada valor del lado en casos válidos y no válidos
  - Es importante la relación entre estas entradas
    - Por ejemplo: cuántas entradas son iguales
    - Esto se desprende del comportamiento

Atención: Si bien se parten las entradas, es muy importante basarse en el comportamiento y para esto trabajar con la especificación

Técnicas de Caja Blanca

- También existen diversas técnicas:
  - Basadas en el flujo de control
    - Sentencias
    - Decisión
    - Condición/decisión
    - Condición múltiple
    - Trayectorias linealmente independientes
    - Etc
  - Basadas en el flujo de datos
    - Todos los c-usos
    - Todos los p-usos
    - Todos los caminos definición-uso
    - Etc
  - Basadas en mutantes

Cubrimiento de Sentencias

- ¿Cuántas y cuáles sentencias cubrimos al ejecutar el conjunto de casos de prueba?
- El cubrimiento de sentencias se cumple cuando se ejecutan el 100% de las sentencias, al menos una vez, al ejecutar el conjunto de casos de prueba
- ¿Qué puede ocurrir si esto no se cumple?
- Este cubrimiento es de Caja Blanca

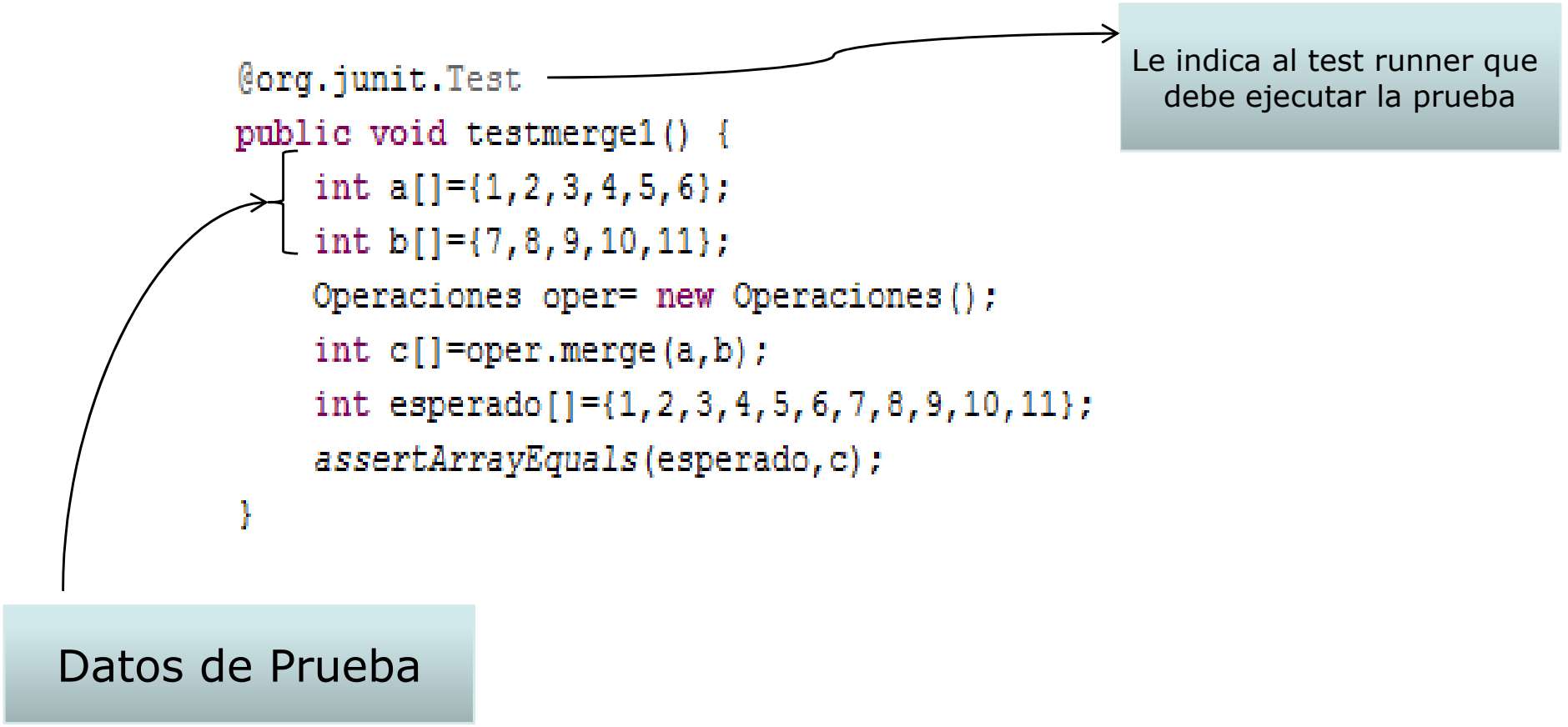
Ejemplo

```
public class Operaciones {
    public int[] merge( int[ ] a, int [ ] b) {
        int i= 0;
        int j = 0;
        int k = 0;
        int c[] = new int[a.length + b.length];
        while( i < a.length && j < b.length ){
            if( a[i] < b[j]){
                c[k]=a[i];
                k++;
                i++;
            }else{
                c[k]=a[j];
                k++;
                j++;
            }
        }
        for (int iter=i;iter<a.length;iter++){
            c[k]=a[iter];
            k++;
        }
        for (int iter=j;iter<b.length;iter++){
            c[k]=b[iter];
            k++;
        }
        return c;
    }
}
```

Ejemplo (2)

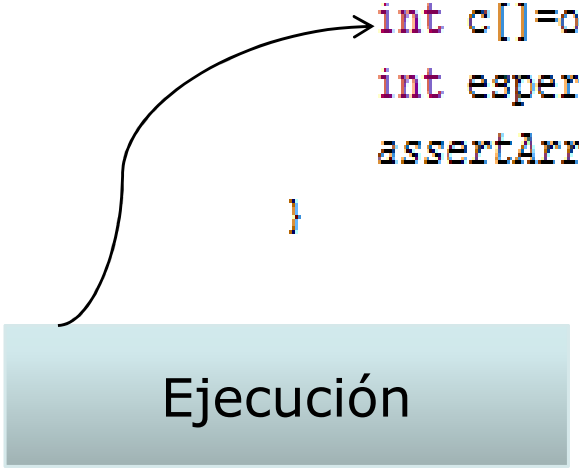
- JUnit
  - Vamos a presentar JUnit como herramienta para ejecutar casos de prueba unitarios.
  - Versión usada 4.4
  - JUnit da soporte a pruebas de caja negra.
  - Permite automatizar las pruebas.
- EclEmma
  - Vamos a presentar EclEmma para conocer el cubrimiento de sentencias alcanzado.
  - Versión usada 1.3.2

Prueba en JUnit



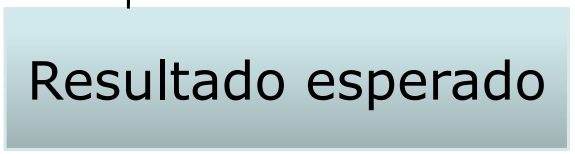
Prueba en JUnit (2)

```
@org.junit.Test
public void testmerge1() {
    int a[]={1,2,3,4,5,6};
    int b[]={7,8,9,10,11};
    Operaciones oper= new Operaciones();
    int c[]=oper.merge(a,b);
    int esperado[]={1,2,3,4,5,6,7,8,9,10,11};
    assertArrayEquals(esperado,c);
}
```



Prueba en JUnit (3)

```
@org.junit.Test
public void testmerge1() {
    int a[]={1,2,3,4,5,6};
    int b[]={7,8,9,10,11};
    Operaciones oper= new Operaciones();
    int c[]=oper.merge(a,b);
    int esperado[]={1,2,3,4,5,6,7,8,9,10,11};
    assertArrayEquals(esperado,c);
}
```

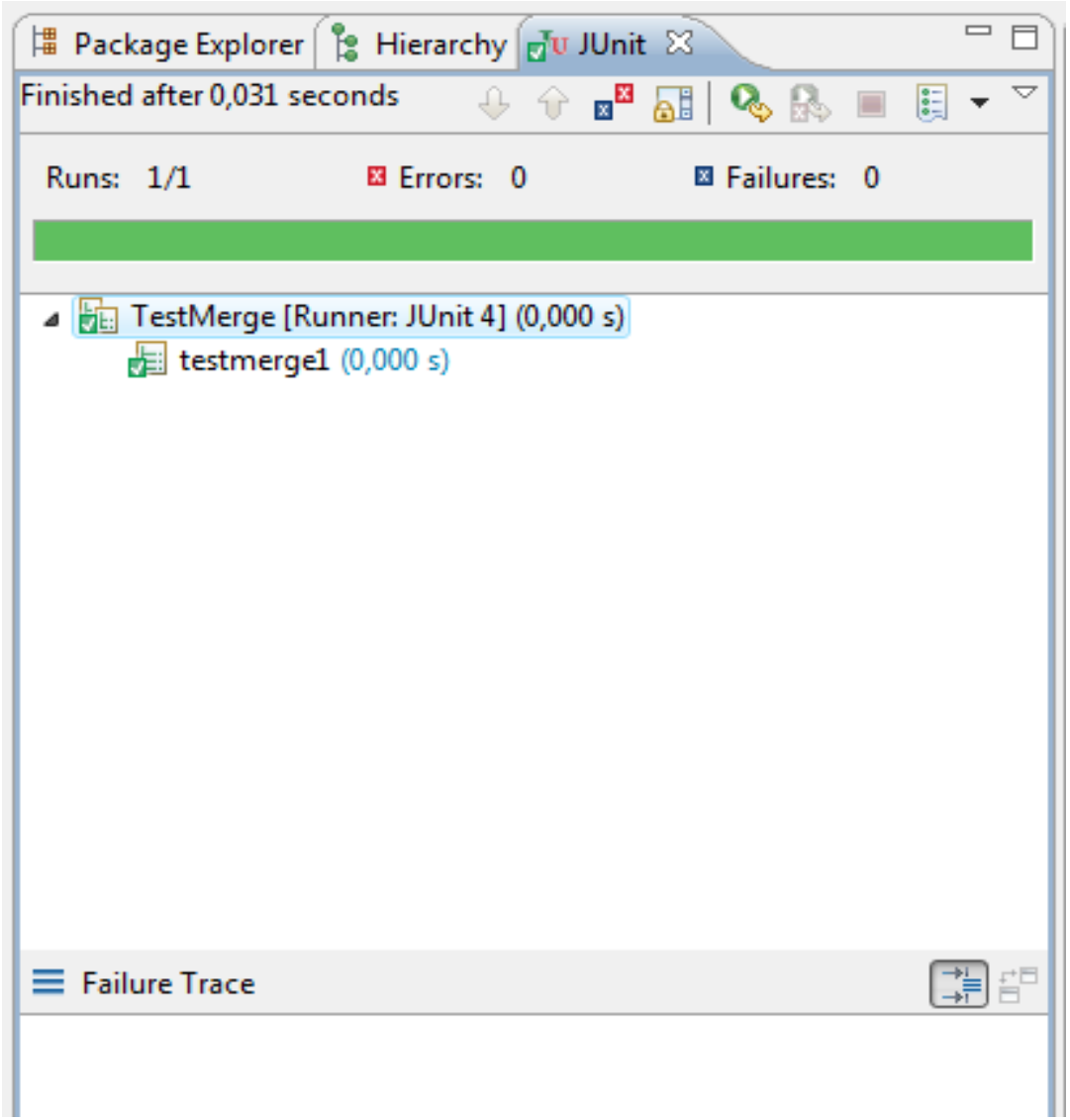


Prueba en JUnit (4)

```
@org.junit.Test
public void testmerge1() {
    int a[]={1,2,3,4,5,6};
    int b[]={7,8,9,10,11};
    Operaciones oper= new Operaciones();
    int c[]=oper.merge(a,b);
    int esperado[]={1,2,3,4,5,6,7,8,9,10,11};
    assertArrayEquals(esperado,c);
}
```

Comparación
Resultado de la prueba (pasa o falla)

Resultado en JUnit



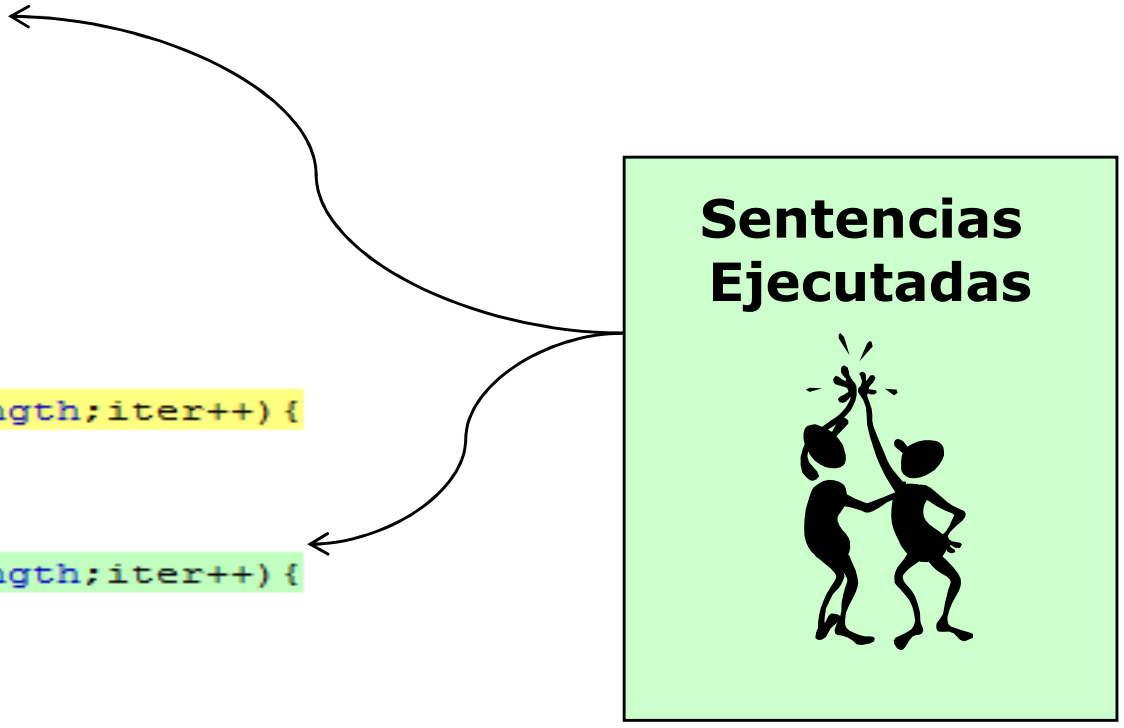
Cubrimiento con ECLEmma

```
public class Operaciones {
    public int[] merge( int[ ] a, int [ ] b) {
        int i= 0;
        int j = 0;
        int k = 0;
        int c[] = new int[a.length + b.length];
        while( i < a.length && j < b.length ){
            if( a[i] < b[j]){
                c[k]=a[i];
                k++;
                i++;
            }else{
                c[k]=a[j];
                k++;
                j++;
            }
        }
        for (int iter=i;iter<a.length;iter++){
            c[k]=a[iter];
            k++;
        }
        for (int iter=j;iter<b.length;iter++){
            c[k]=b[iter];
            k++;
        }
        return c;
    }
}
```

Cubrimiento obtenido al ejecutar el caso de prueba testmerge1

Cubrimiento con ECLEmma

```
public class Operaciones {
    public int[] merge( int[ ] a, int [ ] b) {
        int i= 0;
        int j = 0;
        int k = 0;
        int c[] = new int[a.length + b.length];
        while( i < a.length && j < b.length ){
            if( a[i] < b[j]){
                c[k]=a[i];
                k++;
                i++;
            }else{
                c[k]=a[j];
                k++;
                j++;
            }
        }
        for (int iter=i;iter<a.length;iter++){
            c[k]=a[iter];
            k++;
        }
        for (int iter=j;iter<b.length;iter++){
            c[k]=b[iter];
            k++;
        }
        return c;
    }
}
```



Cubrimiento con ECLEmma

```
public class Operaciones {
    public int[] merge( int[ ] a, int [ ] b) {
        int i= 0;
        int j = 0;
        int k = 0;
        int c[] = new int[a.length + b.length];
        while( i < a.length && j < b.length ){
            if( a[i] < b[j]){
                c[k]=a[i];
                k++;
                i++;
            }else{
                c[k]=a[j];
                k++;
                j++;
            }
        }
        for (int iter=i;iter<a.length;iter++){
            c[k]=a[iter];
            k++;
        }
        for (int iter=j;iter<b.length;iter++){
            c[k]=b[iter];
            k++;
        }
        return c;
    }
}
```

Sentencias No Ejecutadas

Cubrimiento con ECLEmma

```
public class Operaciones {
    public int[] merge( int[ ] a, int [ ] b) {
        int i= 0;
        int j = 0;
        int k = 0;
        int c[] = new int[a.length + b.length];
        while( i < a.length && j < b.length ){
            if( a[i] < b[j]){
                c[k]=a[i];
                k++;
                i++;
            }else{
                c[k]=a[j];
                k++;
                j++;
            }
        }
        for (int iter=i;iter<a.length;iter++){
            c[k]=a[iter];
            k++;
        }
        for (int iter=j;iter<b.length;iter++){
            c[k]=b[iter];
            k++;
        }
        return c;
    }
}
```

El arreglo a tiene todos sus elementos menores que los elementos del arreglo b. Por eso nunca se entra al primer else ni primer for.

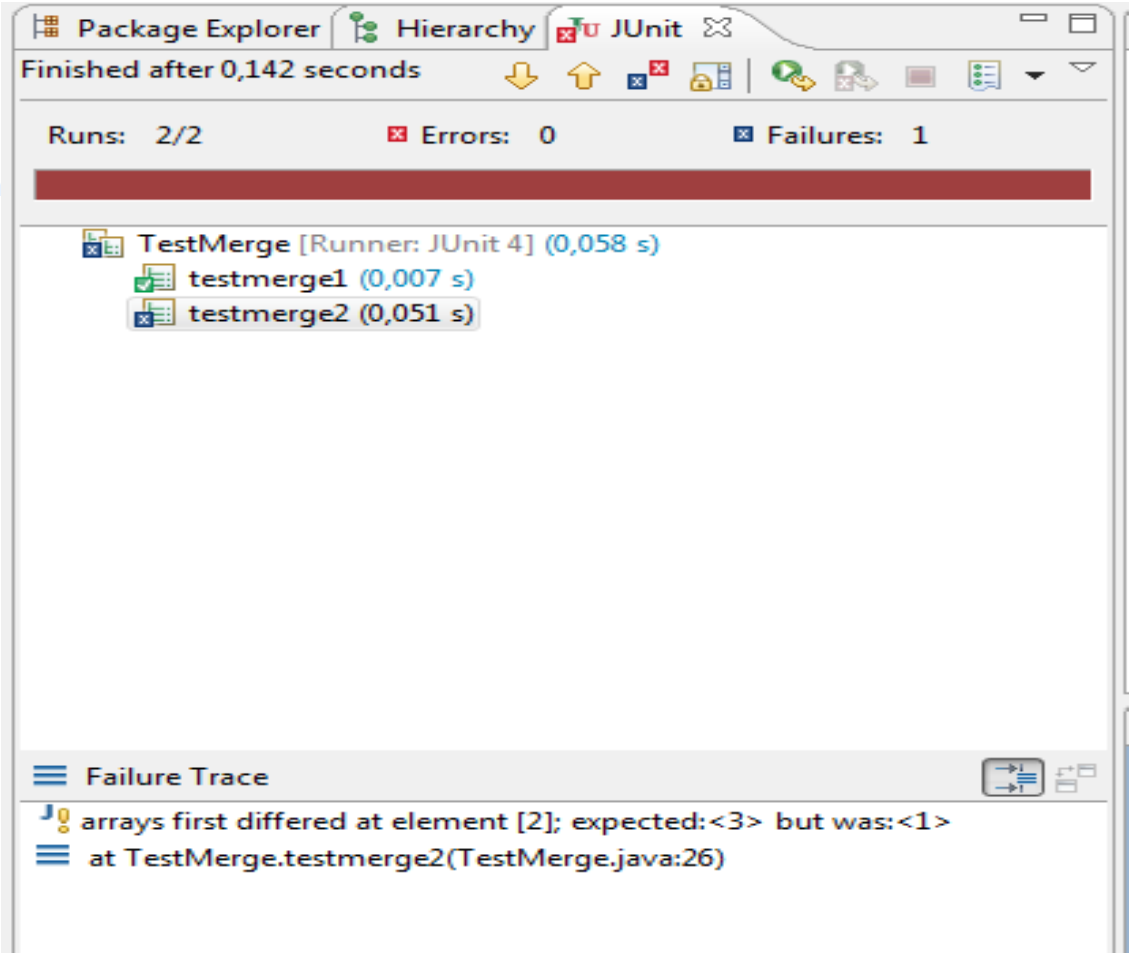
Cubrimiento con ECLEmma

```
public class Operaciones {
    public int[] merge( int[ ] a, int [ ] b) {
        int i= 0;
        int j = 0;
        int k = 0;
        int c[] = new int[a.length + b.length];
        while( i < a.length && j < b.length ){
            if( a[i] < b[j]){
                c[k]=a[i];
                k++;
                i++;
            }else{
                c[k]=a[j];
                k++;
                j++;
            }
        }
        for (int iter=i;iter<a.length;iter++){
            c[k]=a[iter];
            k++;
        }
        for (int iter=j;iter<b.length;iter++){
            c[k]=b[iter];
            k++;
        }
        return c;
    }
}
```

No se ha ejecutado completamente. En particular no ha incrementado a "iter"

Agregar una Prueba

```
@org.junit.Test
public void testmerge2() {
    int a[]={1,2,5};
    int b[]={3,4,8,10};
    Operaciones oper= new Operaciones();
    int c[] =oper.merge(a,b);
    int esperado[]={1,2,3,4,5,8,10};
    assertArrayEquals(esperado,c);
}
```



Encontrar el Defecto

- Se ha provocado una falla mediante una prueba
- Ahora hay que encontrar el defecto
  - Revisar el código y/o
  - Hacer “debug” del código
  - basándose en los datos de entrada del caso de prueba que falló
- Luego de encontrar el defecto
  - Corregir
  - Realizar pruebas de regresión

Defecto y Corrección

```
public class Operaciones {
    public int[] merge( int[ ] a, int [ ] b) {
        int i= 0;
        int j = 0;
        int k = 0;
        int c[] = new int[a.length + b.length];
        while( i < a.length && j < b.length ){
            if( a[i] < b[j]){
                c[k]=a[i];
                k++;
                i++;
            }else{
                c[k]=a[j];
                k++;
                j++;
            }
        }
        for (int iter=i;iter<a.length;iter++){
            c[k]=a[iter];
            k++;
        }
        for (int iter=j;iter<b.length;iter++){
            c[k]=b[iter];
            k++;
        }
        return c;
    }
}
```

Error al hacer la asignación

Se cambia
c[k] = a[j];
por
c[k]= b[j];

Cubrimiento Luego de Ejecutar los 2 CP

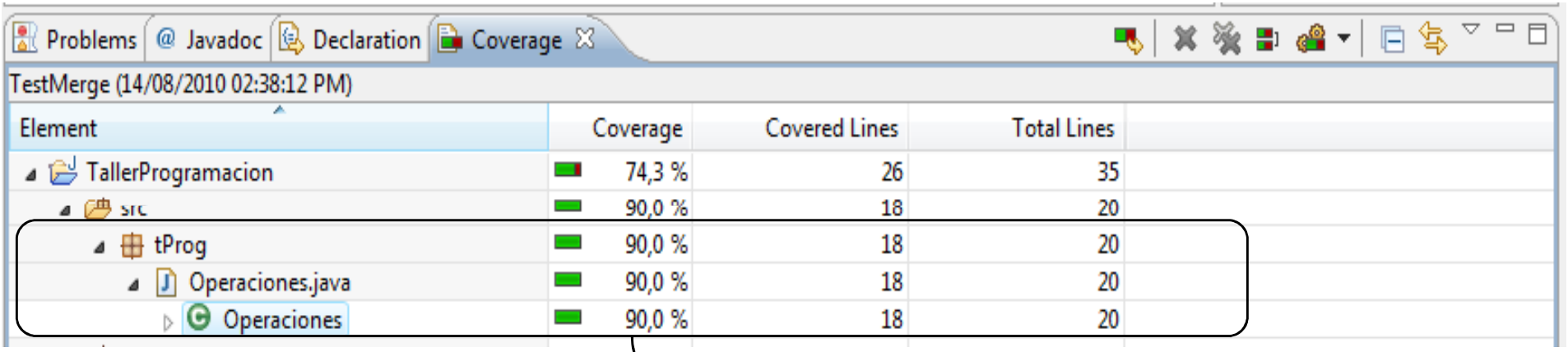
- Primero se ejecuta sin cubrimiento para ver si la corrección del defecto hace ejecutar el caso sin fallas

Sentencias No Ejecutadas

```
public class Operaciones {
    public int[] merge( int[ ] a, int [ ] b) {
        int i= 0;
        int j = 0;
        int k = 0;
        int c[] = new int[a.length + b.length];
        while( i < a.length && j < b.length ){
            if( a[i] < b[j]){
                c[k]=a[i];
                k++;
                i++;
            }else{
                c[k]=b[j];
                k++;
                j++;
            }
        }
        for (int iter=i;iter<a.length;iter++){
            c[k]=a[iter];
            k++;
        }
        for (int iter=j;iter<b.length;iter++){
            c[k]=b[iter];
            k++;
        }
        return c;
    }
}
```

¿Qué caso falta?

Cubrimiento Luego de Ejecutar los 2 CP



Porcentaje de cobertura obtenido luego de ejecutar los casos de prueba testmerge1 y testmerge2

Otro Caso Más

- Siempre el elemento más grande pertenece al arreglo b, por eso nunca se entra al segundo for.
- Se agrega el caso de prueba:

```
@org.junit.Test
public void testmerge3() {
    int a[]={1,2,5,11};
    int b[]={3,4,8,10};
    Operaciones oper= new Operaciones();
    int c[] =oper.merge(a,b);
    int esperado[]={1,2,3,4,5,8,10,11};
    assertArrayEquals(esperado,c);
}
```

Observación

- Se podría sacar el segundo caso
- Manteniendo el caso 1 y el 3
- Se obtiene el mismo cubrimiento

Algún Otro Conocimiento Interesante

- ¿Qué ocurre si cumplimos con el criterio de cubrimiento de sentencias?
  - Puede haber **trayectorias** particulares que igual contengan defectos
  - Puede haber **datos** particulares que igual provoquen defectos
  - Pero de todas formas ganamos más confianza en nuestro código y en nuestros casos
- Es imposible realizar pruebas exhaustivas
- La prueba (test) demuestra la presencia de faltas y **nunca su ausencia** (Dijkstra)

Sugerencias

- Revisar el código
  - Es cuando se detecta la mayor cantidad de defectos
  - Es bueno detectarlos tempranamente
- Construir casos de caja negra
  - La funcionalidad del software es lo que hay que asegurar
- Ver el cubrimiento alcanzado
  - Conocer qué tan buenos son mis casos
- Asegurar cubrimiento
  - Complementar con casos faltantes para asegurar cierto criterio de caja blanca

Sugerencias 2

- ¿Qué hacer cuando queda poco tiempo?
  - Revisar, construir casos de caja negra, luego ejecutar, corregir (varias veces), ejecutar regresión (varias veces), ver cubrimiento alcanzado, construir nuevos casos, ejecutar, corregir (varias veces), ejecutar regresión (varias veces), ver cubrimiento alcanzado… **Lleva mucho tiempo, mejor sólo codificar.**
  - ¿Han contado alguna vez el costo de retrabajo?
    - No se olviden que las clases que no verifico hoy son las que van a fallar mañana. Incluso, van a fallar al momento de ejecutar otra clase que la “usa” y va a ser difícil y muy costoso ubicar el defecto.

Lectura Complementaria

- http://www.junit.org/
  - Cuidado, la sintaxis de las versiones anteriores a la familia 4.x es diferente.
  - JUnit Cookbook: http://junit.sourceforge.net/doc/cookbook/cookbook.htm
  - Junit 4.x how to http://pub.admc.com/howtos/junit4x/junit4x.pdf
  - JUnit Test Infected: Programmers Love Writing Tests
    - http://junit.sourceforge.net/doc/testinfected/testing.htm (ojo! versión 3.8)

- http://www.eclemma.org/
  - En el menú de la izquierda
    - Instalación
    - Manual de usuario